AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej

Praca dyplomowa

Autonomous mobile manipulation robot
Autonomiczny mobilny robot manipulacyjny

Autor: *Maciej Stępień*
Kierunek studiów: *Automatyka i Robotyka*
Opiekun pracy: *dr inż. Łukasz Więckowski*

Kraków, 2022

Contents

1. Introduction

Nowadays robots are becoming more popular not only for industrial applications, but also for everyday life problems. Personal assistant type robots that can help with chores around the house are already here, but they aren't that useful yet.

On the mechanical side these robots are close to being practical, but there's still a lot to do on the side of the software - that's why platforms for research can hold great value. Robots like these exist, but they have one huge drawback - they are really expensive and only few of the top universities can afford them. In this thesis I would like to present a process of developing a low cost robotic platform that will be able to navigate autonomously around a house-like environment and pick up objects.

Figure 1.1. Finished robot created as a part of the thesis

Main focus of this research is on creating the robot - starting from the mechanical construction, through electronics, low-level logic and finally higher level software for autonomous behavior. Apart from this I will also present approaches taken and discuss how modern, mostly open source tools can accelerate this process.

1.1. Related work

One of earlier examples of personal robot research platforms is PR1 [1] developed at Stanford University under the Personal Robots program. In their work authors created robot with enough power to execute useful tasks, also making it safe to operate around humans.

To achieve mobility authors used a differential driven base with two pneumatic driven wheels and two suspension casters. Using this kinematics configuration, instead of omnidirectional wheels, provided better reliability when driving over doorway thresholds or other small bumps present in real life scenarios. Base motors used had continuous torque of $6\,\mathrm{N\,m}$ and allowed the robot to drive with speeds up to $2\,\mathrm{m/s}$.

Two 7-DoF (degrees of freedom) arms, with $5\,\mathrm{kg}$ payload each, were mounted on the torso, which had additional two degrees of freedom - it was able to rotate (to approximate holonomic motion) and move up and down, increasing robot's manipulation capabilities. Each arm consisted of the following joints: shoulder pan and lift, upper arm rotation, elbow flexion, forearm rotation and wrist flexion and rotation. For safety reasons actuators used in PR1 were low-reduction and backdrivable, with an ability to control output force directly with current. Spring based gravity compensation system decreased required torque from actuators, which helped to lower necessary reduction. Using a spring system instead of mass counterbalance helped authors to reduce additional inertia, which makes robot more safe to use in collaboration with people.

PR1's successor PR2 [2] [3] was developed by Willow Garage and this version was available for purchase. Arms design was similar to the previous version, kinematics configuration of the mobile base changed. Four rotating caster wheels located in each of the robot's corners were introduced, which made the base omnidirectional. Because of this change torso rotation was no longer required, so only up and down movement was used. It is also important to mention that this robot is quite heavy ($220\,\mathrm{kg}$) and expensive (two armed version was available for purchase for 400 000$).

Apart from novel mechanical design, Robot Operating System was developed as a part of the Personal Robotics Program in Willow Garage and it is widely used in all sorts of robots nowadays.

Another example of a personal robot for research is Fetch[4]. It was built taking into account experiences from PR1 and PR2 and it was designed to be robust, but also even more cost effective than PR2. Mobile base configuration consisted of two brushless hub motors (differential drive kinematics) and four casters. Placing driven wheels in the middle and making the base round allows for in-place rotations. Unlike two previous examples, the robot has only one arm (with $5\,\mathrm{kg}$ payload). Authors argued that applications requiring two manipulators to complete a task are sparse. This construction has no gravity compensation system, as it would result in additional weight in the upper part of the robot and more mass

in the base will be necessary to keep the robot stable. When compared to PR2, the robot has much lower mass (133.3 kg). Another improvement was increased battery lifetime (designed to be sufficient for an 8 hour work day, while PR2 lasted about 2 hours).

1.2. Goals

Main goal of this thesis is to create a robot capable of autonomously driving up to the table in one location, detecting and picking up an object and delivering it to the other location.

Taking the research platform aspect of the construction into consideration, there should be easy access into substituting various parts of the autonomous software. When the focus is on developing better solutions for navigating around dynamic obstacles, one should only have to worry about that, not rewriting low level communication with motor drivers. Thus modular software architecture should be used.

Another presumption of this work is that robot should be relatively cheap and easy to make without professional tools.

Safety should also be taken into consideration - the intended environment of the robot is house, potentially with humans, so taking precautions is necessary. Platform should be safe to work with, not cause any damage to its environment and itself.

1.3. Contents

In this thesis I will describe the full design of the autonomous mobile manipulation robot, starting with the construction side in chapter 2. All electronics and low level logic will be described in chapter 3. These two chapters will sum up how platform was constructed, chapters 4, 5, 6 and 7 will focus on software that was developed on the robot. General structure of the system and important software elements will be introduced in chapter 4. Chapter 5 will focus on the mobile part of the construction that enables the robot to move around autonomously. It will contain information about configuration as well as final results achieved. Chapter 6 in a similar manner to chapter 5 describes configuration and results of the manipulation task. In chapter 7 manipulation and navigation systems will be combined to achieve service robot application, in which the robot will pick up and deliver bottle.

2. Mechanical Design and Components

In this section the process of selecting materials, components and design details will be discussed.

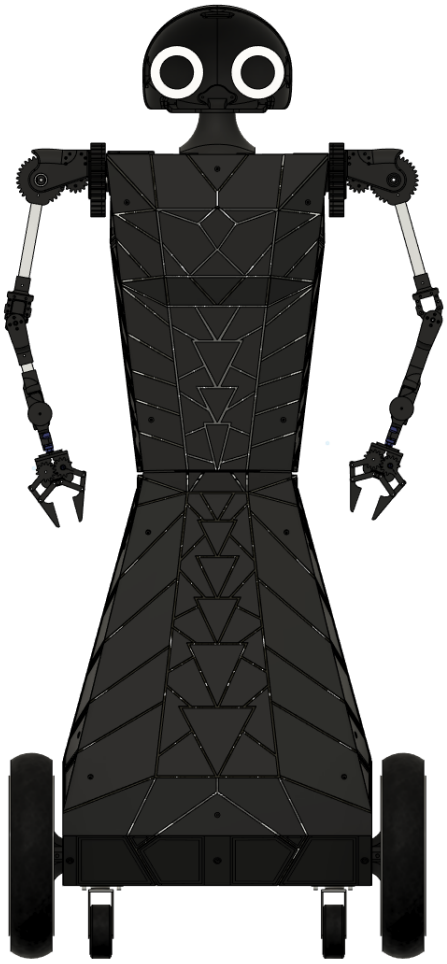

Figure 2.1. Full robot design in Fusion 360

2.1. Materials and manufacturing

To construct the robot I heavily relied on 3D printing - the main reason being that it allows making quite strong and precise custom parts and it is great for prototyping. There weren't any good alternatives

that could have been used - off-the-shelf parts would force many compromises, as they aren't any dedicated for this type of construction. Other types of producing custom elements usually require expensive equipment and ordering manufacturing in specialized companies would increase costs substantially. On the other hand 3D printing has many advantages for this use case: good quality 3D printers are within reach of the individual - being able to manufacture parts oneself instead of relying on external services really accelerates the process from design through prototypes to finished parts. Additionally materials used for 3D printing are also generally affordable.

As the material used for 3D printing I decided to use PLA - main advantages being that it is the cheapest and easiest one to print (it reduces time spent on troubleshooting when producing parts). Other options considered were PET and ABS, they usually are tougher to print with, but also more durable, especially when used outside. This robot was intended only for an indoor environment, I did not take into account these characteristics. Nylon is also worth mentioning, as it is really durable, but also expensive. I thought about using it for gears, but discarded it, because my printer wasn't able to achieve required temperatures.

Limitation to be considered when using a 3D printer is volume of the workspace - parts have to be designed in such a way that they will fit in that space. Creality Ender 3 3D printer that I used has a workspace dimensions $220\,\mathrm{mm}$ x $220\,\mathrm{mm}$ x $250\,\mathrm{mm}$, as it isn't that large, printing the whole construction would have been difficult and time consuming. So instead I decided to complement 3D printed parts with frame constructed with $15\,\mathrm{mm}$ aluminum square tubes - they are very light, rigid and easily available in most hardware stores. I also considered using aluminum extrusion instead of tubes, it has advantages that dimensions precision should be higher (as simple tubes from hardware store aren't intended for precise applications, some fine tuning is required to get fittings correct) and also mounting things is easier with screws and T-nuts, but the price is significantly higher as well. When considering dimensions precision it is best to purchase all necessary tubes at once - there is higher probability that they are from the same batch and closer in dimensions - I purchased some tubes from different shop and they differed slightly.

To connect tubes initially I considered welding, but as I didn't have equipment and skills necessary, I decided to 3D print connectors instead. To find good tolerances and create a tight fit I had to create a few test prints first.

In the initial prototype I also used one plywood element - a $12\,\mathrm{mm}$ thick square part was used as a base. During prototyping it was a fast way to create rigid construction and test placement of the components - it was flexible, because to some degree it was possible to move components around and screw them on again. After I was sure how everything should look like I redesigned the base with aluminum and 3D printed parts.

Figure 2.2. Frame design

2.2. Base design

This section is dedicated to the base that makes robot mobile.

2.2.1. Base kinematics

As a kinematics solution for the base of the robot I decided to use a differential drive. It is a good choice for the house-like environment as it makes robot quite agile and allows it to turn in place. Other common kinematics types for indoor robots utilize omni or mecanum wheels, this solution adds another degree of freedom - robot is also able to move in the y axis. It wasn't the best option for this construction though, as such wheels are more expensive, they have more difficulties with bumps on the ground and they also require more driven wheels - additional motors increase costs.

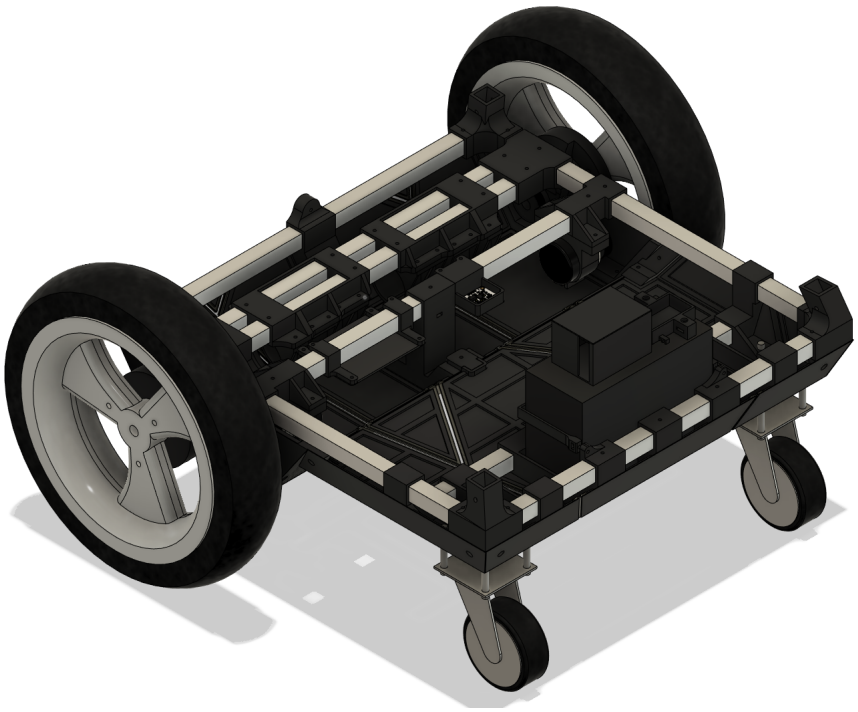

Figure 2.3. Base design

2.2.2. Wheels

When choosing wheels I made a few assumptions: first of all I preferred them to be pneumatic, as it adds some damping. Disadvantage of this solution is that it requires maintenance (inflating inner tubes). Second assumption was the size of the wheels, which I decided to be around $30\,\mathrm{cm}$ in diameter. I considered the ability to traverse over bumps - for example carpet, I also wanted bigger wheels for aesthetic reasons. All the assumptions were met by baby-carriage wheels and additionally they were fairly cheap.

Apart from two main drive wheels, there are also two caster wheels in the back used for stability reasons.

2.2.3. Motors

First step in choosing motors was calculating the required minimal torque.

As this was an early step in robot's design, all the values for calculations had to be approximated, I overestimated them as it was better to choose a stronger motor. Assuming the robot will only operate on flat surfaces, the torque that the motor had to produce was calculated using equation 2.1.

$$T = \frac{1}{2} rma \tag{2.1}$$

,where

- $\frac{1}{2}$ – there are two drive motors and this is torque calculation for one motor
- r – wheel radius ($0.145\,\mathrm{m}$)
- m – robot's mass ($20\,\mathrm{kg}$)
- a – desired acceleration ($2.5\,\mathrm{m/s^2}$) - it will allow the robot to stop when going full speed ($0.5\,\mathrm{m/s}$) within $5\,\mathrm{cm}$.

Substituting values torque is equal to

$$T = 3.625Nm \tag{2.2}$$

That is the minimum torque required to move the robot on a flat surface, I used it to have a starting point in choosing a motor. Preferred value is higher - it will then allow the robot to also travel over some bumps, for example carpet.

Second parameter to consider when choosing a motor is rotational speed. For my intended application I assumed that the robot should be able to achieve $0.5\,\mathrm{m/s}$ linear speed. Taking into account wheel diameter, required motors RPM (Revolutions Per Minute) is about 33.

Drill motors with gearboxes met these requirements regarding torque and rotational speed and they were the most affordable option. The model used in my robot has a listed torque of $24\,\mathrm{N\,m}$ and $350\,\mathrm{RPM}$. I chose it, mainly because it was the cheapest option that met minimum required criteria. Torque is a lot bigger than required, but it should be also taken into account that it is not an industry grade product, so it is safe to assume that this measurement shouldn't be exactly trusted and advertised value may be overestimated.

Additionally it is most likely stall torque, as it isn't precisely specified. In this case, assuming linear relationship between torque and speed, stall torque of $24\,\mathrm{N\,m}$ and no load speed of $350\,\mathrm{RPM}$ (it also isn't specified if it is no load speed, but it is worst case scenario) torque for maximum speed of $33\,\mathrm{RPM}$ should be equal to $21.74\,\mathrm{N\,m}$. But I decided that with that large margin from minimum required torque they will be alright.

To transmit power from drill motors to wheels I used an attached chuck, which made it fairly easy to fasten the axle. It isn't perfect though, as it sometimes becomes loose and requires tightening. One of the possible improvements could be making this connection more sturdy. As an axle I used a $10\,\mathrm{mm}$ aluminum rod, filed into a triangular shape with rounded corners on the end mounted to chuck, to provide better grip. On the side of the wheels I designed a hub with set screws that allowed mounting the wheel on the axle (axle was also flattened in places where set screws are). Additional bearings between the motors and the wheels transferred perpendicular loads.

Downside of the drill motors used is that they have high gear ratio (three stage planetary gearbox) which makes it impossible to back drive them, so during transport (when the robot is not powered on) it has to be lifted up. At the time that I was starting the construction direct drive or low reduction motors weren't really available with accessible prices, so they weren't a valid choice. Since then electric scooters became really popular and hub motors used in them became available at quite low prices and they can be a great alternative to drill motors.

Another drawback of motors used is high backlash, around $24°$, which equals to about $6\,\mathrm{cm}$ of the robot's translational movement. This makes control of the robot more difficult, when the robot stops with high deceleration it can "bounce back" and move backward due to backlash. Thus, to achieve higher precision in position control, it was necessary to limit deceleration value.

As drill motors don't have any form of feedback additional encoders were necessary. Timing belt transmission is used to connect the main driven axle with the encoder's shaft.

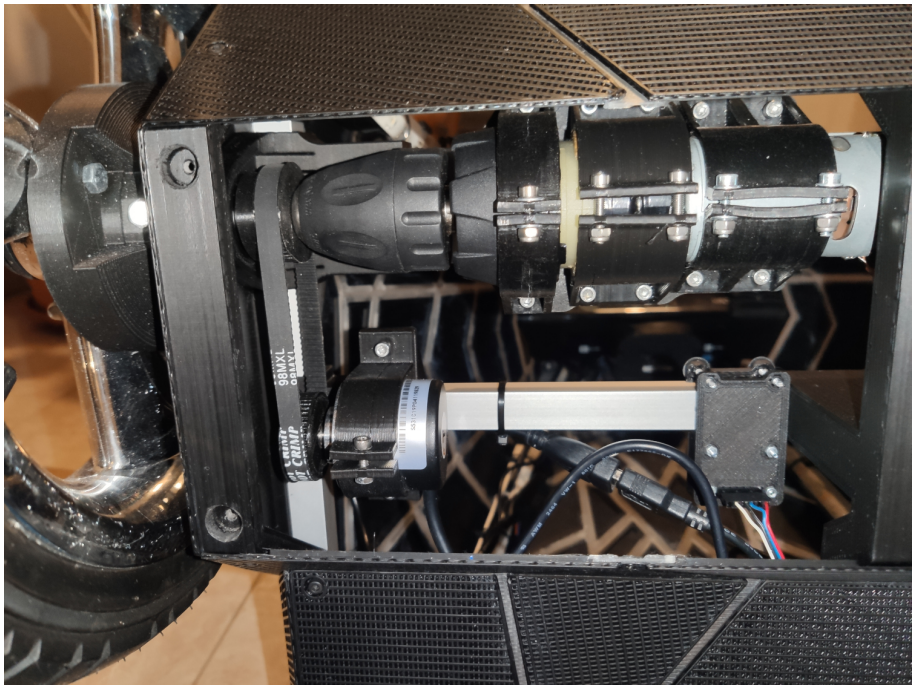

Figure 2.4. Timing belt transmission

2.2.4. Base construction

First constraints on this design was that it should be wide enough to place all the components and not too wide to be able to drive through door frames (the most narrow passages in the intended environment). I decided that the base width should be $42\,\mathrm{cm}$. It provided enough space for placing standard 15-inch screen laptop with some additional space for side USB connections and also for the drivetrain (drill motors used are quite long, so it was necessary to take that into consideration). When width of the wheels was added it summed to total $57.5\,\mathrm{cm}$ width of the robot, which was also narrow enough to satisfy the second constraint.

For the length of the base there wasn't an exact constraint, it should be long enough, so that wheels and casters could be mounted in such distance that the construction will be stable and not tip over, especially when decelerating. It also should allow for placement of the components, but they didn't require much space in that direction. I decided to make the base square, and also make it $42\,\mathrm{cm}$ long, as it appeared to provide enough stability.

Caster wheels were mounted in rear corners, moved slightly inward, so that mounting nuts were separated from the frame. It was more difficult to choose the right position for the front wheels. By placing them more in the front, the robot will become longer (some parts of wheels will exceed base) and less agile in tight environments. When the front wheels will be closer to casters, the robot will be less stable. Because of using plywood in initial design, changing this configuration wasn't difficult, so I decided on initial placement that also in terms of design looked best and tested it. My initial guess was performing well, so I decided to leave it that way.

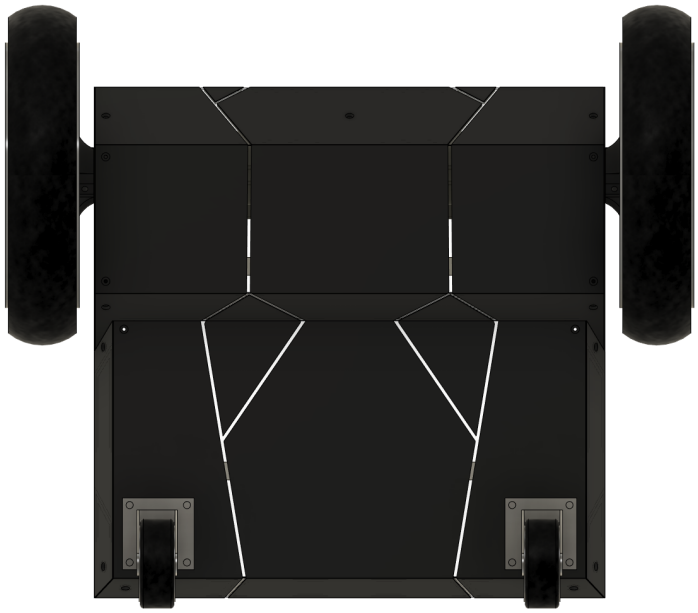

Figure 2.5. Wheels and casters placement

2.2.5. IMU mounting

When mounting an IMU it is really important not to put the board under any stress, as it will affect readings. To achieve it I used foam between mounting and PCB and also I didn't tighten screws too hard. Using foam has another benefit - it should help with vibrations.

2.3. Upper body

When deciding on the robot's height the main requirement was that it should be able to reach objects from tables. From this assumption I decided that the height of the arm should be about $100\,\mathrm{cm}$.

2.3.1. Kinect

As a main sensor I chose a Kinect RGB-D camera to detect obstacles and create a map of the environment.

Alternative considered was Intel Realsense D435 - it generally would be better for my robot. It has a lower minimum distance, is much smaller and doesn't need external power (only through USB). All these qualities would have made it easier to mount and use. Additionally, by using a stronger IR projector it is able to operate outside - Kinect has problems with direct sunlight outside, sometimes even inside (on sunny days). One crucial factor which swung in favor of Kinect was price, which was about ten times lower.

Mounting point was selected according to assumptions that it should be able to detect obstacles that are right in front of the robot, about at the line of end of the wheels. Following this assumption the best place was under the middle part of the robot.

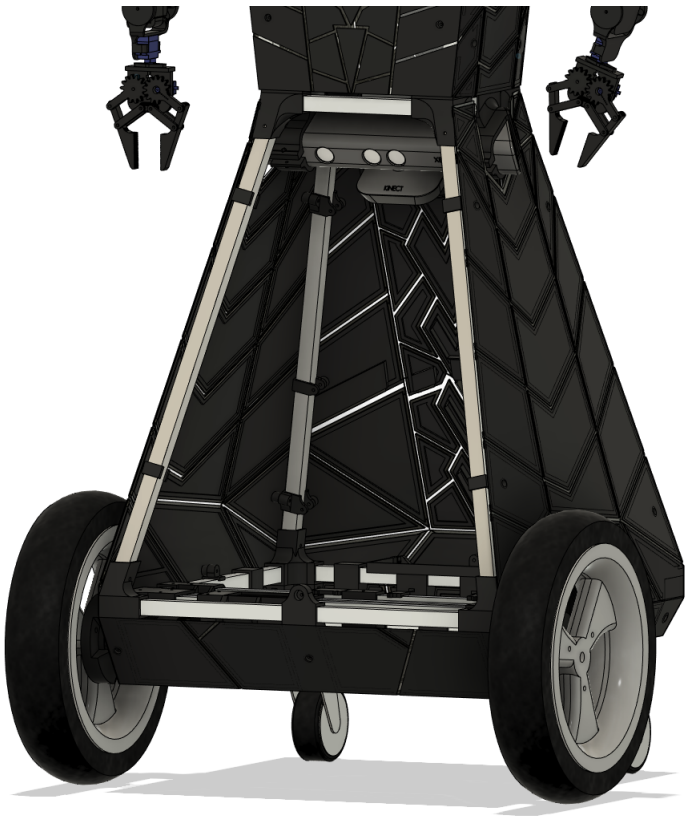

Figure 2.6. Kinect mounting

Due to the large size of the sensor, to mount it is necessary to unmount side parts of the body and front panel - I wasn't able to find a better workaround for this problem. It can be solved by changing the camera - a smaller Realsense D435 could be possibly embedded into the front panel, but it would substantially increase costs.

2.3.2. Arm design

First step in designing the arm was choosing kinematics configuration. I decided to use 5-DoF for the arm (with an additional one in the gripper), I found this configuration to be sufficient for the specified tasks that the robot should be able to complete. Increasing this number to 6 or even 7, as is used for industrial manipulators, would also increase complexity and costs, 5-DoF is a good balance between these aspects for this application.

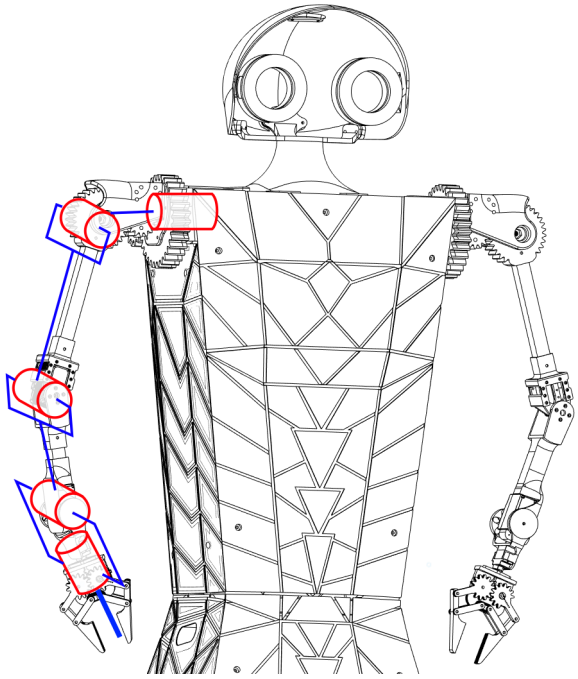

Figure 2.7. Arm kinematics diagram

Then I calculated the required length of the arm, approximate dimensions between joints and required torques. Total length of arm and division between joints were based on proportions to other body parts and ability to manipulate objects on table. To estimate torque mass of the servos was also necessary, so I started calculations from the gripper choosing servos simultaneously.

Gripper design was taken from a different robotic arm [5], its size allows it to pick up objects with size up to $3.9\,\mathrm{cm}$ which was sufficient for my application. It was already adapted to micro type hobby servos, my choice was TowerPro SG92R, as I already used them in my previous projects.

After choosing the gripper I estimated the torque necessary for the shoulder joint. It was quite rough estimate, as it was done still early on in the project, when I still considered using plywood for construction. Estimation was made assuming the worst case when the arm is fully extended perpendicular to the robot. Torque was calculated by dividing arm into few mass points, using these assumptions I derived equation 2.3.

$$\begin{aligned} T &= (r_{se}(2m_{ss} + m_{bs} + m_{mse}) + r_{ew}(m_{ss} + m_{bs} + m_{mew}) + r_g(m_g + 2m_{ms} + m_l))g \\ &= 2.20Nm \end{aligned} \tag{2.3}$$

,where

- r_{se} – distance to middle point between shoulder and elbow, $0.125\,\mathrm{m}$
- m_{ss} – standard servo mass, $0.06\,\mathrm{kg}$
- m_{bs} – bearings and screws mass, $0.05\,\mathrm{kg}$
- m_{mse} – mass of the arm materials for shoulder elbow(assuming constant density, they can be approximated by one mass point in the middle), $0.2\,\mathrm{kg}$
- r_{ew} – distance to middle point between elbow and wrist, $0.310\,\mathrm{m}$
- m_{mew} – mass of the arm materials for elbow wrist, $0.1\,\mathrm{kg}$
- r_g – distance to gripper middle point, $0.435\,\mathrm{m}$
- m_g – mass of the gripper, $0.13\,\mathrm{kg}$
- m_{ms} – mass of the micro servo, $0.015\,\mathrm{kg}$
- m_l – mass of the desired load, $0.1\,\mathrm{kg}$
- g – earth's acceleration, $9.81\,\mathrm{m/s^2}$

In my approach to choosing servos I used really rough estimates and then tried to test servos before committing to them. This approach isn't the best, but it has some strengths - it allowed me to choose servos quite fast. Servos that I used usually come in standard packaging, so replacing them doesn't require changing the whole project - for micro and standard servos that I initially chose there were more powerful alternatives available. XYZrobot servos also had alternatives, but not more powerful, so in this case it would be also required to redesign mounting.

Additionally all parts were quite cheap, so testing few options didn't come with much costs. Also cheap elements aren't necessarily well tested, so there could be discrepancies between documented value and real one, that's why it was better to also test them.

For gripper twist joint I decided on micro servo TowerPro SG92R and for wrist - TowerPro MG996R. I also tested another combination - TowerPro MG90S (gripper twist joint) and TowerPro SG5010 (wrist joint), but both of them were too weak. Symptoms of it were oscillations that happened due to high inertia of the rest of the arm - aggressive PID gains (which are not possible to be changed in hobby

servos) weren't able to stop in the correct spot and were overshooting. In both cases the problem was solved by using servos with higher torque output.

For the final 3 joints I chose XYZrobot A1-16 servos - according to specifications they were powerful enough for shoulder joints. Unlike hobby servos they use digital communication for control and provide many options for tuning operation. After initial testing though I found that the torque specified by the manufacturer wasn't exactly right and I was able to get a maximum of about $10\,\mathrm{N\,m}$. Later I found that the issue was PID controllers gains - for smaller changes in angle increase in current wasn't enough to actually make arm move and after some time with increased current draw servo overheated. During experiments I found settings that allowed for operating with higher torque, but it caused oscillations in some situations.

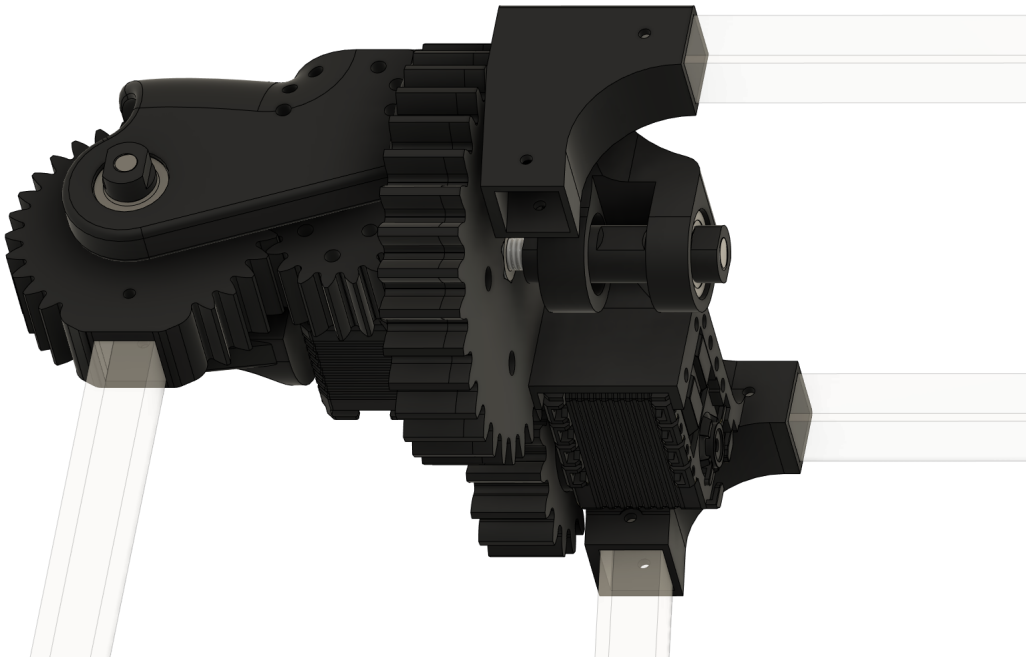

Figure 2.8. Shoulder design

To achieve higher torque I decided to use additional gear reduction. Using gears also helped with mounting arm, as it couldn't be placed directly on the motor hub - load would be too high. When choosing gear reduction, the limited operating range of the servos (330°) also had to be considered. Taking that into account 2:1 gear ratio for both shoulder pitch and roll joints seemed like a reasonable choice. 3D printing gears has proven to be quite difficult, I had to do some test prints to find correct tolerances even though gears were oversized (that way I was able to attach mounting parts of the arm directly onto gears). First problem that I encountered was so-called "elephant's foot". It happens when the printing bed is calibrated too close to the nozzle - then initial layers are squashed. My solution was using a raft - first pad is printed, then main part is printed on it, so first squashed layers are present only in the pad. Additionally I designed gears to be slightly shifted and printed on the opposite sides, so that bumps

will be present in the areas where gears aren't in contact. Finding correct tolerances for gears was also difficult and I had to settle for a solution with some backlash introduced.

Another problem during gear design was mounting with bearings. For simplicity I used a threaded rod, which didn't work that well with bearings. One thing was that they didn't fit tightly inside bearing holes and nuts, when tightened, were slightly angled, which introduced additional frictions. To get better results I decided to also print nuts and separator elements between bearings.

For the elbow joint I also used XYZrobot servo, this time I mounted the rest of the arm directly to the servo, as loads are significantly lower there.

2.3.3. Second arm

At the time of writing I decided to make only the right arm actuated. Left arm was added to complete the design, but joints were stiffened. The application that I was preparing for the robot didn't require two manipulators, actually many tasks can be completed using only one. Controlling two manipulators is also more challenging and actuating the second manipulator increases costs, which were quite big for the first arm. Additionally, the XYZrobot A1-16 servos that I used were discontinued and I couldn't buy them. It isn't that large of an issue though, because Dynamixel AX-12A servos should be able to replace them without any change in design, but still will be a minor inconvenience, so I decided to use only one manipulator.

2.3.4. Head

Head is not moving and its main purpose is to complete the appearance. Eyes are equipped with LED rings that allow to program some animations (for example blinking), which can be useful when we consider Human Robot interactions. When the robot has to perform some longer calculations - for example when finding optimal arm trajectory for picking up an object in an environment full of obstacles - I think that it is better to have some response, such as blinking eyes that can be a feedback that the robot is working. I also left mounting holes that can be used for mounting additional camera, I designed them with Realsense D435 in mind, but hadn't opportunity to use it.

2.4. Body design

To complete the design and also protect internal components I designed body covers in the form of panels. As they had to cover areas much larger than what was possible to create on my 3D printer, I divided them into smaller plates that were later on glued together to frames.

2.5. Dimensions and weight

Whole robot has a height of $132\,\mathrm{cm}$, $58\,\mathrm{cm}$ width and $50\,\mathrm{cm}$ length. Weight of the robot with the battery is around $16\,\mathrm{kg}$.

2.6. Safety

Overall I think that this robot is safe to work around humans. Due to low power actuators used it is not really possible for the robot to cause real harm. The most powerful actuators in the robot are XYZRobot servos and drill motors. The former ones overheat and turn off quite fast upon impact, for the latter ones it should not be possible to achieve the highest possible speed due to clamped maximum PWM signal in the firmware. Additionally, the robot is quite light ($16\,\mathrm{kg}$), so when moving it doesn't have much energy.

Another safety measure is an emergency stop button located at the back of the robot, that cuts off power when pressed.

One of the issues that needs addressing are open gearboxes in shoulder joints, especially the shoulder pitch one. It could be dangerous if something, for example hair, gets caught up in it, so I plan to add one more element that will shield it.

2.7. Software

For designing parts I used Fusion 360 CAD software, then for 3D printing I used Cura for generating G-code from STL files.

Using Fusion2URDF extension [6] allowed to directly generate files used later on in ROS. It greatly helped with development - I didn't have to manually read positions of the sensors and arm joints positions. It also allowed me to generate simulation model in almost no time.

2.8. Components list and cost estimate

Cost was an important variable during the construction, so I would like to present a brief estimate of the costs. These are prices that I paid for these elements, some of them already changed between purchasing them and time of writing of this thesis.

2.8.1. Electronic components

Part name	Quantity	Part cost [zł]	Total cost [zł]
LED ring	2	40	80
LED strip	1	135	135
Power on button	1	18	18
Emergency stop button	1	13	13
Voltage meter	1	8	8
XT60 connectors	2	6	12
HFKW-012-1ZW relay	1	13	13
DC-DC converters (step-down)	2	4	8
DC-DC converters (step-down) for LED strip	1	40	40
DC-DC converters (step-up/step-down)	4	11	44
Gel battery	1	48	48
Cytron SmartDriveDuo MDDS10 motor driver	1	137	137
STM32F103C8T6 development board	2	12	24
FTDI USB-UART converter	2	10	20
Level shifters	2	4	8
Angular USB cables	2	9	18
USB cable 1.5 m	1	15	15
Power cable	2	2 (per meter)	4
Cables	1	10 (per meter)	10
			Total cost: 655 zł

2.8.2. Mechanical construction

Part name	Quantity	Part cost [zł]	Total cost [zł]
Aluminum tube [1m]	9	13	117
Aluminum 10mm rod [1m]	1	11	11
Caster wheels	2	20	40
Black PLA filament	6.7 [kg]	49 (per kg)	329
Natural PLA filament	0.6 [kg]	49 (per kg)	30
Wheel	2	26	52
Bearings	1	11	11
Screws, nuts, threaded rods	1	33	33
Synchronous belt	2	10	20
			Total cost: 643 zł

Filament weights also include supports necessary for printing parts. Development of the parts was more costly, as some prototypes were necessary, here are included only weights of final parts.

2.8.3. Actuators

Part name	Quantity	Part cost [zł]	Total cost [zł]
Drills (for base motors)	2	70	140
XYZrobot Smart Servo A1-16	3	220	660
TowerPro MG996R	1	33	33
TowerPro SG-92R	2	17	34
			Total cost: 867 zł

2.8.4. Sensors

Part name	Quantity	Part cost [zł]	Total cost [zł]
Microsoft Kinect from Xbox 360	2	130	260
CNIKESIN Photoelectric Rotary Encoder	2	50	100
Bosch BNO055 IMU	1	200	200
			Total cost: 560 zł

2.8.5. Total cost

It is important to mention that the computing device used as a controller for autonomous operations isn't taken into account for cost estimation. Robot has dedicated space for placing laptop, which isn't really an integral part of the robot.

Total cost of the robot construction was about 2725 zł, which makes this platform really affordable, especially when compared with products currently available on market. Of course this estimation included only prices of parts, without the cost of assembling the robot and other costs that have to be included when considering production and distribution in larger quantities. It wasn't my intention to create a product, only a prototype that proves that it is possible to create a low cost construction. Also capabilities of the robot, when compared to more expensive construction, is quite low. Manipulator's maximum payload, number of DoF and reach makes it hard to apply it in some more practical situations, but I would argue that the ratio of robot's capabilities to price is really good.

3. Electronics

In this chapter I will describe electronic components of the robot and low level software.

3.1. Power source

As the main power source for the robot I decided to use a gel battery, as it has the best price to the capacity ratio and it is quite safe to use. Other considered options were LiIon or LiPol, but they are much more expensive. Disadvantage of the gel batteries is that they are heavy - they have the lowest capacity vs mass ratio (for LiIons and LiPols that is much better). It wasn't that big of a problem in my case though, it even had some positive sides - because of placing the battery as low as possible and in the back of the robot it improved stability.

I chose to use $12\,\mathrm{V}$ as it allows me to power drill motors directly from the battery, without the need for an additional voltage converter, which could be costly taking into account how much power these motors can draw. One disadvantage of this solution is that the voltage slightly changes with discharging.

Final parameter that I needed for choosing battery was capacity, when calculating it I assumed that robot should work for about 4 hours of typical usage, to estimate power draw I included only most energy consuming elements - the arm and base motors:

- drill motor $18.0\,\mathrm{W}$
- XYZrobot A1-16 $12.0\,\mathrm{W}$
- TowerPro SG5010 $5.4\,\mathrm{W}$ (I was initially using it, only later changing to TowerPro MG99R, but it has comparable power draw)
- TowerPro SG92R $1.5\,\mathrm{W}$

It is important to note that choosing the battery was quite early in the design phase, so all these values were approximated based on datasheets and rough measurements, which were made while parts were not mounted to the robot. I overestimated them, so the real power draw of the robot should be lower.

Assuming that robot will be either driving or executing manipulation tasks:

- driving $36.0\,\mathrm{W}$
- manipulation $44.4\,\mathrm{W}$

Based on these calculations and checking available options of batteries, their sizes and weights I chose to use 7 A h one (84 W h). In the worst case scenario it should be able to power about 2 hours of constant manipulation tasks. In typical usage I hoped to get 4 hours runtime with this battery - power consumption of each servo should be lower than estimate, usually not all servos move at once when executing trajectories and in typical usage arm movement isn't constantly executed. When running fetch object tests I typically got about 4-6 hours of work time.

3.2. Shutdown circuit

Between battery and other components I added intermediate step, shutdown circuit, which allows to disconnect power supply. It consists of a relay controlled by two buttons connected in series - normal power on button and emergency stop button. Relay used is in normally open configuration, when either of the buttons is in the off position or there is a problem with connection, power is cut from all robot's components.

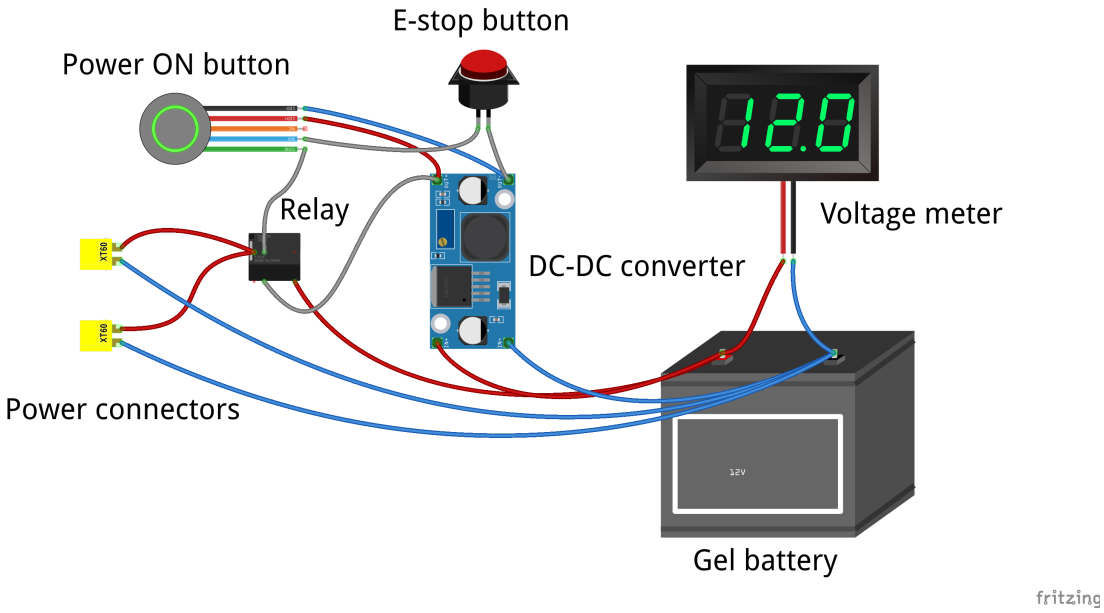

Figure 3.1. Shutdown circuit schematic

3.3. Base

3.3.1. Motors and motor controller

Drill motors used in the base are 12 V DC motors, to control them I chose a Cytron MDD10A dual channel driver based on a MOSFET H-bridge. It can drive motors with voltage up to 30 V and current draw 10 A (momentary 30 A), which is sufficient for this application. It is driven by the PWM signal and one digital signal, which controls whether the motor is rotating in positive or negative direction.

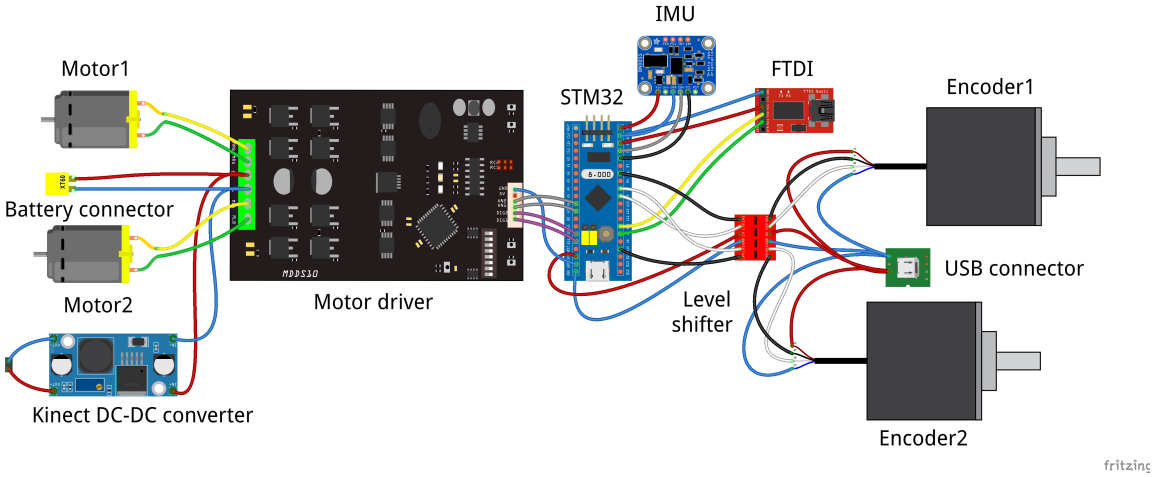

Figure 3.2. Base electronics schematic

This configuration is not safe, when initially I was programming a microcontroller still connected to the driver, PWM line would go high, resulting in motors going full speed. I was able to fix it by including a pull-down resistor on the PWM line. Better configuration is used for example in L293D H-bridge, which has an enable line that has to be high and additionally two input lines, which separately controls two output lines connected to the motor. In result it is necessary to provide 10 or 01 combinations on the inputs, which is much safer.

The motor driver has the possibility to change PWM control into exponential mode, which was very useful as the speed range required for this application was much lower than motors' available speed range. In result this mode allowed to achieve higher resolution of PWM control in operating speed range.

3.3.2. Encoders

To get feedback from wheel position, incremental optical quadrature encoders were used. When choosing resolution I was considering odometry accuracy, which I wanted to be below $1\,\mathrm{cm}$. With 100 ticks per resolution it is possible to detect 400 signal changes, taking into account the wheel's radius, it equals to the $2\,\mathrm{mm}$ translational shift. I made a mistake with this choice though, because I didn't take into account that the PID controller also used these encoders for feedback and in this case the selected resolution was quite low and it forced me to choose a lower PID frequency.

Output from this type of encoder consists of two signals, usually named A and B, that change between logical 0 and 1 values. They are shifted in phase with respect to each other, which allows to read not only changes in position, but also direction of these changes. To convert this information to angle I connected signals to the microcontroller and used external interrupts that detected raising and falling edges. On each interrupt checking the type of the edge and logical state of the other signal allowed me to decide if the tick counter should be increased or decreased. Two counters were used - one for the odometry and one for PID. I decided to set the counter to 0 after each angle calculation - this way overflowing the counter is

prevented and a smaller 8-bit variable can be used. As both applications work with different frequencies they needed separate counters.

After initial tests I found problems with noise induced by the motors, as it sometimes caused fluctuations high enough to be considered a change in signal, causing incorrect readings. Interestingly it happened only when one of the motors was reversing, which made the real problem harder to find. As encoder cables are shielded I didn't expect it to be caused by magnetic field, rather noise in the electric circuit, which I was able to confirm by connecting encoders to external power supply, as problems were no longer present then. I tried few filtration options, among them adding LC filter on the signal lines (to reduce high frequency noise), adding filtration to the motor terminals, adding filtration to encoders' power supply and changing DC-DC converter used to powered them, but none of it appeared to work. I was able to find a temporary workaround by using a separate power source for the encoders provided by USB connection. As a laptop, with external battery, was chosen as the main computing device, it wasn't a big problem, but it is only a workaround and as a improvement a proper solution should be found (such as galvanic insulation, which I haven't tried).

3.3.3. IMU

As the IMU sensor I chose Bosch's BNO055. Main advantage of this specific board is that it not only has sensors, but also an M0 processor that runs fusion algorithms on board. In result it is able to return accurate orientation, that otherwise would have been fused on the computer (which won't be that accurate, as Bosch's algorithm is well tuned for this board). In section 5.1 accuracy testing is described in more detail.

To communicate with the board I2C is used, for which I used a base microcontroller. Initially I had problems with it, one of the reasons being the high capacitance of the cables, which acted as a low pass filter, disrupting proper communication. I was able to solve this problem by lowering communication speed. Another issue was incorrect I2C interface initialization on the microcontroller, additional reset of the interface fixed this problem.

As an alternative to BNO055 I considered using one of the Phidgets devices, which are theoretically more accurate and have a USB interface with plug-and-play functionality, making it easy to use, but they are much more expensive as well. Another available alternative is MPU6050, which is cheaper, but from my experience also less accurate.

3.3.4. Microcontroller

To handle communication with motor driver, encoders and IMU I chose STM32F103C8T6 based development board, commonly known as a "Blue Pill", as it is cheap and quite powerful. It operates in $3.3\,\mathrm{V}$ and logic level shifters were necessary to connect it to the encoders and motor driver. FTDI USB-UART converter was used to provide communication with a computer.

For developing firmware I used *STM32Cube* with *HAL* library and *roslib* for communication with higher level software (which is also using ROS). Main tasks implemented on base controller are:

- reading signals from encoders
- IMU communication
- kinematics calculations
- PID controller for base motors

3.3.4.1. PID controller

When designing a PID controller for motors I had to choose what frequency it should work with. I considered that with too high PID frequency there will be problems with feedback for lower speeds - encoders chosen have quite low resolution and there won't be enough change in the wheel angle to have any feedback from encoders. On the other hand higher frequency allows to achieve faster responses to commands, which is also important in this application. Taking that into account I chose $24\,\mathrm{Hz}$ as a good compromise between these aspects. I decided to calculate speed based on encoder tick changes between iterations of the controller, so the estimated speed can be calculated following formula:

$$v = 2\pi r \frac{n}{N} f \tag{3.1}$$

,where

r – wheel radius ($0.145\,\mathrm{m}$)

n – number of encoder ticks

N – total number of encoder ticks per revolution (400)

f – frequency of calculation ($24\,\mathrm{Hz}$)

When considered a minimum amount of 1 tick per iteration it is possible to achieve the lowest speed of $0.055\,\mathrm{m/s}$. Because of placing calculations in the controller loop and sampling encoders' ticks with constant frequency error is introduced, assuming constant speed its value could be up to $0.055\,\mathrm{m/s}$. This reduces quality control, as a further improvement, calculating speed using time measurement between encoder ticks could be considered.

To choose PID controller gains I implemented an interface that allowed changing them in the real-time, which accelerated tuning. As the output speed feedback was sent to the computer where, also in real time, plots were created. Changing gains and plotting speeds was based on ROS subscribers and publishers, just like all other communication. First I did tuning with wheels lifted above the ground, based on speed responses to the $0.5\,\mathrm{m/s}$ setpoint step.

After initial testing I implemented some modifications to the original PID algorithm - I found that the minimum PWM signal required to start moving motors is quite big, which introduced delay when

beginning with stopped motors. Trying to reduce this delay by increasing P gain introduced oscillations, so my final solution was to offset response from PID by measured minimum PWM signal. Another modification was adding a minimum set speed, which helped with efficient motors stopping. Setting a threshold for the PWM value to be 1/3 of the maximum value improved safety, as even if there was an error in the controller, motors still couldn't go full speed.

It is also important that the controlled variable is not directly PWM value, but change of it, so the previous PWM value is stored and then modified based on the output from PID. Controlling PWM directly isn't correct, because with error close to 0, PWM value should be kept constant so that motors will rotate at the constant speed, not changed to 0 as it will stop motors.

Satisfactory performance was achieved with the P-only controller, so I only focused on tunning P gains. After initial tests with wheels lifted above ground were finished, I checked how it worked when the robot was driving. It performed worse as responses changed when introducing additional load, I then fine tuned them mostly by observing the robot's response to my input commands.

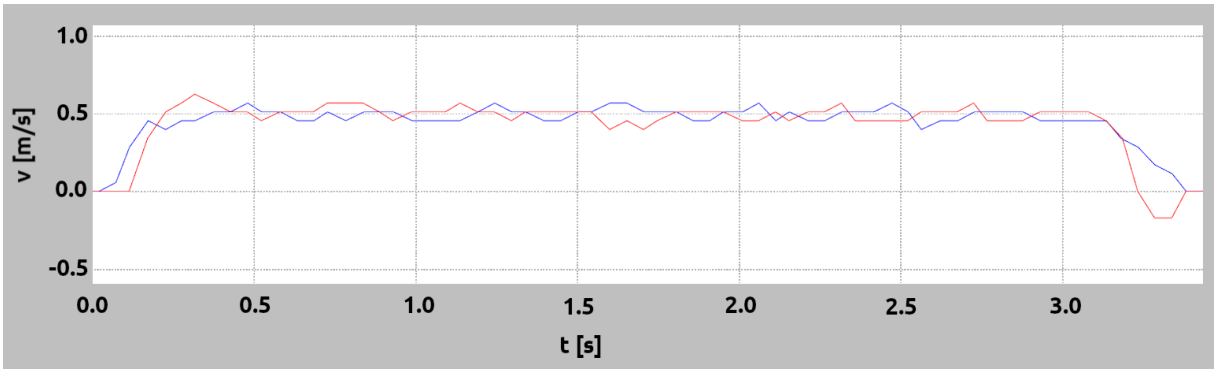

Figure 3.3. Result for P equal to 0.7, blue line represents right motor, red - left motor, setpoint is equal to $0.5\,\mathrm{m/s}$, wheels lifted above ground

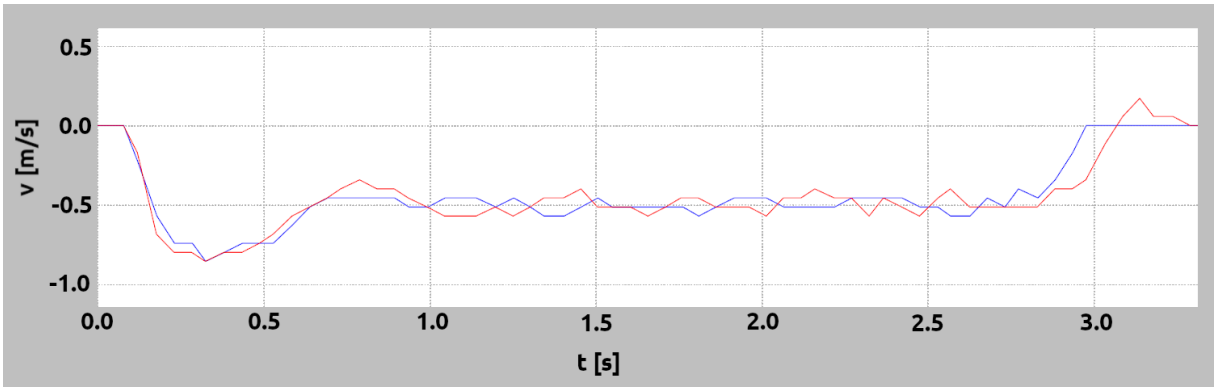

Figure 3.4. Result for P equal to 1.5, blue line represents right motor, red - left motor, setpoint is equal to $-0.5\,\mathrm{m/s}$, wheels lifted above ground

3.3.4.2. Kinematics calculations

Kinematics calculation was also implemented in the base controller. It consisted of the two parts: translating velocity commands and calculating odometry information. Equations were based on [7].

Translating velocity commands

The robot's base is controlled by velocity commands, which are in the form of linear and angular velocity of the robot. They need to be converted to the setpoints of the individual motors. To do that, equations 3.2 and 3.3 were used.

$$v_{r\_sp} = v_x + \frac{b}{2}\omega_z \tag{3.2}$$

$$v_{l\_sp} = v_x - \frac{b}{2}\omega_z \tag{3.3}$$

,where

- b – distance between wheels ($0.507\,\mathrm{m}$)
- v_x – desired linear velocity of the robot
- ω_z – desired angular velocity of the robot
- v_{r_sp} – setpoint for the right motor
- v_{l_sp} – setpoint for the left motor

Additionally timeout functionality was implemented - velocity commands have to be received with minimum $2.5\,\mathrm{Hz}$ frequency for the controller to work. It is useful to detect if communication wasn't lost and if that's the case the robot will stop.

Odometry calculations

Information in the form of encoder ticks needs to be transformed into position of the robot, defined as (x, y, θ). First encoder ticks were translated to the wheel angles:

$$\alpha = 2\pi\frac{n}{N} \tag{3.4}$$

,where

- α – wheel angle
- n – number of encoder ticks
- N – total number of encoder ticks per revolution (400)

Then to calculate robot's position:

$$x = x + \frac{r}{2}(d\alpha_l + d\alpha_r)cos(\theta) \tag{3.5}$$

$$y = y + \frac{r}{2}(d\alpha_l + d\alpha_r)sin(\theta) \tag{3.6}$$

$$\theta = \theta + \frac{r}{b}(d\alpha_r - d\alpha_l) \tag{3.7}$$

,where

- r – wheel radius ($0.145\,\mathrm{m}$)
- b – distance between wheels ($0.507\,\mathrm{m}$)
- $d\alpha_l$ – change of the left wheel angle
- $d\alpha_r$ – change of the right wheel angle

As a possible improvement ROS control [8] could be used - it provides useful mechanisms for low level motors control that is more standardized. Additionally it has already implemented plugins for differential base kinematics. So instead of calculating these values in firmware, there should be an interface that outputs current wheel angles and as input takes setpoint to the motors. All kinematics calculations will then be moved to a higher level computing device. In this case it will be able to replace current functionalities implemented in the base controller, but also add some other useful things such as velocity, acceleration and jerk limits.

3.3.5. Kinect power converter

Final component that is also present in the base electronic circuit is the power converter used as a power supply for Kinect. Kinect needs stable $12\,\mathrm{V}$ and the battery used has nominal $12\,\mathrm{V}$ voltage but it changes depending on current charge, so to achieve exact $12\,\mathrm{V}$ I had to use a buck-boost converter that was able to increase/decrease voltage depending on the situation.

3.4. Upper electronics

Main component of the upper part of the robot is manipulator, apart from it there are also LEDs.

3.4.1. Power stage

As a power source multiple DC-DC converters were used:

- 3 step-up/step-down converters for supplying $12\,\mathrm{V}$ to the XYZRobot servos
- step-down converter for $6\,\mathrm{V}$ to supply wrist servo
- step-down converter for $5\,\mathrm{V}$ to supply micro servos in gripper twist joint and gripper and eyes' LED rings

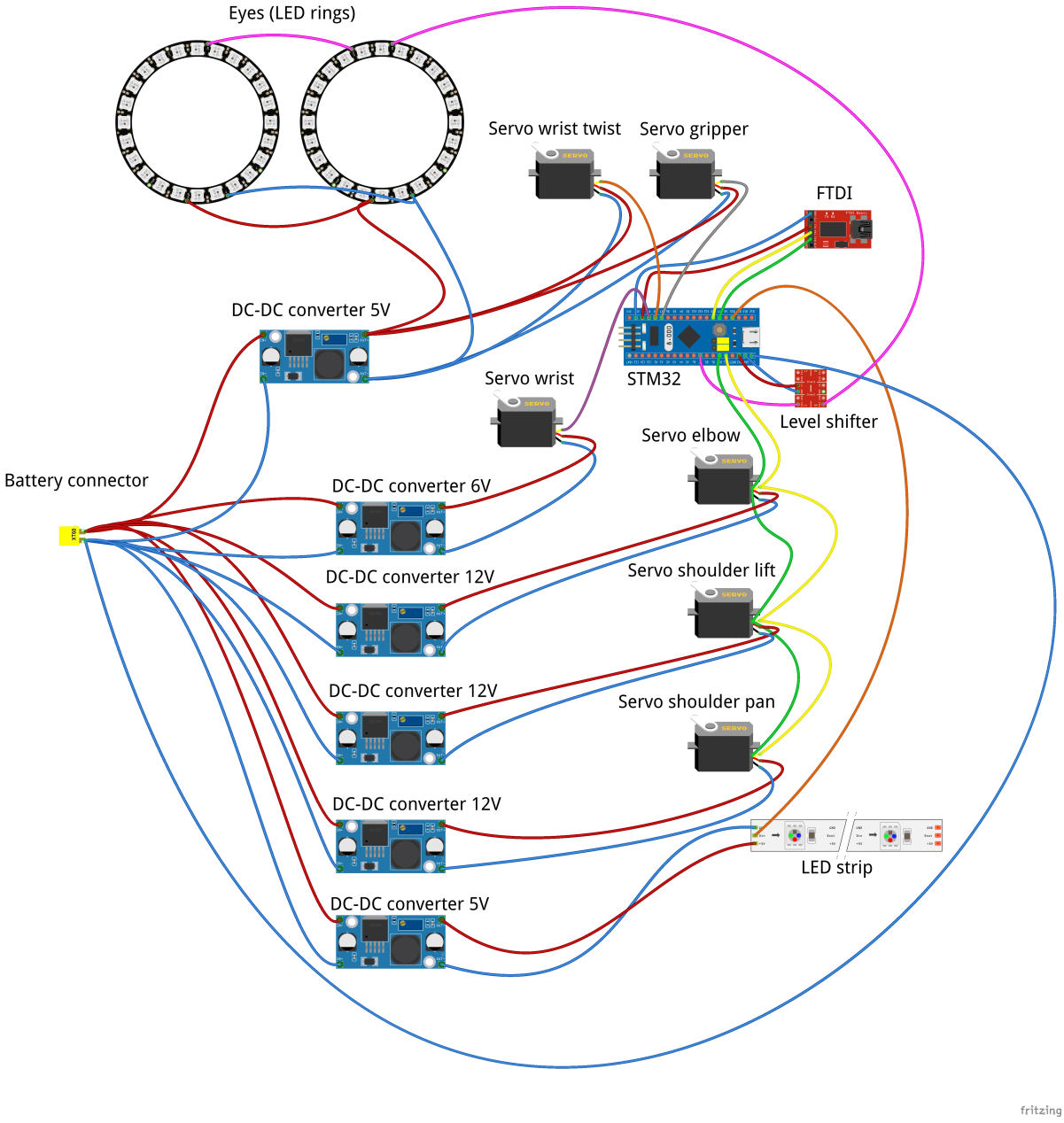

Figure 3.5. Upper electronics schematic

3.4.2. Upper microcontroller

As an upper controller, a second STM32F103C8T6 based development board was used. It has two main tasks: controlling the arm's servos and LEDs mounted in the eyes and body. To program it, *Arduino for STM32* was used, mainly because of already available libraries wrapping communication with XYZServos, analog servos and two LED types that I used, which made development faster. Similar to the base controller, FTDI USB-UART converter was used to provide communication with a computer.

3.4.2.1. Communication with smart servos

Smart servos communicate with a special protocol using an UART interface. In the protocol IDs of the servos are specified, so they can be connected using one interface, which saves the number of cables required. To achieve desired movement two values are sent to servos:

- **position** (value from 0-1024) specifying angle
- **playtime** (value from 0-255) specifies how much movement from current angle to new position should take (this value should be multiplied by 10ms to get actual time)

3.4.2.2. Communication with hobby servos

Controlling hobby servos was done by sending $50\,\mathrm{Hz}$ PWM signal with pulses between $0.5\,\mathrm{ms}$ to $2.5\,\mathrm{ms}$. By default these servos are really simple and will always go with maximum speed (depending on servo and given voltage) to new position, so for better control over arms' trajectory in time it was also necessary to implement an algorithm allowing changing speed of the servo.

I was able to achieve it with the following approach: control loop worked with period of $7\,\mathrm{ms}$ and every iteration it changed the current signal sent to servos by specified step until desired signal was achieved. By calculating various incremental steps different "speeds" could be achieved. To be precise it isn't exactly speed control, only an approximation consisting of changing position with full speed and stopping, which when averaged gives desired movement. To get a better approximation of the movement it is also important to set quite high frequency of this controller.

Another aspect of this controller is that it is only able to lower the speed of the servo, physical limitation of the servo's speed can't be changed, but servos used had maximum speed more than enough for my application.

Commands send to control the servo consisted of the following values:

- **position** (value from 500-2500) specifying angle
- **step** (float value without required range) which is used in control algorithm to change speed of servo movement

3.4.3. LEDs

For improved appearance and to allow better indication of the robots state I added LEDs. Two LED rings in the eyes consist of 48 combined (24 each) WS2812B LEDs and the body strip consists of 150 SK6812RGBW LEDs. To control them Arduino libraries for these LEDs models were used, they used modified SPI communication using only MOSI port to send commands.

4. System Design

This chapter focuses on software configuration of the robot.

4.1. Docker

Docker [9] is a technology that allows OS level virtualization, separating software components into containers. When running on Linux it shares a kernel with the host system, which allows it to be much more efficient than other virtualization techniques such as virtual machines. It is vastly used in other fields, but also starts to become standard in robotics.

Using docker in system design has many advantages:

- **version control** - problems caused by version mismatches of the dependencies are often an issue during software development. Docker allows recreating the whole system on different machines, with the same versions of dependencies.
- **portability** - the main computing device of the robot is a laptop, which isn't really a fixed part of the platform. It should be possible to use different machines and with docker setup step of the new one will be much shorter.
- **scalability** - due to shortened time of new computers configuration, it is also better when robots are manufactured on a larger scale.
- **development** - it also is very beneficial for developers, as it is possible to have multiple software configurations for different projects, minimizing the possibility of collisions. Additionally each configuration will have the same package versions as the target system.

In this project portability aspect is important when considering that an external laptop will be used inside the robot as a computing device. It makes this solution less dependent on a specific unit, as setting up a new machine will be fast.

Portability of the containers also allows to create a simulation that could be easily used by other people to develop software for this project. Also I initially struggled with versions of packages on the simulation being different than on the robot, before I switched to containers. I often used simulation for early development, so it was important to avoid this situation.

Considering these benefits and also ability to work with new technologies I decided to use it in this project.

4.2. ROS

Software components architecture was based on *Robot Operating System* (ROS) [10]. It allows dividing complicated systems into modules by providing a communication layer between them. In the most common cases *nodes* (programs that provide specific functionality) communicate through *topics* by publish-subscribe messaging pattern, which is a many-to-many one-way transport type. For other applications ROS also provides a request-reply model with *services*, in which the caller can request specific procedure or information from the server. Another available type of communication is *action*. Like services, actions are used to request nodes to perform some task, but call to action server is non-blocking. Instead actions provide feedback on current status of execution to track status. This property, with additional possibility to cancel a task, makes them suitable for situations, when execution of a procedure takes more time to complete.

One of the core premises of the ROS was to allow the robotic community to share solutions to common problems, so that it won't be necessary to reimplement them every time. Vast collection of packages and libraries available for ROS enables much faster development of the robots. Also in this work I heavily relied on these off-the-shelf solutions. Integrating them to provide desired application required only tuning parameters and providing some smaller nodes for higher level logic e.g. associating table with position on the map.

4.3. Navigation system structure

Navigation system consists of all elements necessary for the robot to move around the environment to a specific location. It can be represented by a black box model with one input (*move_base action*) to which a specific destination (x, y, θ) is sent. In the following sections I will briefly introduce some of the most important packages used.

4.3.1. Robot Localization

Robot Localization [11] is used to fuse data from multiple sensor sources by providing an implementation of nonlinear state estimators such as EKF and UKF. In this robot it was used to combine data from encoders with IMU to achieve a higher precision odometry position (process of configuration is described in more detail in section 5.1).

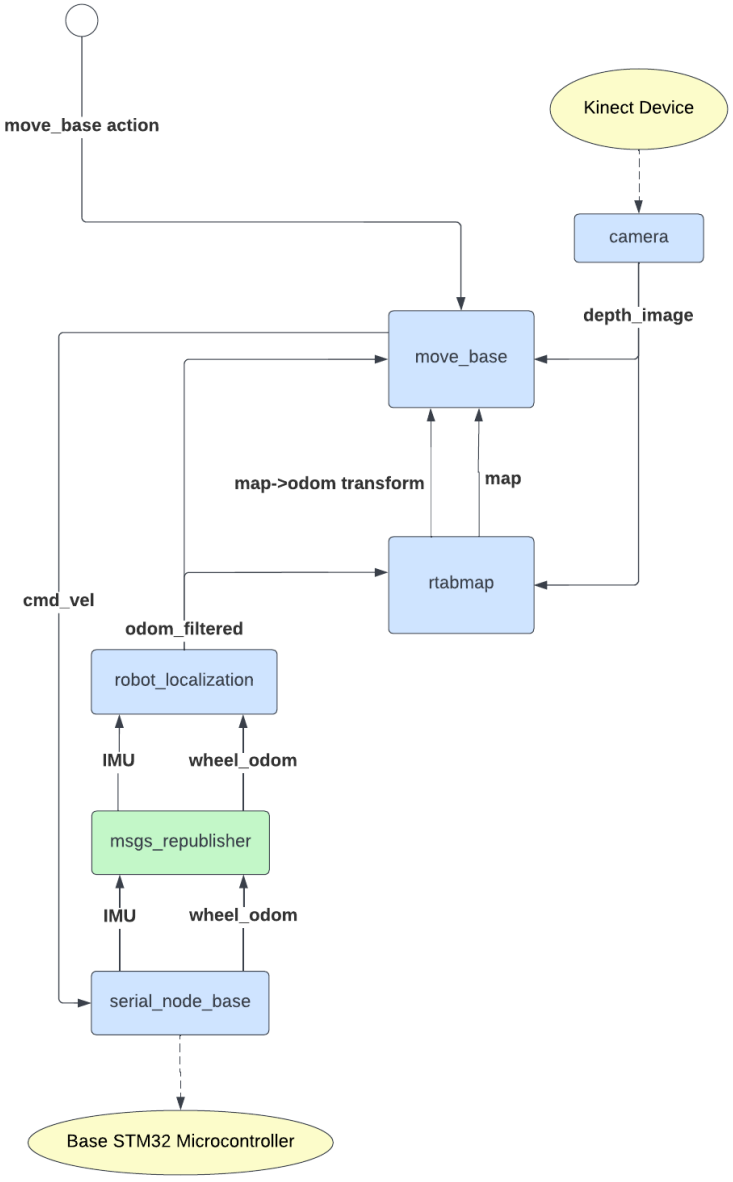

Figure 4.1. Structure of the navigation system, blue color represents ROS packages, green - custom nodes and yellow represents devices

4.3.2. RTABMap

RTABMap (Real-Time Appearance-Based Mapping) [12] package is used to solve a problem known as *Simultaneous Localization and Mapping* (SLAM) - given data from odometry and range sensor (in this case point cloud from Kinect) it creates a map and calculates corrected global position of the robot. Data from odometry is not perfect and it is prone to drift over time. By analyzing data from Kinect, RTABMap is able to detect so-called loop closures (places already visited by the robot) and correct drift from odometry.

RTABMap is also able to work in localization mode - previously created map is loaded and position is corrected by looking for loop closures (map is not modified).

RTABMap uses a visual bag-of-words approach for detecting loop closures, while constructing 3D point cloud map. Map is stored as a graph of positions and corresponding data, which allows reconstructing it while additional constraints (loop closures) are inserted. That way corrections can be applied during optimization.

As a result it provides map data and correction of the odometry error in the form of *map->odom* transform.

4.3.3. Move Base

Move Base [13] implements *navigation stack*, which main function is planning and executing paths necessary to drive around the environment. To achieve it *costmaps* are used, which are grid maps with different weights corresponding to obstacles nearby.

Move Base can be divided into two main logical components, each with corresponding costmap and planner:

- **global part** - global costmap is usually a global map from SLAM along with data from a longer distance sensor. Global planner uses a global costmap to find the shortest path to the goal. This path is more of a general indication of where a robot should go, for example which route to take to get faster to the goal. Robot doesn't necessarily follow it closely - for example if some dynamic obstacle is on this path, it will try to avoid it. Algorithms used for planning global paths are for example A* or Dijkstra.

- **local part** - local costmap consists of a smaller area (e.g. $3\,\mathrm{m}$ square around the robot) with data from range sensors included. It usually should have a high update rate, as it is responsible for reacting to dynamic obstacles. Local planner creates executable trajectory with length of usually about $2\,\mathrm{m}$, checking dynamic collisions and sends velocity commands to the robot. It also takes into account other limits such as maximum acceleration.

All these parts are configurable through using various *plugins*, which are different implementations that provide required functions.

Costmaps can be configured by including different plugins that implement handling data from various sources or different operations on costmaps.

Local planner is also specified by plugin, the most popular ones are:

- **Dynamic Window Approach (DWA)** [14] [15] is based on discretely sampling robot's control space - in case of differential base choosing possible combinations of (v_x, v_θ). Number of samples is controlled by parameters. Then for each chosen sample simulation of the robot's trajectory with given control is executed and evaluated by the cost function. Duration of simulation is also controlled by the parameter. Control pair (v_x, v_θ) with the best score is then used to control the robot. Because of its simple nature - trajectory with constant control (v_x, v_θ) is chosen, it is not able to model more complex behavior. For example backing up is problematic for DWA, because at first it will have a worse score of the cost function. In a longer horizon it can be more optimal, but DWA chooses only locally optimal solution.

- **Timed-Elastic-Band (TEB)** [16] [17] takes the initial path generated by the global planner and modifies it using a timed elastic band approach. Classic elastic band algorithm modifies path given by robots coordinates (x, y, θ), TEB adds also time, which allows it to take into account robot's dynamics. Apart from optimizing duration time also other objectives are optimized. Resulting trajectory from TEB is the fastest one, while elastic band returns the shortest one.

Move Base receives the goal position in map coordinate frame, then using data from sensors, map and robot's global position it controls a robot to set destination.

4.3.4. Messages republisher

It is a simple node written in Python that converts IMU and odometry data from the base STM controller to standard ROS messages used by Robot Localization. This conversion is necessary, because I used a custom, lightweight message definition for sending IMU and odometry data. Standard message types for these readings also include covariance matrices, which contain arrays of float numbers. In result these messages are large and sending them over serial connection was problematic. Final configuration of the Robot Localization actually wasn't fusing data from multiple sensors, rather just combined them. In result covariance matrices were just set to constant values, so sending this data over serial communication wasn't necessary.

4.3.5. Rosserial Python

Rosserial Python [18] package acts as a bridge between the microcontroller and the rest of the system by translating communication into appropriate messages using *rosserial* protocol. It provides two way communication - it receives serialized messages from microcontroller publishing them to appropriate topics and subscribes to required topics, serializes messages and sends them to microcontroller. In a similar manner it also translates service communication.

4.3.6. Freenect Stack

Freenect Stack [19] is a package that wraps *libfreenect* Kinect driver and provides ROS interface for it.

4.4. Pick up system structure

Pick up system implements the whole procedure of picking up an object. It consists of a few steps:

- **object detection** - object's position is calculated
- **robot's position detection** - camera is mounted outside of the robot, so it is necessary to find appropriate transformation and find object's position in relation to the robot
- **calculating trajectory** - object's position is sent to MoveIt, which calculate not only inverse kinematics to find final position of the manipulator, but also collision free trajectory to this point
- **executing trajectory** - whole trajectory is then sent to the controller, which sends commands to the microcontroller.

Similar to the navigation system it can be represented by a black box model with one input (*pick_object action*), which doesn't need any additional information, it just triggers the whole pick up procedure.

Some of the packages (Rosserial Python and Freenect Stack) are the same as in the navigation system, new ones will be briefly introduced in the following sections.

4.4.1. ARTrackAlvar

ARTrackAlvar [20] is a ROS package based on Alvar ARTag tracking library. It allows generating special ARTag images and later track their position using image data with option to also take depth data into account. In this project one ARTag was mounted on the robot's shoulder and ARTrackAlvar was used to find the transformation between the upper camera and robot.

4.4.2. MoveIt

MoveIt [21] implements a manipulation stack, which consists of calculating inverse kinematics to assigned position, planning and executing collision-free motion that gets there. It also allows integrating range sensors to detect obstacles dynamically in a changing environment. Similar to Move Base, by using plugins it is possible to alternate MoveIt behavior by choosing different planners and kinematics solvers. Default ones are best suited for 6 or 7 DoF manipulator configuration, other configurations may require some adjustments.

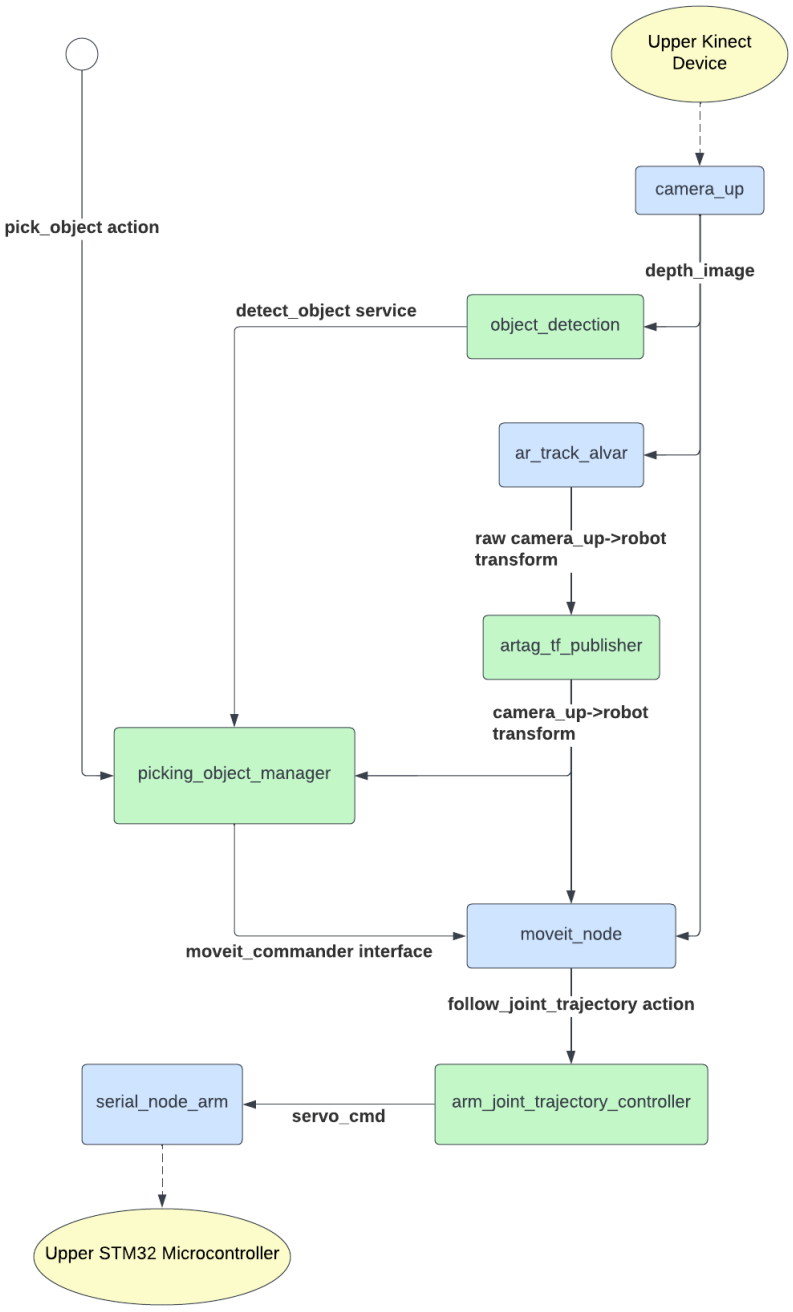

Figure 4.2. Structure of the pick up system, blue color represents ROS packages, green - custom nodes and yellow represents devices

Most popular planners available are:

- **OMPL** (Open Motion Planning Library) [22] - default planner used by MoveIt, it implements randomized motion planners
- **CHOMP** (Covariant Hamiltonian Optimization for Motion Planning) [23] - this motion planning algorithm is based on trajectory optimization, for which it uses gradient approaches

Kinematics solvers:

- **KDLKinematicsPlugin** [24] - default kinematic solver, uses Orocos Kinematics and Dynamics Library. It provides a numerical inverse kinematics solver based on the pseudo-inverse Jacobian method.
- **BioIKKinematicsPlugin** [25] [26] - based on a memetic algorithm that combines gradient optimization with genetic and particle-swarm optimization. If an exact solution is not available, this method will return the best approximate solution.
- **IKFastKinematicsPlugin** [27] - it is possible to generate an analytical inverse kinematics solver for custom configuration using OpenRAVE. With this approach results are really fast, but initial configuration is more difficult than for other solvers.

4.4.3. Custom nodes

Apart from using ROS packages, also some custom nodes were necessary:

- **Pick up object manager** (Python) communicates with MoveIt to achieve desired behavior. It implements a pick up task - first it sends the object's position as a desired pose of the end effector to MoveIt, then closes gripper and sends another request to MoveIt, this time with a point slightly higher to lift up the object. It also sets up the MoveIt scene by adding a table and robot's body (in the URDF model body panels aren't present, so it is necessary to add an artificial box to prevent collisions).
- **ARTag tf publisher** (Python) publishes transform from upper camera to robot. ARTags are detected by ARTrackAlvar with assumption of the 6-DoF of the ARTag. As Kinect is mounted parallel to the ground, it will also be parallel to the ARTag mounted on the robot. Taking this assumption into account, transformation is republished using detected x,y,z position and yaw angle. Roll and pitch angles are constant. This transformation helped quite a lot with reliability, because detected roll and pitch angles had quite large errors.
- **Arm joint trajectory controller** (Python) is a custom implementation of the *follow_joint_trajectory* action required by MoveIt. It receives arm trajectory as a set of joint angles and timestamps at which this point should be reached. Then the controller translates it to servo commands and sends them to the microcontroller.

- **Object detection** (C++) analyzes point cloud data from upper Kinect and returns the bounding box and position of the table and object. Object data is then used for picking up, table data is used to set up the scene in MoveIt for better collision avoidance.

In more detail object detection and arm joint trajectory controller nodes will be described in the chapter 6.

4.5. Communication structure

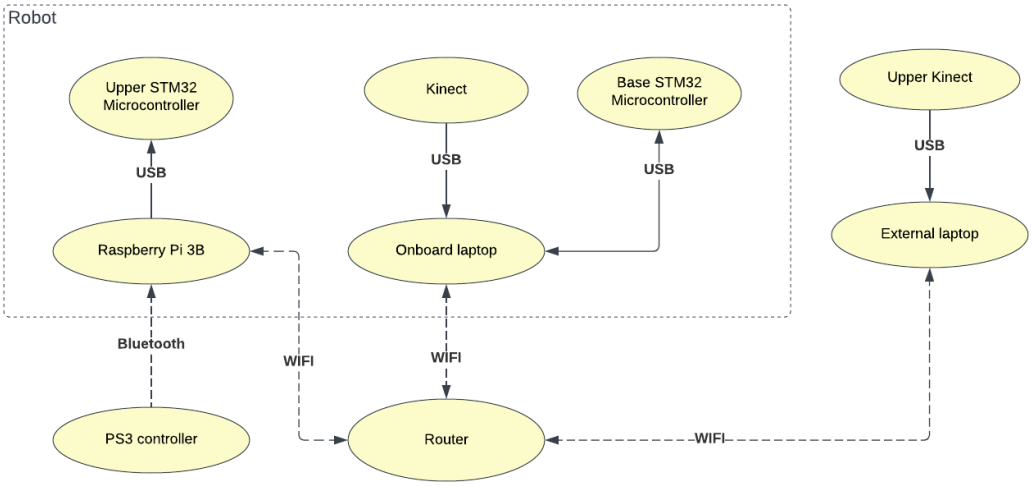

Figure 4.3. Default structure of communication

Implementation of the fetching task required more than one machine to work. Configuration presented on the figure 4.3 is the final one I used, but it is optimized for development and not every element is necessary.

Raspberry Pi was mainly used for convenience, but it is not necessary and all its functions could be moved to the onboard laptop after proper configuration. Another optional equipment is external router - WiFi communication could be also set up by configuring one of the devices' WiFi card into Access Point Mode. PS3 controller connected over bluetooth is used for manual operation of the robot, especially used for mapping. It is a convenient way to do a map, but it is also possible to achieve using other ways, for example controlling using a keyboard.

WiFi communication isn't always reliable, so in this setup most computations are done locally. Messages sent over the network are mostly lightweight and not required to arrive quickly. One of them is the trajectory of the arm sent from the external laptop to Raspberry Pi. As arm controller works locally on Raspberry Pi and message sent contains whole precalculated trajectory delays in communication aren't that crucial. Another information sent from the external laptop to the onboard laptop is the destination

for the navigation system. Also visualization messages are sent to the external laptop, but they are not crucial for working of the system.

During the design phase I also tested other configurations - I tried to use only Raspberry Pi as a computing device, without the onboard laptop, but it didn't work well enough. When trying to run all nodes necessary for autonomous navigation, processing power wasn't sufficient. Another configuration I tried was sending all necessary data to the external computer where computations were run. It also didn't work well enough due to the large amount of data (mainly point clouds and images) that had to be transported over the network. Bandwidth was far too low and the resulting frequency of received data wasn't sufficient.

4.5.1. Time synchronization

When using multiple machines with ROS it is crucial to synchronize their clock times as messages use current system time for a timestamp. To achieve it I used *chrony*, which is an implementation of *Network Time Protocol* (NTP) available on linux. In this application, the robot's onboard laptop was configured to be an NTP server and other devices were connecting to it for time synchronization.

4.6. Simulation

Figure 4.4. Robot picking up bottle in Gazebo simulation

Figure 4.5. Robot fetching bottle in Gazebo simulation

To create a simulation of the robot I used Gazebo [28], which is the default simulation environment for ROS, capable of recreating complex behaviors and environments. By using Fusion2URDF extension it was possible to quickly generate robot model from Fusion 360 design, which also included Gazebo

configuration. Only changes necessary were some minor tweaks in gains of arms' PID controllers and addition of Kinect camera.

Gazebo allows the creation of *worlds*, which are different environment setups. There are many off-the-shelf Gazebo worlds available, so I didn't have to create a new one. I decided to use a *AWS RoboMaker Small House World* [29], which fits well to the robot's targeted use-case. I modified it slightly by adding an upper Kinect, table, obstacle under table and bottle to pick, so that it will match the real setup. In more detail this setup will be described in section 6.1.

Figure 4.6. Simulation environment

As it was quite difficult to create a simulation of the real gripper used in the robot, I instead created a simplified one that consisted of two "fingers" sliding on the perpendicular bar. Its size is larger than the real gripper, because models I used for pick up task were bigger than real ones. To create a simulation closer to reality it would be good to change these dimensions.

Figure 4.7. Grippers in simulated robot, simplified one (on the left) that works in simulation. One on the right is not actuated in simulation, so it isn't possible to pick objects with it.

Simulation wasn't a necessity for this project, but it accelerated software development. First of all it was quite easy to prepare, thanks to tools for generating model and available worlds. With simulation I

could still work without having physical access to the robot and starting simulation was faster than setting up the real robot, which was really convenient especially during early stages of software development. It already allowed me to catch most bugs in the code, leaving only final tests to do on the robot.

Simulation can be also used to prepare integration tests, in this project I created one that consisted of robot driving up to a table, picking up a bottle and delivering it to a designated location. Position of the bottle was read from simulation to later confirm completion of the task. As a further improvement this test can be automatically run every time a change is made to the main code branch, which should allow me to find issues early.

In larger projects simulation could be also useful when multiple people are working on development of the robot and only one physical machine is available.

5. Autonomous Navigation

In this chapter the process of configuring and testing the autonomous navigation system will be described.

5.1. Odometry

Initially I planned to use only encoders as a source of odometry data. In this configuration accuracy highly depends on the traction of the wheels on the floor. After initial tests I found that wheels were slipping quite a lot, which wasn't a surprise as the floor on which I checked it was covered with tiles that don't provide much grip. In result odometry errors were too large for the RTABMap to correct, causing inaccuracies in the created map and robot's position. Quality of navigation was thus too low and I needed to improve odometry precision.

I decided to add an IMU sensor, which can provide orientation data. Before purchasing the sensor I prepared a simple test to confirm if it will help and assess how much accuracy would improve. I mounted my smartphone on the robot and, using the ROS Sensors Driver application, shared data from the IMU to the rest of the ROS system. Because of high latency I had to first record data and synchronize it to check accuracy. In this test orientation from IMU significantly improved odometry quality and RTABMap was able to create an accurate map, so I proceeded with this solution.

The sensor chosen was Bosch BNO055, which provides angular velocity data from gyroscope and fused orientation. To verify improvements and choose the correct fusion configuration I prepared a test: first I marked the starting point of the robot, drove around the house and returned to the same spot. I recorded data from wheel encoders, IMU orientation (fused on board) and angular speed from IMU (raw data from gyroscope) and prepared 4 configurations for Robot Localization package, which used following information:

1. Orientation from IMU and linear velocity from wheel encoders
2. Angular velocity from IMU and linear velocity from wheel encoders
3. Position and orientation from wheel encoders
4. Linear and angular velocity from wheel encoders

Encoders return only position data, to calculate velocities difference quotient was used.

Test	Path information	Configuration	Distance error [m]	Angle error [°]
1	Total distance: 20 m Total angle: 1710°	1	0.14	-4.4
		2	0.67	11.0
		3	2.13	39.0
		4	2.21	39.6
2	Total distance: 24 m Total angle: 2566°	1	0.44	-8.5
		2	1.04	18.8
		3	2.31	35.4
		4	2.48	38.6
3	Total distance: 25 m Total angle: 2488°	1	0.43	1.4
		2	0.89	13.8
		3	1.42	51.3
		4	1.82	65.8
4	Total distance: 26 m Total angle: 2429°	1	0.05	-1.9
		2	1.57	23.1
		3	3.72	60.7
		4	3.72	62.3

Table 5.1. Odometry accuracy test results

Returning the robot to the same point was based on aligning it with the pattern on the floor. It is an approximate method and controlling the robot very precisely is also difficult, so the results aren't perfect. Based on how the tests looked like I estimated that distance uncertainty should be within 2 cm for distance and about 2° for angle. Certainly differences between configurations were much larger than possible inaccuracies in tests, so tests were valid for choosing the right configuration.

All tests were done on pre recorded data, realtime results might differ slightly, but differences in accuracy are large and will still hold.

I didn't have ground truth data, so paths' distance and total angle change is based on the most accurate configuration - the one with orientation from IMU (1). This data isn't perfect, but gives an idea of how tested paths looked like.

I did 4 similar tests and in all of them the best results were achieved for fusing linear speed from wheels and orientation from IMU, so it was used in the final configuration.

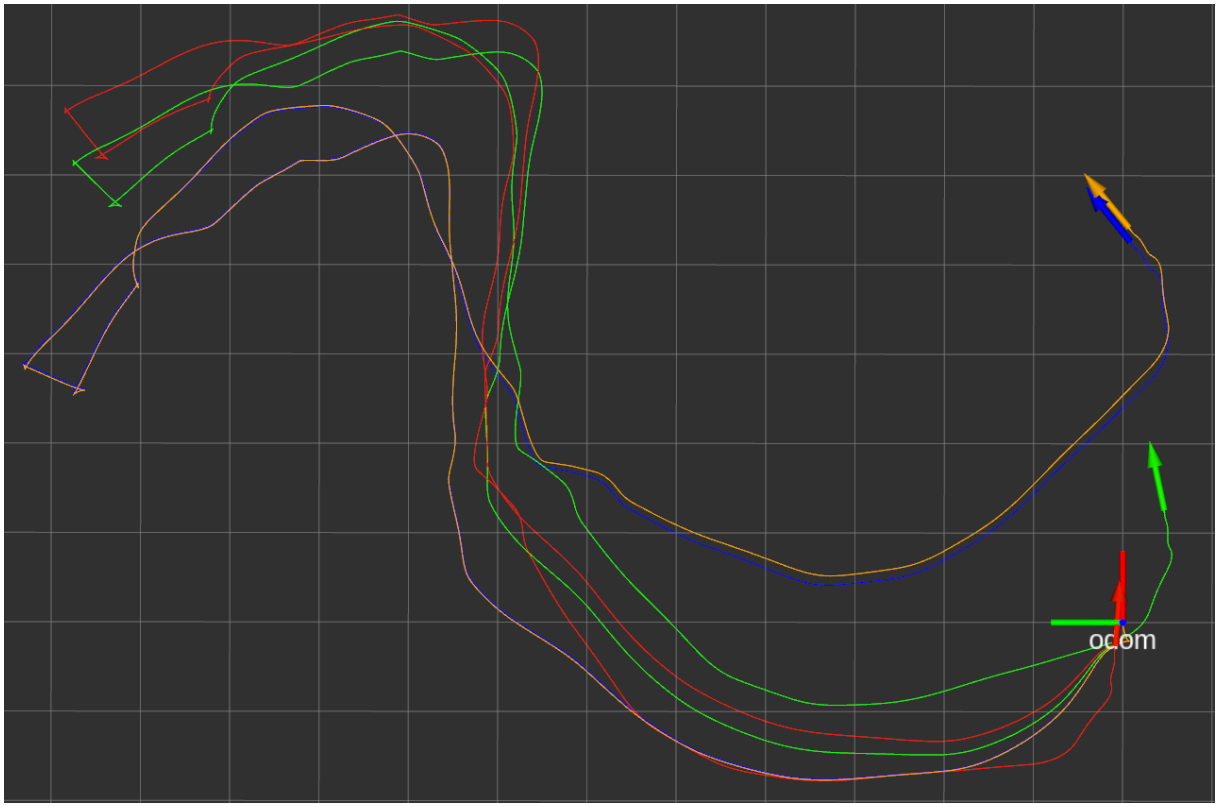

Figure 5.1. Results from odometry accuracy test, plotted lines are trajectories traveled according to each configuration, arrows represent final position and orientation calculated using configurations: red (1), green (2), blue (3), orange (4). The *odom* coordinate frame represents the starting point, in reality the robot returned to it, so ideally arrows should be in this position. Grid's scale is $0.5\,\mathrm{m}$. As the configuration with IMU orientation was quite accurate, the red trajectory could be used as a reference.

5.2. Kinect

Apart from encoders and IMU, one more sensor is used for navigation - Kinect camera. Its intended use case wasn't robotics, but due to its availability, low cost and good quality of data, it became quite popular to also use in robotics. It has some drawbacks, one of which is problems when working in sunlight, which disrupts IR projection of Kinect. This makes Kinect not really usable outdoors, but I also had problems with it during sunny days near windows - some areas were not detected then.

Also because I used only one sensor robot was limited by its field of view. Because it was mounted in the lower part of the robot, tilted towards the ground, the robot was not able to detect higher obstacles. This was a problem when driving close up to the table - the robot was only able to detect legs, not tabletop, and it sometimes planned trajectories colliding with tabletop. To solve this problem I added an obstacle under the table that prevented the robot from going too far and colliding. This solution is only a workaround, proper one will require adding an upper sensor, which I plan to do as an improvement to this project.

Also Kinect is facing forward, so when the robot is backing up it is doing it blindly, without current information about collisions.

5.3. SLAM

Mapping quality improved a lot after getting a more accurate position from odometry, but the default configuration of the RTABMap still wasn't performing well enough. Detecting loop closure wasn't always working correctly and many times false detections happened, which often completely broke map and robot's position. With help of [30] I was able to improve configuration.

As Kinect in my robot is tilted towards the ground, much space on the image is taken by it. Using ground images for detecting loop closures isn't a good idea because it usually has repetitive patterns or not many distinct features, which confuses visual matching. Changing the *Region of Interest* to cut out the bottom part of the image helped with this problem. Second improvement was decreasing maximum distance of the sensor data - some data from Kinect that were further away wasn't that accurate.

```
Kp/RoiRatios: "0 0 0 0.3"
Grid/RangeMax: 1.5 # [m]
```

Listing 5.1. RTABMap configuration

Figure 5.2. Cloud map of my house created by RTABMap

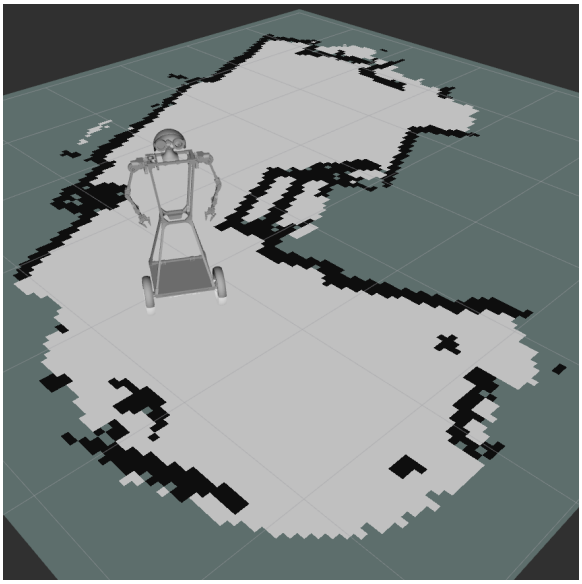

Figure 5.3. Grid map, which is a cloud map from figure 5.2 converted to 2D

After the initial map is done I save it and switch RTABMap to localization mode in which it only provides the current position of the robot on the map. If RTABMap remained in mapping mode there could be problems with memory and computational load, as more data will be included and old data is not removed. This could become a problem quite fast, as the visual based approach with 3D point clouds

generates a lot of data, for example the map of part of my house generated in final application tests in chapter 7 had around 150 MB.

```
Mem/IncrementalMemory: false
```

Listing 5.2. RTABMap localization configuration

As an alternative to RTABMap I also considered using a 2D based mapping method. It required to simulate data generated by planar LiDAR by sampling data from a point cloud with a given height and projecting it into a 2D plane. I was able to set it up, but it didn't perform well enough - simulated LiDAR data from Kinect wasn't good enough to perform accurate scan matching, additionally Kinect's horizontal FoV is $58.5°$, which is low, when compared to 2D LiDARs, which usually rotate and provide full $360°$ FoV.

5.4. Move Base

Once mapping and localization were working well enough I was able to configure autonomous navigation. I will describe the most important parameters' settings.

5.4.1. Move Base parameters

Footprint is a polygon that represents a robot during path planning, when checking for collisions. With arms straight, pointing to the ground, the robot's base is the widest and longest part, so its dimensions were chosen for this parameter. Footprint chosen is a rectangle, specified by four corners, measured in *base_link* coordinate frame, which is a special location that represents the robot's origin. In my case I set it to be on the front wheels' rotation axis, exactly in the middle. Also I specified padding, which increases footprint size by specified value, to add a safety margin. These parameters are common for global path planning and local path planning.

```
footprint: [[0.15, 0.2875], [-0.355, 0.2875],
            [-0.355, -0.2875], [0.15, -0.2875]] # [m]
footprint_padding: 0.02 # [m]
```

Listing 5.3. Footprint configuration

Another important parameter is the controller's frequency. Controller sends velocity commands to be executed by the robot. I set it to $10\,\mathrm{Hz}$, as the compromise between time of response (robot should be able to stop immediately if an obstacle is detected) and amount of data sent (with too high frequency microcontroller could not be able to process data).

```
controller_frequency: 10.0 # [Hz]
```

Listing 5.4. Controler frequency configuration

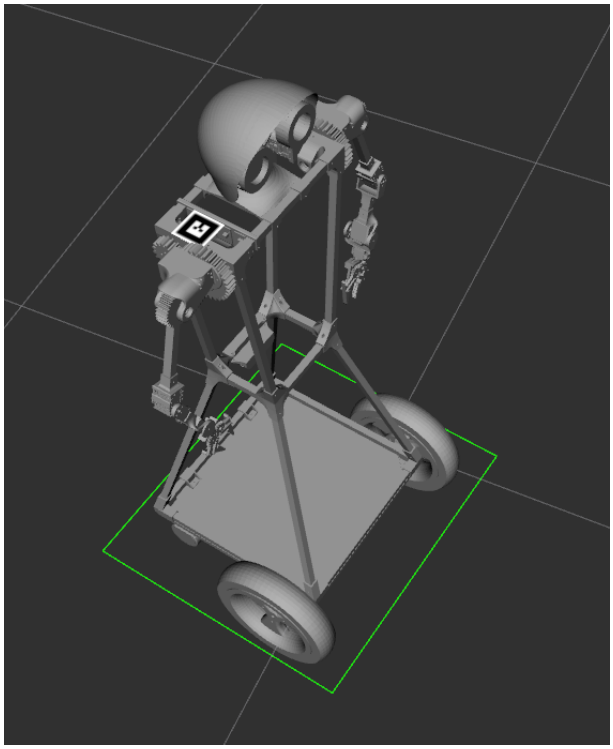

Figure 5.4. Footprint (green rectangle)

5.4.2. Global costmap

plugins parameter sets what information should be taken into account in the global costmap. I used two plugins - *StaticLayer*, which takes a map of the environment created by the SLAM algorithm. Second one is *InflationLayer*, which modifies the costmap from StaticLayer by adding increased costs around obstacles.

```
plugins:
  - {name: static_map_layer, type: "costmap_2d::StaticLayer"}
  - {name: inflation_layer, type: "costmap_2d::InflationLayer"}
```

Listing 5.5. Plugins used in global costmap

Typically StaticLayer will have only values for free space, obstacle and unknown. I set *inflation_radius* (what distance should costs be increased in [m]) parameter to quite high value so that the global planner will prefer planning paths further from obstacles. Global planner tries to find the shortest path, so it can prefer ones that can get close to obstacles, by modifying costs it will change behavior. *cost_scaling_factor* modifies how costs will change within inflation radius.

```
inflation_layer:
  inflation_radius: 0.75 # [m]
  cost_scaling_factor: 2.0
```

Listing 5.6. Inflation layer configuration

As the global costmap doesn't include sensors that can detect dynamic obstacles, it is not crucial for it to work with high frequency. Lowering frequency decreases computations necessary.

```
update_frequency: 2.5 # [Hz]
```

Listing 5.7. Update frequency of global costmap

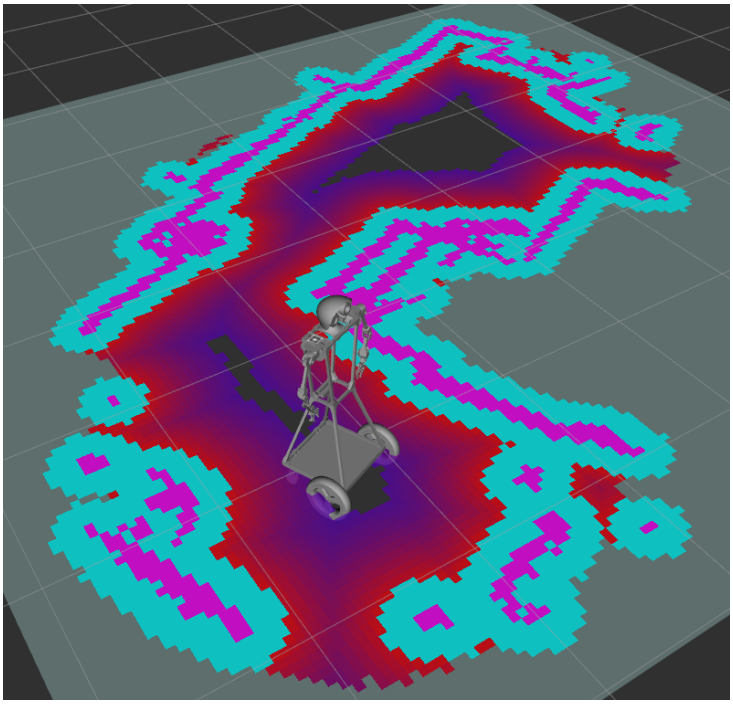

Figure 5.5. Global costmap

5.4.3. Global planner

neutral_cost and *cost_factor* parameters specify the cost function used when searching for the path. They were set, so that plan will be farther from obstacles, values were based on [31]. It is also important to make the global plan quite smooth, as it is taken into account during local planning. Not smooth global plan have a negative impact on the performance of TEB local planner.

```
neutral_cost: 66
cost_factor: 0.55
```

Listing 5.8. Global planner configuration

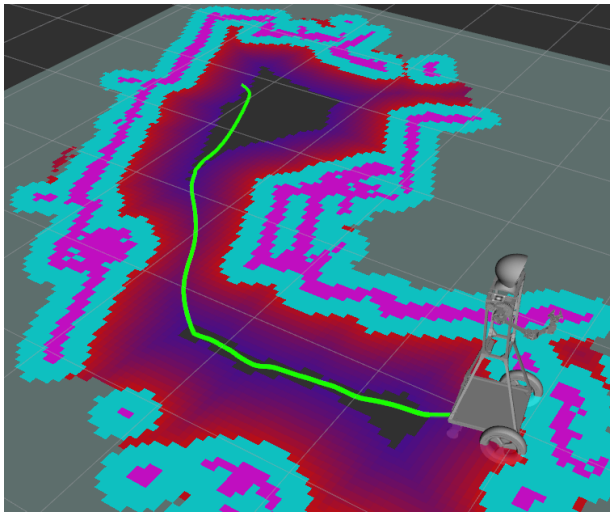

Figure 5.6. Global plan

5.4.4. Local costmap

In local costmap two sources of obstacles were used: *static_map_layer*, which includes static obstacles present during mapping and *rgbd_obstacle_layer*, which includes current detections from Kinect sensor. The latter one leverages *SpatioTemporalVoxelLayer* [32], which does some preprocessing prior to marking them on the map. First voxel filter is used to reduce resolution and size of the received point cloud. Detections below minimum height are removed, as they contain ground points. After detected points leave the field of view of the sensor, they are preserved for some time, after which they are decaying. Main goal of the rgbd_obstacle_layer is to detect dynamic changes of the robots environment, so that it can react to it.

I decided not to use the inflation layer in the local costmap. It wasn't necessary, as TEB local planner uses its own inflation parameter and doesn't take it into account inflated costs on costmap. It was an optimization done after deciding to use TEB local planner, when using DWA local planner the inflation layer was needed.

```
plugins:
  - {name: rgbd_obstacle_layer,
     type: "spatio_temporal_voxel_layer/SpatioTemporalVoxelLayer"}
  - {name: static_map_layer, type: "costmap_2d::StaticLayer"}
```

Listing 5.9. Plugins used in local costmap

Local costmap is set to be square with a robot in the middle. Length of the side was set to $6\,\mathrm{m}$, as maximum range used from Kinect was set to $3\,\mathrm{m}$. Another important parameter is *update_frequency*, which should be high enough to enable the robot to react to dynamic changes of the environment, but not too high, so that computational load won't be too large. Similar to the static map, resolution of the local costmap was set to $0.05\,\mathrm{m}$.

```
update_frequency: 5.0 # [Hz]
width: 6.0 # [m]
height: 6.0 # [m]
resolution: 0.05 # [m]
```

Listing 5.10. Local costmap settings

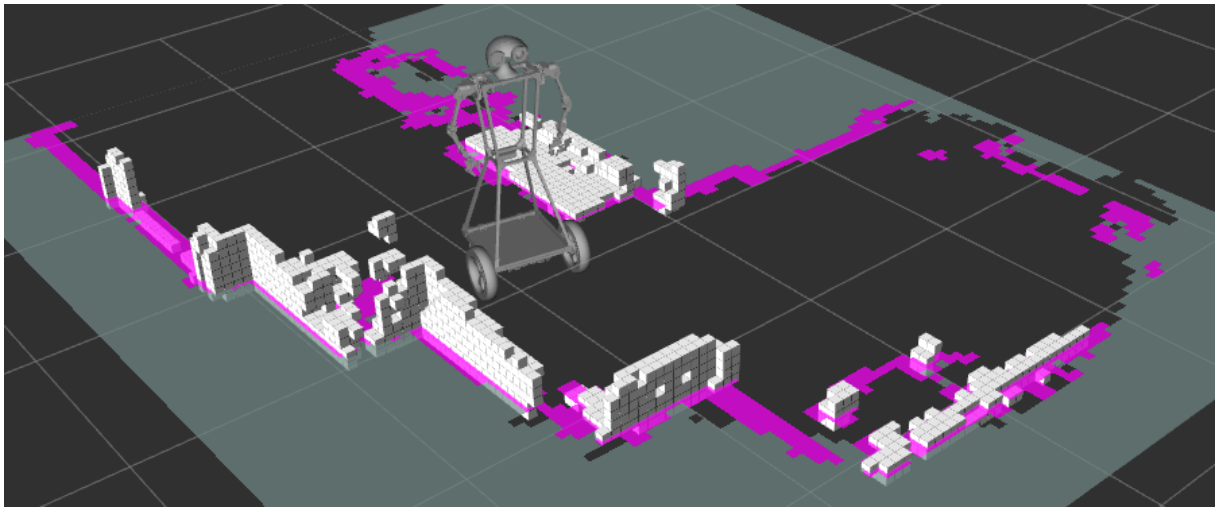

Figure 5.7. STVL layer (white cubes) and final local costmap

Decay model for the STVL was set to be linear and decay time, which specifies how long obstacles will persist after they were observed, was set to $15\,\mathrm{s}$. The decay time is modified by the decay acceleration parameter when the observed point is within FoV of the sensor, but isn't detected.

Size of the voxel was configured to be the same value as the resolution of the costmap, $5\,\mathrm{cm}$ is a good compromise between computation time and accuracy of the costmap.

I set the minimum obstacle height to $-0.1\,\mathrm{m}$. Zero point is located at the base_link, which is defined to be on the axis of rotation of the wheels, directly between them. This means that zero point is located $0.145\,\mathrm{m}$ (radius of the wheel) above the ground, that's why the negative value of this parameter. When transformed to the ground coordinate frame, this setting equals to $4.5\,\mathrm{cm}$ minimum height of detected obstacles. Setting this value higher could result in some of the floor points being detected, as the point cloud registered by Kinect has some noise, so $4.5\,\mathrm{cm}$ is the lowest value that also guarantees no false detections.

RGBD sources were set so that Kinect is able both to mark detected points and clear them, when they weren't detected. Vertical and horizontal FoV angles were specified based on a datasheet for Microsoft Kinect sensor.

```
rgbd_obstacle_layer:
  voxel_decay: 15 # [s]
  decay_model: 0 # linear
  voxel_size: 0.05 # [m]
  obstacle_range: 3.0 # [m]
  observation_sources: rgbd1_mark rgbd1_clear
  rgbd1_mark:
    marking: true
    clearing: false
    min_obstacle_height: -0.1 # [m]
  rgbd1_clear:
    marking: false
    clearing: true
    vertical_fov_angle: 0.7958701 # [rad]
    horizontal_fov_angle: 1.021018 # [rad]
    decay_acceleration: 5.0 # [1/s^2]
```

Listing 5.11. rgbd_obstacle_layer configuration

5.4.5. Local planner

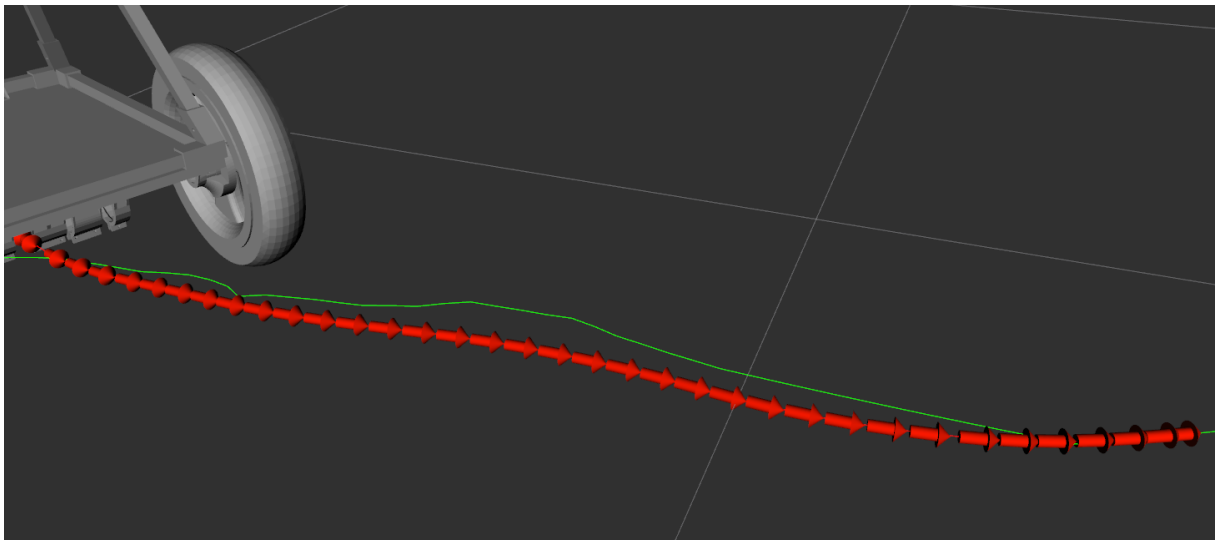

Figure 5.8. Local plan (red arrows)

Choosing and configuring a local planner was the biggest challenge in setting up the Move Base. I checked two available planners: DWA and TEB, starting with DWA, as it is a simpler one. Quite quickly I was able to achieve a working configuration that executed simple trajectories. There was a problem though, it wasn't able to plan more complex paths, for example ones that consisted of backing up, which are crucial for application of driving close to the table. Due to the simple nature of the algorithm it isn't really able to consider what will be more beneficial in the future and it was a limitation that was not possible to fix with configuration, so I decided to switch to TEB. It is more complex and also tougher in

configuration, so it took me some time until I was able to achieve results comparable to DWA, but after some further configurations results were much better.

I set acceleration and velocity limits to quite low values to ensure smooth movement. The backward limit is set to be almost the lowest achievable speed. As the robot doesn't have any backward facing sensors it isn't safe to move backwards, so it should be used only when there is no other solution to path planning.

```
acc_lim_theta: 1.5 # [rad/s^2]
acc_lim_x: 0.2 # [m/s^2]
max_vel_x: 0.2 # [m/s]
max_vel_theta: 0.5 # [rad/s]
max_vel_x_backwards: 0.08 # [m/s]
```

Listing 5.12. Robot configuration in TEB

Goal tolerances were set such that the robot will navigate quite close to the goal position, which is important when considering the ability to pick up objects from the table, as arm reach is limited and the robot has to drive precisely to table's position. On the other hand they can't be set too low - it has to be taken into account that localization isn't perfect, position can oscillate. Also velocity commands sent to base aren't executed perfectly, so too low tolerance will result in the robot oscillating around goal position. With chosen values the robot was able to achieve destination smoothly, usually in the first attempt, rarely needing to back up and correct.

```
xy_goal_tolerance: 0.1 # [m]
yaw_goal_tolerance: 0.15 # [rad]
```

Listing 5.13. Goal tolerance configuration

Length of the local plan is specified by *max_global_plan_lookahead_dist* parameter. Larger values increase computational load, but too small values will make it difficult to make an optimal plan avoiding new obstacles.

dt_ref parameter sets desired temporal resolution of the trajectory. This value was set to quite low value (by default it is set to $0.3\,\mathrm{s}$), which allowed better navigation in narrow spaces that was needed in my application. If this value was set too low, the computational load could be too large and the controller won't be able to work with desired frequency, so I tried to find an optimal setting.

Temporal resolution of the trajectory is also part of the optimization, it isn't set to be constant dt_ref value, but change in dt_ref $\pm$ *dt_hysteresis* range. I started with dt_hysteresis equal to 10% of dt_ref, but found that performance was better with higher value. dt_ref and dt_hysteresis parameters are correlated and were tuned together.

global_plan_viapoint_sep parameter specifies how points from the global plan will be sampled. Higher distance between these points will allow local planner more freedom in generating trajectories, when distance is lower TEB will follow the global plan quite closely. I set it to quite low value, because the global plan was tuned to be possibly quite far from obstacles, which was desired behavior. Otherwise, with higher separation distance, TEB sometimes planned trajectories closer to obstacles.

Parameter that greatly improves smoothness of path execution is *control_look_ahead_poses*. Instead of sending current velocity control commands, it specifies to take future commands. In my case it helps, most likely because of the long response time of the PID control. By sending commands earlier the robot is able to then respond in the correct time.

```
dt_ref: 0.16 # [s]
dt_hysteresis: 0.032 # [s]
global_plan_viapoint_sep: 0.5 # [m]
max_global_plan_lookahead_dist: 1.2 # [m]
control_look_ahead_poses: 6
```

Listing 5.14. Trajectory configuration

Minimum distance to obstacles that robot should keep when executing trajectory was set to be $10\,\mathrm{cm}$. Keeping this distance is one of the criteria in optimization function.

```
min_obstacle_dist: 0.1 # [m]
```

Listing 5.15. Obstacle configuration

no_inner_iterations and *no_outer_iterations* control the behavior of the optimizer. Total number of iterations is a product of these two numbers, during outer loop trajectory is additionally resized taking into account the *dt_ref* parameter. Number of optimizer iterations changes required computation time and they were set to the highest values that allowed the controller to work with $10\,\mathrm{Hz}$.

```
no_inner_iterations: 4
no_outer_iterations: 2
```

Listing 5.16. Optimization configuration

Epsilon parameter is used to add safety margin for hard constraint - penalty starts to be non-zero slightly before actual constraint is broken. This value was chosen based on the smallest constraint value (*max_vel_x_backwards*), so that the robot is able to have some backward velocity without penalty.

```
penalty_epsilon: 0.04
```

Listing 5.17. Optimization configuration

As TEB is based on the optimization of the cost function, also correct weights had to be chosen. I will describe the most important changes to the default values. Very large value had to be set to *weight_kinematics_nh*, because the robot is nonholonomic and can't move in the y direction. By setting quite large values to the *weight_max_vel_x* and *weight_max_vel_theta* I made sure that planned commands will be within set limits. Another large value was set to *weight_obstacle* to keep minimum distance from obstacles. Quite large value was set to *weight_kinematics_forward_drive* - robot has only front facing sensor, so it should move forward most of the times, only backing up when there is no other possibility. *weight_viapoint* was slightly increased, so that the robot will more closely follow the global path, which was set to be possibly far away from obstacles. By being closer to the global path local planner should also keep greater distance from obstacles.

```
weight_kinematics_nh: 1000.0
weight_max_vel_x: 100.0
weight_max_vel_theta: 100.0
weight_obstacle: 100.0
weight_kinematics_forward_drive: 10.0
weight_viapoint: 2.0
```

Listing 5.18. Optimization weights

Whole tunning was done based on real robot tests, which made them quite time consuming. It was necessary to repeatedly drive the robot to destinations and check performance. Simulation isn't ideally modeling the robot's performance, especially flaws of the drill motors, which makes it impossible to tune parameters for the real robot on simulation. Also it has to be taken into account if there will be enough computation power to keep the controller working with desired frequency, which is hard to determine when working on simulation.

5.5. Autonomous navigation tests

To measure the accuracy of the autonomous navigation I marked the position of the map origin, (x: $0\,\mathrm{mm}$, y: $0\,\mathrm{mm}$, alfa: $0°$) point, in the real world and repeatedly commanded the robot to navigate to it. Each time the path length to map origin was around $5\,\mathrm{m}$.

Points were measured using two strings that were pushed under wheels. With this method it was quite easy to precisely measure orientation angle and position. I assumed that the point in the middle of the rectangle created by strings and wheels will be the *base_link* point projected on the ground.

Figure 5.9. Measuring method of robot's position during navigation precision tests

Test	Measured position			Error	
	x [mm]	y [mm]	alfa [°]	Distance [mm]	Angle [°]
1	24	25	-5	35	5
2	-22	-19	-2	29	2
3	24	6	2	25	2
4	27	0	3	27	3

Table 5.2. Results of going to setpoint (x: 0 mm, y: 0 mm, alfa: 0°)

In one test robot had to slightly back up, because it overshot the destination, otherwise in all tests the goal was reached directly. Achieved precision is quite good, every time the robot was able to reach it within 3.5 cm with maximum rotation of 5°.

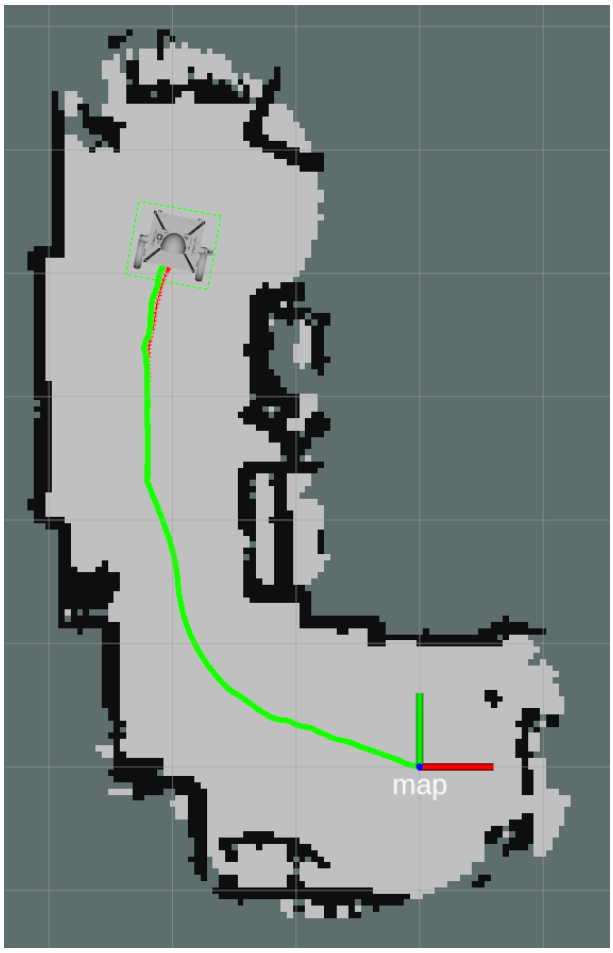

Figure 5.10. Path to point in autonomous navigation tests, grid scale is 1 m

Using data from this test it is also possible to check accuracy of the robot's position on the map - I measured robot's position in real world in reference to map origin and I checked the *map->base_link* transform to have the position where the robot thinks it is.

Test	Measured position			Position on map			Error	
	x [mm]	y [mm]	alfa [°]	x [mm]	y [mm]	alfa [°]	Distance [mm]	Angle [°]
1	24	25	-5	25	25	-6	1	1
2	-22	-19	-2	-24	-15	-1	4	1
3	24	6	2	22	7	2	2	0
4	27	0	3	31	3	4	5	1

Table 5.3. Results of robot's position accuracy

These results are also really precise, the robot is able to localize itself with errors lower than $0.5\,\mathrm{cm}$ and $1°$.

5.6. Dynamic obstacles tests

To check how the robot responds to dynamic obstacles I waited outside the field of view of the Kinect sensor and jumped in front of the robot. I used a pattern on the floor to approximate distance when appearing in front of the robot. In all tests I jumped in front of the robot when it was going in a straight path at around maximum autonomous velocity ($0.2\,\mathrm{m/s}$).

Test	Approximate distance of obstacle to robot	Did robot avoid collision with obstacle
1	$80\,\mathrm{cm}$	yes
2		yes
3		yes
4		yes
5	$40\,\mathrm{cm}$	yes
6		yes
7		yes
8		yes

Table 5.4. Results of dynamic obstacle tests

In all test cases the robot stopped before collision, tests were done in quite narrow place, so it wasn't possible for the robot to go around me, but in more open space it should be also able to move around an obstacle that appeared. In tests 5-8 robot stopped around $10\,\mathrm{cm}$-$15\,\mathrm{cm}$ in front of my legs, so $40\,\mathrm{cm}$ is around the limit where the robot is safely able to avoid collision.

6. Picking Up Objects

In this chapter the process of detecting an object and picking it up will be described.

6.1. Scene configuration

The Kinect sensor already mounted in the robot is used for navigation and it is positioned in the lower part of the robot. With this placement it is impossible to detect objects for manipulation, as they are outside of the field of view of the sensor. Because of that I decided to use a second Kinect for object detection.

It wasn't possible to place it on the robot, as it was too big to mount in the head or upper side of the robot - additionally to dimensions it also requires a minimum distance of about $0.5\,\mathrm{m}$ to detect objects. Instead I decided to mount it externally, above the table. It introduces limitations, as objects can be picked only from the location with sensor mounted, but it also was an interesting problem to solve. In this configuration not only the object's position has to be detected, but also the robot's position in the camera frame.

6.2. ARTag detection

To detect the robot's position I decided to use ARTags, as there are solutions available to detect markers such as the ARTrackAlvar ROS package that I used.

Final configuration that allowed me to achieve quite precise detection was a bundle that consisted of 2 ARTag markers. It used a special bundle mode available in ARTrackAlvar, for which it was necessary to define a configuration file with position of the one marker with respect to the other. Apart from it I manually set detected pitch and roll angles to $0°$, as Kinect was mounted parallel to the robot. This assumption was left from previous experiments with detection, in bundle mode detections were more precise, so that assumption may no longer be necessary, but I haven't verified it.

It took me some time to achieve a final configuration that was satisfactory, it is also worth mentioning other approaches. Initially I used configuration with only one ARTag marker mounted on the robot. In this case detections were very noisy, so I added EKF through the Robot Localization package as an estimator. By modifying variances I changed the behavior of the filter, so that high frequency noises were mostly filtered, but it still wasn't enough. What I found out was that roll and pitch angle estimations

Figure 6.1. Scene configuration, Kinect mounted above the table is marked with ellipse 1 and ARTags mounted on robot are marked with ellipse 2

were not really accurate. As Kinect was mounted parallel to the table, I added this assumption and it improved detection accuracy. Still it wasn't good enough, especially yaw angle detection was sometimes incorrect and noisy, which was finally improved by using bundle mode.

6.3. MoveIt configuration

Initial configuration of the MoveIt using setup wizard went smoothly, but then I discovered that the default kinematics solver wasn't designed to work with a 5-DoF arm. Robot's arm configuration doesn't allow finding a precise solution for most cases when specifying the full position of the end effector $(x, y, z, roll, pitch, yaw)$. Setting only position option helped, but it wasn't the best solution, as orientation also needs to be defined to pick up an object. I was able to achieve better results by changing the kinematics solver to *BioIKKinematicsPlugin*. As it calculates approximate solution it is

able to find the closest achievable position, not necessarily the exact one. I also confirmed with tests that these approximated positions were accurate enough for picking up a bottle.

Another parameter that I had to determine was the search resolution of the inverse kinematics solver. Good results were achieved with $1\,\mathrm{cm}$, I also tested lower values, but for the task of picking up a bottle this value looked sufficient. It could be too low when the object's width is closer to the max span of the gripper, then more precision will be necessary. Additionally lower resolution speeds up computations.

Solver's timeout parameter was set to quite high value, as I prioritized finding a solution, not necessarily doing it quickly.

```
kinematics_solver: bio_ik/BioIKKinematicsPlugin
kinematics_solver_search_resolution: 0.01 # [m]
kinematics_solver_timeout: 0.2 # [s]
```

Listing 6.1. Kinematics solver parameters

Next issue with initially generated configuration was planning taking too long time. By changing the sensor plugin to use depth image instead of point cloud I was able to achieve large improvements and reduce this time significantly.

I also set the range of the data used to be approximately below the arm's lowest point - these points were mainly ground and other objects below the reach of the arm, so it wasn't possible to collide with them.

Padding parameters are used to exclude points that belong to objects already present in the scene - for example so that the robot's arm detected from Kinect won't be treated as an obstacle (otherwise the robot will be constantly in collision state). I set the scale to low value, below recommended 1.0, because the robot has to be quite close to the table to pick up an object and with larger scale values some of the table points could be removed. Instead I set offset to a higher value, which worked best.

Shadow points are the ones that appear below the object in the scene - for example point below robot arm enlarged with padding. By default all the points below the arm will be removed apart from ones with distance smaller than shadow threshold. To find this detailed description I had to refer to comments in the MoveIt code, but still I found this parameter setting quite tricky, as its behavior didn't quite match expectations and sometimes points on the table below arm were still removed, but I got best results for setting it to $0.1\,\mathrm{m}$.

```
sensors:
  - sensor_plugin: occupancy_map_monitor/DepthImageOctomapUpdater
    far_clipping_plane_distance: 1.8 # [m]
    shadow_threshold: 0.1 # [m]
    padding_scale: 0.4
    padding_offset: 0.2 # [m]
```

Listing 6.2. Configuration of Kinect sensor in MoveIt

Another problem was trajectories being planned too close to obstacles, which wasn't safe. I haven't found any possible parameters to set in the motion planner, as a workaround I tried increasing *octomap_resolution*. Octomap is a 3D occupancy grid map that is used to represent obstacles detected by depth sensor. Resolution increased voxel size, but it didn't work properly, as this method doesn't guarantee that the voxel will be inflated around points - points could as well be near the edge of the voxel. I was able to fix this problem by externally detecting the table and inserting it into the planning scene with a bigger bounding box. Finally I set the resolution of the octomap to $5\,\mathrm{cm}$.

```
octomap_resolution: 0.05 # [m]
```

Listing 6.3. Octomap configuration

For every joint I set velocity and acceleration limit. The values were chosen based on tests on the real arm, I observed the smoothness of the arm's movement and decided which configuration looked best.

```
max_velocity: 2.0 # [rad/s]
max_acceleration: 3.0 # [rad/s^2]
```

Listing 6.4. velocity and acceleration configuration

As the motion planner I tried OMPL, CHOMP and a combination of the OMPL and CHOMP. In combination OMPL provided an initial plan, which was then optimized by CHOMP. Through testing different planners I hoped to solve the problem of arm moving too close to obstacles described earlier. I didn't notice much difference between these configurations though, so I decided to use the default OMPL planner.

```
pipeline: ompl
```

Listing 6.5. Motion planner setting

Arm controller manager was set to default MoveItSimpleControllerManager. It requires a controller supporting FollowJointTrajectory action, which I implemented (described in 6.4).

```
moveit_controller_manager: moveit_simple_controller_manager/MoveItSimpleControllerManager
```

Listing 6.6. Controller manager

Controller had to be defined in the controller_list, with all the joints controlled by it.

```
controller_list:
- name: arm_position_controller
  type: FollowJointTrajectory
  joints:
    - right_shoulder_pitch
    - right_shoulder_roll
    - right_elbow
    - right_wrist
    - right_gripper_twist
```

Listing 6.7. Controller settings

With my setup of multiple devices connected over WiFi network it was also important to increase timeout of the controller connection, as startup in this configuration can take more time and it will fail if MoveIt can't connect to the arm controller within allotted time.

```
trajectory_execution:
 controller_connection_timeout: 60 # [s]
```

Listing 6.8. Connection timeout setting

6.4. Arm controller

To execute commands from MoveIt on the real arm custom controller was implemented. As a communication interface MoveIt requires implementing *FollowJointTrajectoryAction* action server that as a goal receives trajectory - desired joint angles and timestamps when they should be achieved.

In controller angles are converted to servo signals and, by taking into account timestamps, time of movement is calculated and correct command value is determined. In case of digital servos (used in shoulder pitch, shoulder roll and elbow joints) movement time command is playtime, which specifies how long the movement should be. For analog servos this value changes step of the control algorithm which is described in more detail in subsubsection 3.4.2.2. Then a pair of (position, value for time of execution) is sent to separate topics for each servo.

As an alternative I also tried using *JointState* message, which allows sending all values together, but when specifying more than three joints it didn't work (arm didn't respond to commands). It probably was caused by the large size of the message, although I tried reducing string names sent to simple names like J1, J2, ... which were shorter, but it still didn't help.

6.5. Object detection

To detect object on table I tested two approaches:

- **ARTag** - object was always in a fixed position relative to ARTag placed on the table. It was the initial approach I used for testing, as it was much simpler to implement. It has obvious drawbacks of being not flexible to various objects and different positioning. Achieved results of ARTag detection also wasn't totally accurate, so this detection also failed sometimes.
- **point cloud based detection** - I implemented it as an improvement, to achieve better reliability and flexibility. This method was based on simple point cloud analysis, it is far from perfect but it allowed to achieve better results.

Point cloud based object detection consisted of following steps:

- **range filter** - first points that are further away than threshold value were removed. As distance from the table is constant (considering my test setup) some points can be disregarded from the start. Later computations will consider a smaller subset of points, so they will be faster.

- **table detection** - next table is detected. First, using the RANSAC algorithm, the plane is fitted. It is working based on assumptions that the largest plane in the remaining point cloud belongs to the table. Then all the inlier points are selected. As they do not necessarily belong to the table (they could be just coplanar) some further filtration is necessary. First voxel filter is used to lower the resolution (the following calculations were too slow without it). To select only points belonging to table euclidean clustering is performed, table points are assumed to be in the largest cluster. As the final step bounding box of the table is calculated.

- **selecting only points on the table** - after the bounding box of the table was calculated points are filtered so that only points on the table (within detected x-y bounding box) will remain.

- **selecting object** - then on filtered points euclidean clustering is performed to find objects on the table. Object to pick is assumed to be closest to the edge of the table. As a position of the object calculated mass center is used.

Whole object detection was implemented in C++ using PCL library [33] for operations on point clouds. It implements all algorithms that I needed, such as RANSAC plane fitting, voxel filtering, range filtering and euclidean clustering.

In the final configuration I used point cloud based detection.

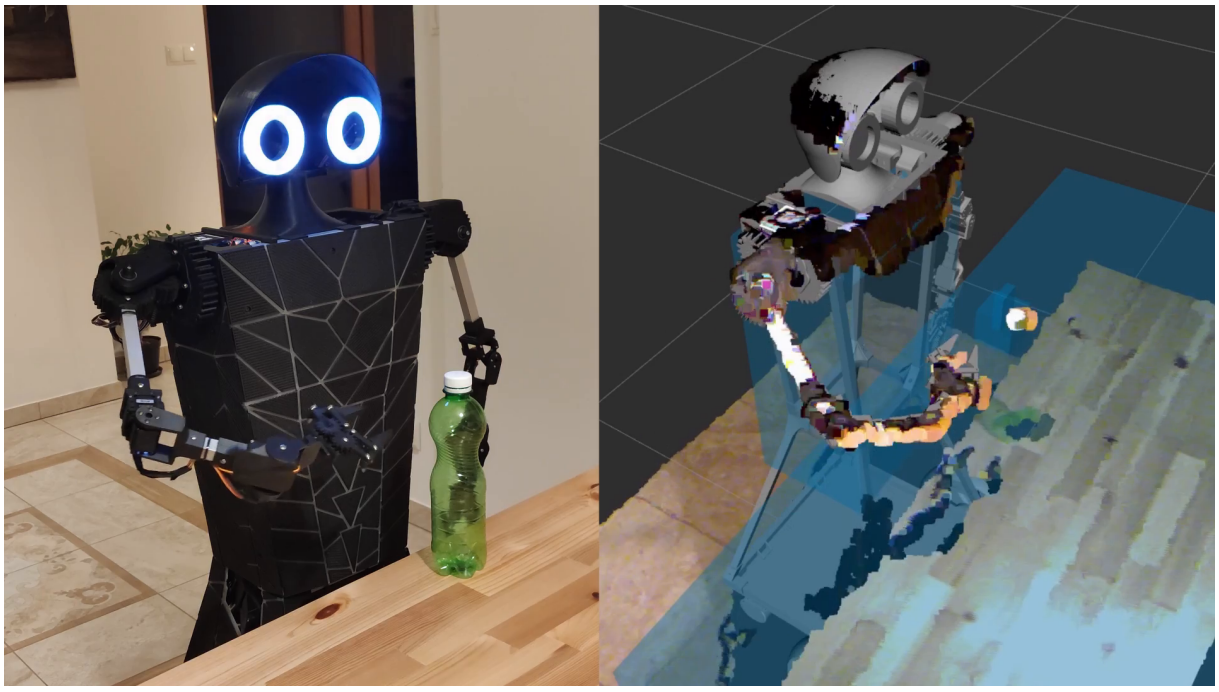

Figure 6.2. Robot during picking up bottle, with RViz visualization on the right

6.6. Arm precision testing

To test arm precision and repeatability, special test elements were designed, one for the gripper and one for the robot mount. As errors during picking up can be caused by detection to test only the arm, an element mounted on the robot is used as the reference point that arm was supposed to reach. Both elements had sharp tips, which made later measurements easier.

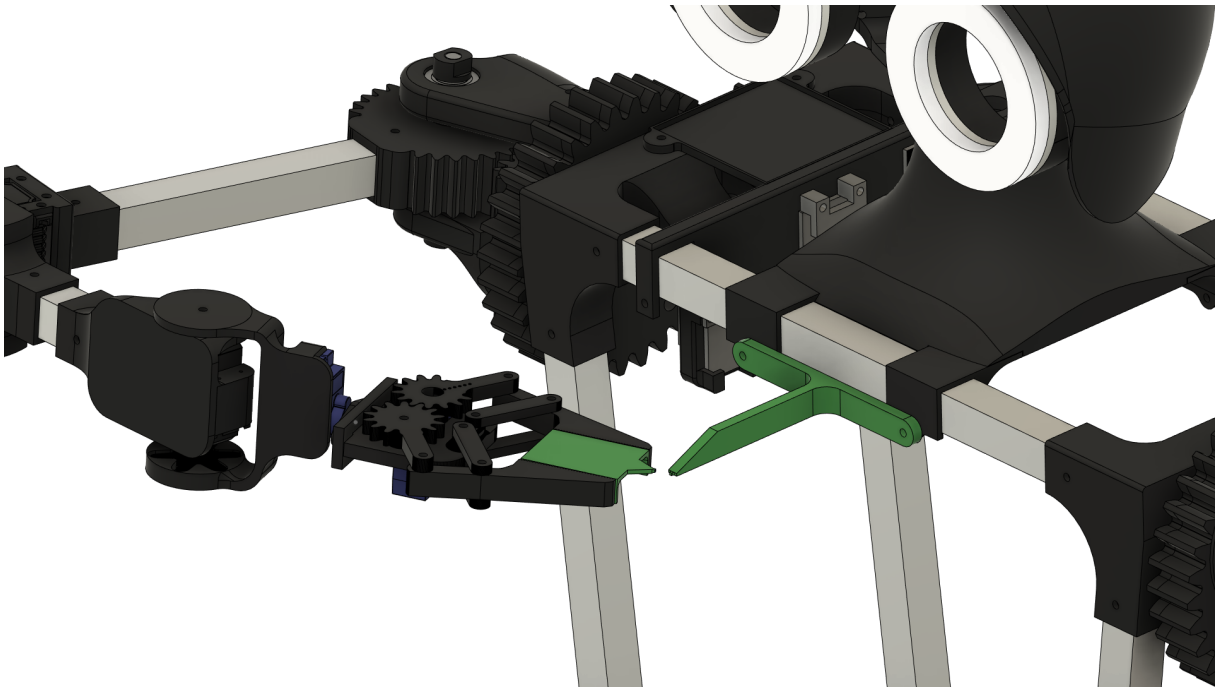

Figure 6.3. Elements designed for the arm precision testing

For all points orientation of the gripper was set to be (roll: $0°$, pitch: $0°$, yaw: $45°$). While only position was measured orientation should have not mattered, I tried configurations with *position_only* parameter for both KDL and BioIK, but performance was much worse and planner errors were bigger, so I used specified orientation with BioIK solver.

It should be noted that all values were manually measured using ruler and set square, so they are not totally accurate and should be considered only as approximations giving general sense of arms performance. It is also hard to estimate uncertainties, but I think that they are about $\pm\ 2\,\mathrm{mm}$ (basing them on observation of how tilting the ruler changed values). As I didn't intend this arm to be used in any precise application, this method of verification seemed sufficient.

As the planner calculated approximate positions, some part of the total measured error was planner error. For better insights I subtracted planner error from measured error to get only arm error.

$$a_{err} = m_{err} - p_{err} \tag{6.1}$$

,where

m_{err} – error measured during experiments

p_{err} – planner error, transform between gripper position after movement and destination point

a_{err} – arm error - all errors caused by inaccuracies in construction and signal calibration, without planner error

Arm error was calculated using equation 6.1 for each coordinate.

Test	**Setpoint (x [mm], y [mm], z [mm])**	**Planner error**			**Arm error**		
		x [mm]	**y [mm]**	**z [mm]**	**x [mm]**	**y [mm]**	**z [mm]**
1	(0,0,0)	0	0	2	25	8	-14
		0	0	2	26	8	-14
		0	0	2	28	8	-15
		0	0	2	28	8	-15
2	(100, 0, 0)	3	14	2	17	-8	-12
		3	14	2	18	-5	-15
		3	14	2	17	-5	-14
		3	14	2	18	-5	-12
3	(0, 100, 0)	1	0	3	17	7	-13
		1	0	3	17	7	-14
		1	0	3	17	7	-12
		1	0	3	18	8	-14
4	(0, -100, 0)	0	0	3	10	6	-13
		0	0	3	11	5	-14
		0	0	3	11	5	-14
		0	0	3	12	6	-13
5	(0, 0, -100)	18	1	61	18	7	-28
		18	1	61	19	7	-27
		18	1	61	21	7	-27
		18	1	61	19	8	-27

Table 6.1. Results of arm precision tests

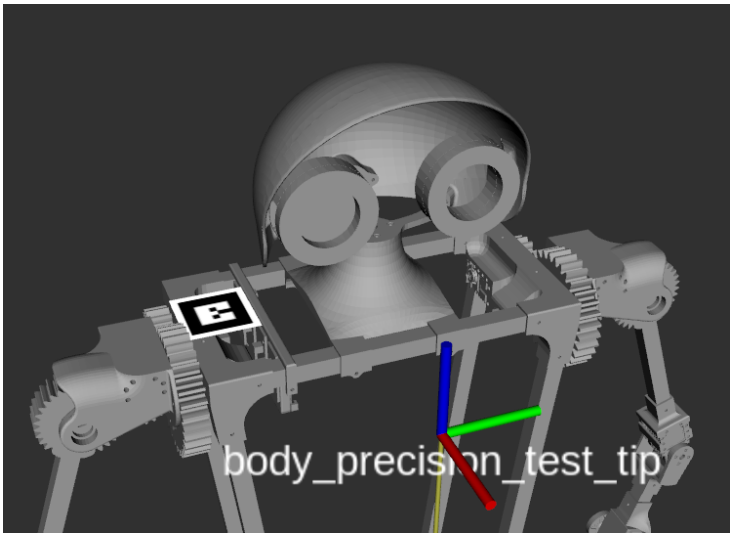

Figure 6.4. Coordinate frame of the test points, x (red) y (green) z (blue)

As it is visible in table 6.1 for some configurations planner was not able to find a close solution and error was quite large. For better understanding of arm error results I calculated mean and standard deviation for each test.

Test	Arm error mean			Arm error standard deviation		
	μ_x [mm]	μ_y [mm]	μ_z [mm]	σ_x [mm]	σ_y [mm]	σ_z [mm]
1	27	8	-15	1.5	0	0.6
2	18	-6	-13	0.6	1.5	1.5
3	17	7	-13	0.5	0.5	1
4	11	6	-14	0.8	0.6	0.6
5	19	7	-27	1.3	0.5	0.5

Table 6.2. Mean and standard deviation of arm error for every test

Resulting mean error is quite high. It is most likely caused by combination of several smaller problems:

- **differences between design and real construction** - model used for calculating inverse kinematics is based on design from Fusion 360. Due to tolerances not being exactly right, some small errors will be introduced when using real parts. For example in the shoulder pitch joint threaded rod is mounted in bearings and as the rod is slightly smaller than holes, some tilt of the shoulder is possible. Apart from 3D printed parts, which are quite precise, there are also manually cut aluminum tubes, which could lower the accuracy. Taking into account composition of few imperfections, they can add up and result in a larger error.

- **backlash in gears** - gears also has to be made with some tolerance and some backlash will be always present, which also lowers the final accuracy
- **calibration of the servos** - another potential source of error is setting what signal sent to servos was equal to the neutral position of the arm, which had to be done manually

All these causes combined resulted in quite large error and low accuracy. On the other hand precision was quite good - gripper was able to repeatedly reach a similar point, which is visible while looking at individual standard deviations for each test in table 6.2.

In table 6.3 I calculated total mean and standard deviation. Standard deviation in this case is much larger than for individual tests. It could be caused by the fact that position error changes depending on the angle configuration of the arm.

Total mean			**Total standard deviation**		
μ_x **[mm]**	μ_y **[mm]**	μ_z **[mm]**	σ_x **[mm]**	σ_y **[mm]**	σ_z **[mm]**
18	4	-16	5.3	5.3	5.7

Table 6.3. Mean and standard deviation of arm error for tests combined

6.7. Picking up bottle tests

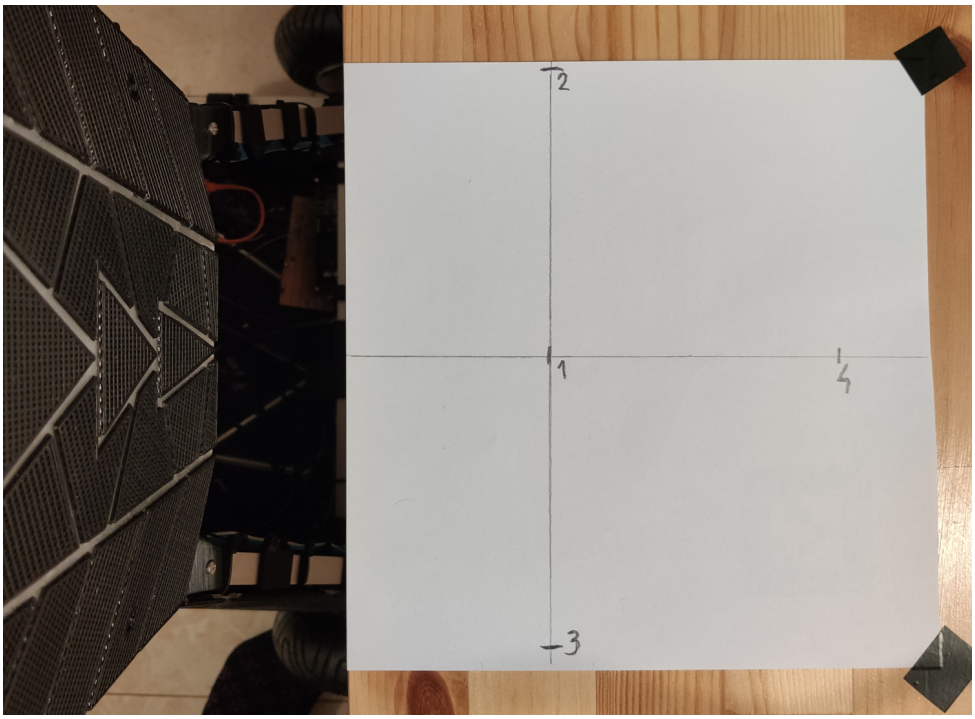

Figure 6.5. Picking up bottle test setup

To check the ability to pick up a bottle I marked 4 different points, on which I positioned the middle point of the bottle and checked if the robot would be able to pick it up. In the test I placed the bottle ($23\,\mathrm{cm}$ height) on the table with $73\,\mathrm{cm}$ height. Bottle cap had $3\,\mathrm{cm}$ diameter, and the gripper's maximum grasping size is $3.9\,\mathrm{cm}$.

Point 1 was positioned in the middle of the robot, $14\,\mathrm{cm}$ measuring from the body panel. I assumed that it is the central point, other points' positions were measured in respect to point 1.

Point	**Test**	**Planner error**			**Total distance error [mm]**
		x [mm]	**y [mm]**	**z [mm]**	
1	1	-0.6	0.3	0.3	0.7
	2	-0.2	-0.1	0.5	0.5
	3	-0.7	0.2	0.5	0.9
	4	-0.1	-0.1	0.5	0.5
	5	0.4	-0.2	0.1	0.5
2	1	0.5	0.1	0.3	0.6
	2	0.1	-0.1	0.3	0.3
	3	0.2	0.2	0.3	0.4
	4	0.3	0.2	0.3	0.5
	5	0.2	0.1	-0.1	0.2
3	1	-0.4	0.0	0.4	0.6
	2	-0.3	0.3	0.7	0.8
	3	0.7	-0.4	-0.3	0.9
	4	0.7	-0.3	-0.3	0.8
	5	0.4	0.2	-0.2	0.5
4	1	0.0	0.2	0.4	0.4
	2	-0.4	0.4	0.4	0.7
	3	1.0	-0.6	0.6	1.3
	4	-0.2	0.1	0.3	0.4
	5	0.0	0.1	0.4	0.4

Table 6.4. Planner errors during picking up bottle test

Point	Point's coordinates (x [mm], y [mm])	Was robot able to pick up bottle?				
		Test 1	Test 2	Test 3	Test 4	Test 5
1	(0, 0)	yes	yes	yes	yes	yes
2	(0, 100)	yes	yes	yes	no	no
3	(0, -100)	yes	yes	yes	yes	yes
4	(100, 0)	yes	yes	yes	yes	no

Table 6.5. Picking up bottle test results

In all but one test attempt planner error was below $1\,\mathrm{mm}$, in failed ones the error wasn't particularly large, so it wasn't the case for failure.

In test 4, point 2 bottle was tipped over. Most likely it wasn't caused directly by incorrect trajectory, but rather error between real and simulated gripper position. This problem can be fixed by making the model of a bottle added to the planning scene a little larger (a bottle is modeled as two cuboids, increasing the size of the lower one can help). Then the planner should create trajectories with larger safety margins, avoiding collisions with a bottle.

Failures in test 5, point 2 and 4, were caused by the gripper colliding with the bottle cap when the arm approached the bottle from above. In this case solving the issue may require mechanical changes in the arm's construction, such as changing gears for the ones with lower tolerances. Another possible improvement is more precise servo calibration, which is necessary to get constants used for converting joint angles to servo signals.

It is also important to mention that in point 4, in all successful test attempts, the bottle was picked up at a higher point of the bottle cap, which resulted in less firm grip than in other points.

6.8. Payload tests

To test the payload that the arm is able to lift I used a setup similar to the one described in the picking up bottle test, this time point 1 was $15\,\mathrm{cm}$ from the robot, but still in the middle. In every test I put the bottle in the same spot (point 1) and sent a command to lift it. To achieve various weights I added water to the bottle, weight of the bottle alone is $17\,\mathrm{g}$ and it wasn't a problem to pick it up. Next I checked $100\,\mathrm{g}$ (total weight of bottle and water). Robot was not able to pick it up by grippers friction alone, when the robot made contact only with the bottle cap (for bottle alone it was able to do it). I set the grasping point to be lower, below the bottle cap, so that a small edge around the bottle, below the bottle cap, was hooked on the upper part of the gripper. In this configuration the robot was able to pick up $100\,\mathrm{g}$.

Next I tried $200\,\mathrm{g}$, and in a similar way the robot was also able to pick it up. When trying to pick up a $250\,\mathrm{g}$ servo mounted in a gripper got damaged. So the maximum payload weight I was able to achieve was $200\,\mathrm{g}$.

Figure 6.6. Grip under bottle cap used in payload test

I wasn't able to verify it, but I think that the robot should be able to pick up even more weight, but some modifications to the gripper will be necessary. First gripper is mounted directly on the shaft of the gripper twist servo, as it is a micro servo, this connection is not too rigid and durable. Second improvement could be adding some material to clamps of gripper that will increase friction. Also servo with higher torque for gripper will be necessary.

7. Service Robot Controller

In this chapter the combination of the autonomous navigation and picking up objects will be described as an example real life use case - service robot that is able to bring a bottle.

7.1. Service robot controller structure

Service robot controller was implemented based on pick up and navigation systems described in preceding chapters, which were used as a kind of black boxes that can be controlled through actions. Service robot controller itself provides action that is used to control its behavior - as a goal request to start a task can be sent. It doesn't contain any additional information and acts only as a trigger. Service robot controller action also provides a way to cancel a task, which is quite useful.

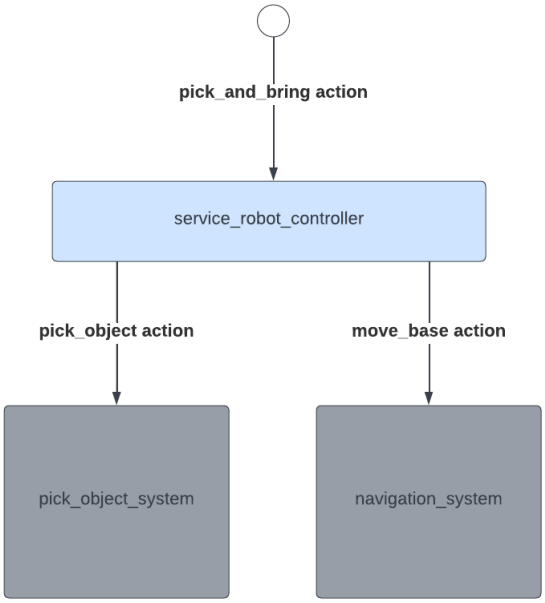

Figure 7.1. Diagram of the service controller

Before executing application it is necessary to create map of the environment and save two positions:

- **table position** - robot will go there to pick up object
- **home position** - where robot will return with object

Then controller can execute action, it consists of the following steps:

1. First table position is sent as a goal to the navigation system.

2. Once robot arrives at the table location it waits for stabilizing of ARTag detection and starts pick action execution.

3. When picking up a bottle is finished it sends the navigation system home position as a goal.

7.2. ROS-Mobile

During development I used terminal ROS commands to control the robot, which isn't user friendly. To create a better interface I used the ROS-Mobile [34] application for Android. It allowed me to prepare various configurations for operating the robot.

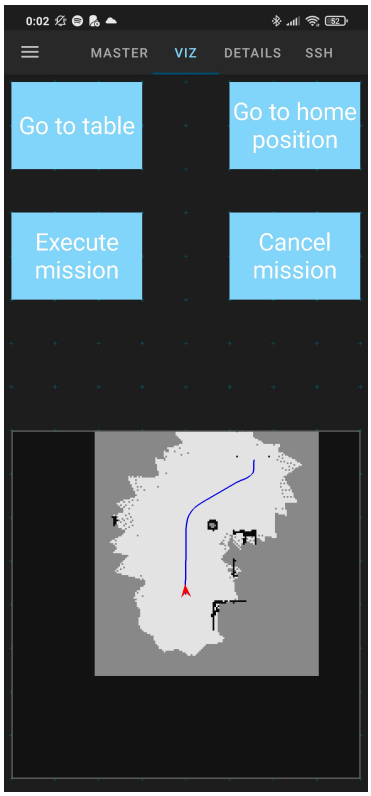

Figure 7.2. ROS-Mobile client view

Figure 7.3. ROS-Mobile mapping view

It still isn't something that could be practically used, but it allowed fast prototyping, which later could be implemented for example as a web service or dedicated application.

7.3. Environment setup

Due to sensor configuration of the robot it isn't really versatile, so provided service robot controller application required a special setup. As described in section 6.1 Kinect was mounted above a table and a bottle was placed near the edge of the table. Additionally cardboard box was placed underneath the table - robot's Kinect is not able to detect tabletop, which could cause collision during navigation, box prevents robot from bumping into the table.

7.4. Service robot controller tests

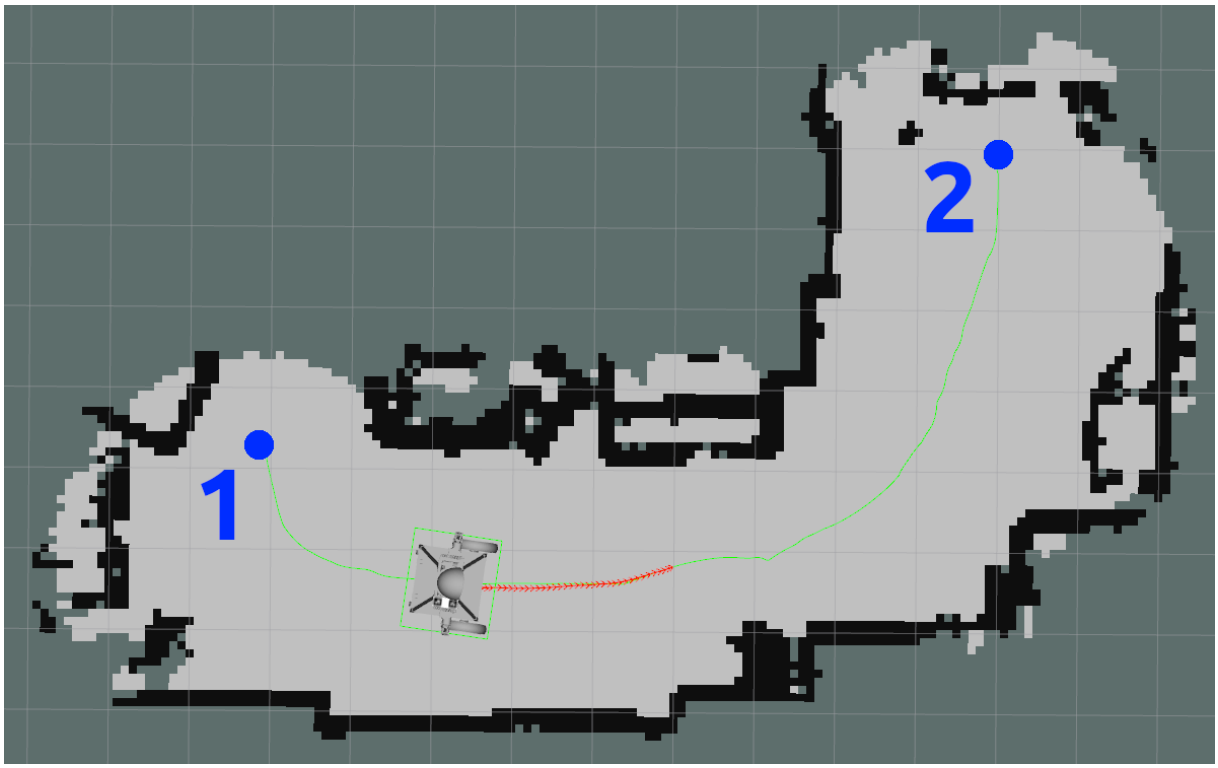

Figure 7.4. Map of the environment used during service controller tests. Points are destinations, point 1 is home position, point 2 is table position. Grid's scale is set to 0.5 m.

To test the service robot controller application I run it 15 times. All of them consisted of going from home position to table position, picking up the bottle and going back to home position. Out of 15 attempts 7 were successful and the robot delivered the bottle to home location, that gives around 47% success rate. 8 failures that happened had following causes:

- **failed driving up to table** (2 times) - this issue happened two times in the same place - the most narrow part of the path, when going from home to table position. Robot drove too close to the obstacle and part of the footprint collided with it. While in collision, the local planner couldn't

execute any commands. Footprint of the robot is rectangular, but in reality the shape should be a more complex polygon. Because the rectangular approximation in the back of the robot footprint is quite far from the real parts, the robot thinks that it will collide with an obstacle, while in reality there is still enough space. So using a polygon that will more tightly fit the robot should help. On the other hand it will increase computational load, which could be problematic. Another possible solution is to further tune the min_obstacle_dist TEB parameter.

- **bottle knocked over** (2 times) - these problems were caused most likely by inaccuracies in calculated gripper position (as tested in 6.6). Also it could be caused by inaccuracy in the planner, which I didn't record for these attempts. As for possible improvements, one time robot tipped the bottle over while approaching it from below, in this case modifying the bottle model could help - it is approximated by two cuboids, by increasing size of the lower one safe distance during planning should also increase and it will prevent collisions. Second failure happened when the gripper approached the bottle from above, in this case it will be necessary to fix inaccuracies in the arm.

- **failed to pick bottle** (1 time) - in this case the robot had too much offset in y direction and missed the bottle. Similar to the previous case it could be caused by arm inaccuracies or planner.

- **failed to find arm trajectory to bottle (robot too close to table)** (2 times) - in this case robot drove up too close to table and MoveIt wasn't able to find collision free trajectory to bottle. In reality the robot wasn't that close and movement was possible, but when adding table to scene its size is increased by a $10\,\mathrm{cm}$ safety margin to prevent movement too close to table. It is quite large and reducing it will help in such situations. Part of the reason why I set it to higher value, was that ARTag detection wasn't too reliable, so I needed higher safety margins, but since I was able to improve it safety margin could be decreased.

- **failed to find arm trajectory to bottle** (1 time) - in one case MoveIt wasn't able to find collision free trajectory, while the robot was in normal position, not too close to the table. In this situation I believe that it wasn't able to find a trajectory that didn't collide with the bottle. It could be solved by adding recovery behavior in which the robot rotates slightly, which could solve problems with arm configuration. In the current configuration a bottle is approached in a perpendicular direction, when unable to solve in this direction, planner could also slightly modify it. This option is now restricted due to the bottle cap being modeled as cuboid that is quite long, but changing it to cylinder it should unlock other grasping possibilities.

When considering the usefulness of this application it is also important to take into account how much time does the execution of the task take. To complete one test attempt it took the robot around 2 minutes, which I think is a good result, as the path from home to table position had around $6.5\,\mathrm{m}$.

As the full task of bringing a bottle consists of many smaller actions there are many things that can go wrong. Achieved success rate is quite low, but the issues that caused failure are still solvable and with more time spent on development success rate should increase. Navigation task is performing quite

Figure 7.5. Failure due to robot driving too close to obstacle

well and with some minor tweaks in parameters it should be reliable. Manipulation is more challenging and although software tweaks will improve success rate it may not be enough. Some improvement to the manipulator will be necessary, first thing could be more precise calibration, that is necessary for converting joint angle to serve signal. If that doesn't improve accuracy enough it may be necessary to reprint gears with lower tolerances, because backlash is quite large.

It is also important to note that I did some fine tuning of the table's position setting, and offsets applied to the bottle's position that compensate for the arm's inaccuracies. By improving the manipulator's accuracy fine tuning should no longer be necessary. Instead of setting the exact table position it might be better to provide a general location to which the robot goes and then, based on detections, moves closer to the table. One possibility is that the robot will have to find a bottle (which will require mounting an additional camera in the upper part of the robot). Another option is that after the robot arrives at the general location of the table, it should be possible to check the robot's position on the upper Kinect and, based on this detection, move the robot closer to the table.

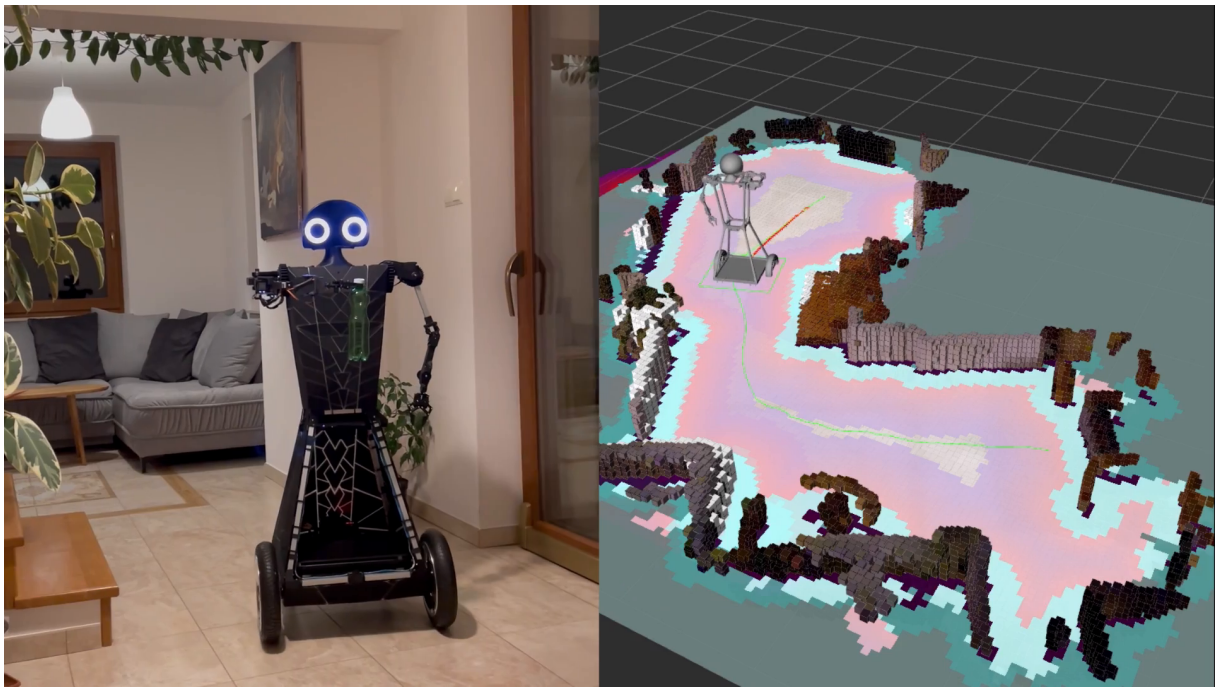

Figure 7.6. Robot delivering bottle

8. Conclusions

8.1. Development approach

The approach that I used for this project was the *spiral* model of development. Development process was iterative, focusing on the division between autonomous navigation (mechanically base of the robot) and manipulating objects (arm). Process of construction consisted of going back and forth between these two main parts. Firstly I focused on getting the robot to a rough working state as soon as possible - I created the first iteration of the base and arm. As they weren't perfect I kept improving them (creating second iterations of the design). Similarly working on software components I alternated between the two aspects of autonomous navigation and picking up objects. This approach is a good fit for R&D work and complicated projects, which was the case for this one.

8.2. Development tools

One of the tools used for development was an Arduino board. Although it isn't used anywhere in the final version, usually when adding a new component I first tested it on Arduino, later porting the solution to STM32 microcontroller. For example initially I had problems with I2C communication with IMU on STM32, which was working fine on Arduino and it helped me to debug the issue.

As a development tool for software components I used git with Github. Apart from the usual benefits of using version control systems, *Github Issues* functionality allowed me to better track things that had to be done. After achieving the first milestone of working service robot application I also started using the *catkin* versioning system. Each feature or bug fix was merged using the *Pull Request* option, which triggered *Github Action* bumping the proper version. It was important to use it after the initial stable version, because I was able to better track changes and localize issues if something becomes broken.

8.3. Results

Overall it was a great learning experience and the outcome was far better than I expected when initially starting this project. There are many things that have to be done before this robot could be useful in real world applications, but with this work I proved that without a large budget it is also possible to

create fairly modern construction. I think that all goals for this project were achieved, which was proven by a working service robot controller application.

8.4. Possible improvements

By the end of this thesis I was able to achieve a satisfactory level of performance, but the work will continue, as there are still many things that can be improved.

One of them is the noise problem when encoders are powered on from the battery. I was able to find a workaround, by powering it from the USB from a laptop mounted on the robot, but it is not a good solution.

Another problem was encoders having too low resolution. By changing them to ones with higher resolution the quality of motor control should be improved.

There are also many other improvements to the construction, but they will come with violation of the low cost constraint. When implementing them, tradeoff between increased usefulness and higher costs has to be taken into consideration.

It will be worth adding another camera in the upper part of the robot, the best option being Intel Realsense D435. With it the robot won't require additional camera setup and will be much more versatile being able to manipulate objects around the environment instead of only one specific place. Additionally it will remove inaccuracies caused by ARTag detection, which in result should provide better pick up accuracy.

Instead of using a laptop, a dedicated computing device could be placed inside of the robot. Possible option is Intel NUC computer, specific configuration could be chosen after analyzing computing power requirements. Pairing it with a larger battery could provide longer working time, which was a problem in my case, as the battery of my laptop provided only 1 hour of high performance usage, which slowed down robot testing.

Another improvement could be using BLDC hub motors instead of drill motors. They should provide better control accuracy especially at lower speeds. Motors that I used had problems when operating at lower speeds and they have quite high minimum speed. This is most likely caused by high friction caused by the high reduction ratio of the gearbox. Achieving low speeds is important for accuracy. Additionally motors used had really large backlash (around 24°), which further lowered accuracy when driving to destinations. With a low reduction ratio they are backdrivable, which is nice when trying to move a robot while powered off.

Also there is much to improve in robot's manipulator. One thing is increasing the number of DoF to 6 or 7 would result in a bigger operational workspace. Also the arm's payload is quite low, increasing it will allow for more real world applications. Adding joint that will allow changing the height of the manipulator will also increase workspace and robot's usefulness.

Only one Kinect sensor used for navigation also isn't ideal. When the robot is backing up it is doing it almost blindly - it can only take into account obstacles present on the previously created map. Robot's safety will be increased by adding more cameras or LiDAR to cover the whole area around the robot.

Bibliography

[1] Keenan A. Wyrobek et al. “Towards a personal robotics development platform: Rationale and design of an intrinsically safe personal robot”. In: *2008 IEEE International Conference on Robotics and Automation*. (May 19–23, 2008). 2008, pp. 2165–2170. DOI: *10.1109/ROBOT.2008.4543527*.

[2] *PR2*. URL: *http://wiki.ros.org/Robots/PR2* (visited on 2022-06-19).

[3] *PR2 Manual*. URL: *https://www.clearpathrobotics.com/wp-content/uploads/2014/08/pr2_manual_r321.pdf* (visited on 2022-06-19).

[4] Melonee Wise et al. “Fetch & Freight : Standard Platforms for Service Robot Applications”. In: *IJCAI-2016 Workshop on Autonomous Mobile Service Robots*. (July 11, 2016). 2016. URL: *https://fetchrobotics.borealtech.com/wp-content/uploads/2019/12/Fetch-and-Freight-Workshop-Paper.pdf*.

[5] *uStepper-RobotArm*. URL: *https://github.com/uStepper/uStepper-RobotArm-Rev3* (visited on 2022-06-19).

[6] Toshinori Kitamura. *Fusion2URDF*. 2020. URL: *https://github.com/syuntoku14/fusion2urdf* (visited on 2022-06-19).

[7] S. M. LaValle. *Planning Algorithms*. Available at *http://planning.cs.uiuc.edu/*. Cambridge, U.K.: Cambridge University Press, 2006.

[8] Sachin Chitta et al. “ros_control: A generic and simple control framework for ROS”. In: *The Journal of Open Source Software* (2017). DOI: *10.21105/joss.00456*.

[9] *Docker*. URL: *https://www.docker.com/* (visited on 2022-06-19).

[10] *ROS*. URL: *https://www.ros.org/* (visited on 2022-06-19).

[11] T. Moore and D. Stouch. “A Generalized Extended Kalman Filter Implementation for the Robot Operating System”. In: *Proceedings of the 13th International Conference on Intelligent Autonomous Systems (IAS-13)*. Springer, 2014. URL: *http://docs.ros.org/en/lunar/api/robot_localization/html/_downloads/robot_localization_ias13_revised.pdf*.

[12] *RTABMap*. URL: *http://introlab.github.io/rtabmap/* (visited on 2022-06-19).

[13] *Move Base*. URL: *http://wiki.ros.org/move_base* (visited on 2022-06-19).

[14] D. Fox, W. Burgard, and S. Thrun. “The dynamic window approach to collision avoidance”. In: *IEEE Robotics & Automation Magazine* 4.1 (1997), pp. 23–33. DOI: *10.1109/100.580977*.

[15] *DWA local planner*. URL: *http://wiki.ros.org/dwa_local_planner* (visited on 2022-06-19).

[16] Christoph Roesmann et al. “Trajectory modification considering dynamic constraints of autonomous robots”. In: *ROBOTIK 2012; 7th German Conference on Robotics*. 2012, pp. 1–6. URL: *http://files.davidqiu.com/research/papers/2012_rosmann_TEB%20Planner%20Trajectory%20modification%20considering%20dynamic%20constraints%20of%20autonomous%20robots%20[ROBOTIK%202012].pdf*.

[17] *TEB local planner*. URL: *http://wiki.ros.org/teb_local_planner* (visited on 2022-06-19).

[18] *rosserial_python*. URL: *http://wiki.ros.org/rosserial_python* (visited on 2022-06-19).

[19] *freenect_stack*. URL: *https://github.com/ros-drivers/freenect_stack* (visited on 2022-06-19).

[20] *ARTrackAlvar*. URL: *http://wiki.ros.org/ar_track_alvar* (visited on 2022-06-19).

[21] Ioan A. Sucan and Sachin Chitta. *MoveIt*. URL: *https://moveit.ros.org/* (visited on 2022-06-19).

[22] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. “The Open Motion Planning Library”. In: *IEEE Robotics & Automation Magazine* 19.4 (2012). *https://ompl.kavrakilab.org*, pp. 72–82. DOI: *10.1109/MRA.2012.2205651*.

[23] Nathan Ratliff et al. “CHOMP: Gradient optimization techniques for efficient motion planning”. In: *2009 IEEE International Conference on Robotics and Automation*. 2009, pp. 489–494. DOI: *10.1109/ROBOT.2009.5152817*.

[24] R. Smits. *KDL: Kinematics and dynamics library*. URL: *http://www.orocos.org/kdl* (visited on 2022-06-19).

[25] Philipp Ruppel. *BioIK*. URL: *https://github.com/TAMS-Group/bio_ik* (visited on 2022-06-19).

[26] Philipp Ruppel. “Performance optimization and implementation of evolutionary inverse kinematics in ROS”. MA thesis. University of Hamburg, 2017. URL: *https://tams.informatik.uni-hamburg.de/publications/2017/MSc_Philipp_Ruppel.pdf*.

[27] R. Diankov. *IKFast: The robot kinematics compiler*. URL: *http://openrave.org/docs/latest_stable/openravepy/ikfast/#ikfast-the-robot-kinematics-compiler* (visited on 2022-06-19).

[28] *Gazebo simulator*. URL: *https://gazebosim.org/home* (visited on 2022-06-19).

[29] *Small house world*. URL: *https://github.com/aws-robotics/aws-robomaker-small-house-world* (visited on 2022-06-19).

[30] *IROS 2014 Kinect Challenge*. URL: *https://github.com/introlab/rtabmap/wiki/IROS-2014-Kinect-Challenge* (visited on 2022-06-19).

[31] Kaiyu Zheng. *ROS Navigation Tuning Guide*. 2017. DOI: *10.48550/ARXIV.1706.09068*.

[32] Steve Macenski, David Tsai, and Max Feinberg. "Spatio-temporal voxel layer: A view on robot perception for the dynamic world". In: *International Journal of Advanced Robotic Systems* 17.2 (2020). DOI: *10.1177/1729881420910530*.

[33] Radu Bogdan Rusu and Steve Cousins. "3D is here: Point Cloud Library (PCL)". In: *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China: IEEE, 2011. DOI: *10.1109/ICRA.2011.5980567*.

[34] Nils Rottmann et al. *ROS-Mobile: An Android application for the Robot Operating System*. 2020. DOI: *10.48550/ARXIV.2011.02781*.