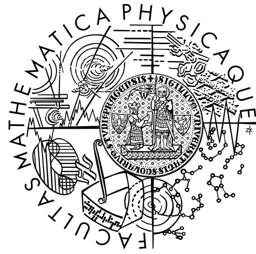


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Martin Červeň

Control system for badminton shuttlecock collecting robot

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis:	RNDr. David Obdržálek, Ph.D.
Study programme:	Computer Science
Study branch:	Artificial Intelligence

Prague 2023

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date .... signature of the author

Title: Control system for badminton shuttlecock collecting robot

Author: Martin Červeň

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. David Obdržálek, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Badminton is a racquet game played on court with shuttles made from feathers or plastic. Top players train with many shuttlecocks at once, which are fed by coach from hand. After a short training period, shuttlecocks are scattered around the court, which need to be picked up so that coach can feed them from hand. In this thesis we created software for autonomous robot that detects shuttlecocks with camera, estimates their position and picks them up. We implemented this as nodes in ROS middleware. During development we created simulated environment in Gazebo, and created plugin that simulates shuttle picking. We also created fully working picking mechanism of real shuttlecocks based on rotary brushes powered by motors, utilising 3D printing. Furthermore, we created and annotated dataset for object detection of over 2500 images and 18500 objects that we used for training and evaluation of state of the art neural network, that detects shuttlecocks from video. As part of our solution we developed ROS nodes that allows us to specify working area and area for filtering detections using RViz interactive markers.

Keywords: Autonomous robot control, Object tracking,Computer vision, Planning, Badminton

I would like to thank to my supervisor David Obdržálek for answering my questions and for his patience. I would also like to thank to my family for continuous support throughout my study.

Contents

1 Introduction 4
1.1 Goals of the thesis . . . 5
1.1.1 Detection and recognition of shuttles . . . 5
1.1.2 Control system . . . 5
1.1.3 Map . . . 5
1.1.4 Planning . . . 6
1.1.5 Movement . . . 6
1.1.6 Visualisation . . . 6
1.1.7 User Interface . . . 7
1.2 Structure of the thesis . . . 7

2 Background 8
2.1 Game of Badminton . . . 8
2.1.1 Shuttlecock . . . 8
2.1.2 Badminon court . . . 9
2.2 Training . . . 10
2.3 Shuttle picking . . . 11
2.4 Environment . . . 12

3 Related work 14
3.1 Fruit picking robots . . . 14
3.1.1 Cucumber picking robot . . . 14
3.1.2 Strawberry picking robot . . . 15
3.1.3 Kiwi picking robot . . . 16
3.2 Plant polination . . . 17
3.2.1 Pepper picking robot . . . 18
3.3 Sport mobile robots . . . 18
3.3.1 Tennis ball picking robot . . . 18
3.3.2 Golf ball picking robot . . . 19
3.3.3 Autonomous Table Tennis Ball Collecting Robot . . . 20
3.3.4 Badminton . . . 21

4 Analysis 22
4.1 Application architecture . . . 22
4.1.1 Monolithic application . . . 22
4.1.2 Using ROS . . . 22
4.2 Hardware . . . 23
4.2.1 Sensors . . . 26
4.2.2 Robotic platform . . . 27
4.2.3 Other . . . 27
4.3 Software . . . 28
4.3.1 Control system . . . 28
4.3.2 Shuttle recognition . . . 31
4.3.3 Mapping . . . 32
4.3.4 Planning . . . 32

4.3.5 Visualisation . . . 32
4.3.6 User interface . . . 33

5 Proposed solution 34
5.1 ROS . . . 34
5.2 Gazebo . . . 35
5.2.1 Preparing the simulation . . . 35
5.3 Control system . . . 35
5.4 Mapping and localisation . . . 36
5.5 Navigation and Planning . . . 38
5.5.1 Mapping . . . 39
5.6 Movement and shuttle picking . . . 39
5.7 Computer vision . . . 40
5.7.1 Object recognition . . . 41
5.7.2 Training neural network . . . 41
5.7.3 Position estimation . . . 42
5.8 Visualization . . . 42
5.8.1 RViz . . . 42
5.9 Shuttlecock picking . . . 43
5.10 User interface . . . 44

6 Implementation 45
6.1 Launchfiles . . . 45
6.2 Neural network . . . 45
6.2.1 Data acquisition . . . 45
6.2.2 Dataset creation . . . 46
6.2.3 Deployment on robot . . . 49
6.3 Visual processing . . . 49
6.4 Visualisation . . . 51
6.5 Control system . . . 51
6.5.1 Concurrent container in SMACH . . . 53
6.5.2 Control System . . . 53
6.6 Gazebo simulation . . . 54
6.6.1 Gazebo Plugins . . . 55
6.6.2 Sensor plugin for picking shuttlecocks . . . 55
6.7 Picking system . . . 56
6.7.1 3D printed parts . . . 58
6.7.2 Other parts . . . 60
6.7.3 Iterative design . . . 60
6.7.4 Motor control . . . 62
6.7.5 Arduino ROS node . . . 63
6.7.6 Arduino Leonardo . . . 63
6.8 Working area customization . . . 64
6.8.1 Allowed area . . . 64
6.8.2 Allowed detection area . . . 65
6.8.3 Waypoints . . . 66
6.8.4 Home position . . . 67
6.8.5 Detection area visualization . . . 68

7 User documentation **70**
7.1 Setup 70
7.2 Mapping the court 70
7.3 Autonomous working 71

8 Results **72**
8.1 Shuttlecock detection 72
8.1.1 Evaluation metrics 73
8.1.2 Training and results 74
8.2 Picking mechanism 77
8.3 Allowed area 78
8.4 Detection area 80
8.4.1 Discussion 80

9 Conclusion **82**

Bibliography **84**

List of Figures **89**

Appendix A Installation documentation **93**
A.1 Robot 93
A.1.1 Jetson Xavier NX 93
A.1.2 ROS 93
A.1.3 Jetson - inference 93
A.1.4 Kobuki base 93
A.1.5 Other dependencies 94
A.1.6 Cameras 94
A.1.7 Vision 94
A.2 Notebook 94
A.3 Source code, dataset and other files of our solution 95
A.3.1 Source code for robot 95
A.3.2 Shuttlecock dataset 95
A.3.3 Dockerfile 95
A.3.4 Yolo node 95

Attachments **96**

1. Introduction

Badminton is a racket game for two or four players played on indoor court by striking projectiles called *shuttlecocks* or *shuttles* (Figure 1.1) over the net. Shuttlecocks have conical shape where top part is made from cork and bottom part called skirt is made from natural feathers or plastic. Top players train with many feathered shuttlecocks which are fed by coach from hand. After a short training period there are many shuttlecocks scattered around the court which need to be manually picked up by players and coach and arranged in rows so that coach can use them again. This *manual* and *monotonous* labor takes lot of time, which if automated, could be spent on more *intensive training* (Figure 1.2) or explaining next exercise. In this thesis we propose control system for autonomous robot that picks these scattered shuttlecocks. We also analyse other necessary parts to design and built such robot. To achieve goal of the thesis we need to design a solution that would to have:

- *Computer vision* to *sense* and *recognize* shuttlecocks as objects in the real world that needs to be picked up.
- *Control system* to decide what robot should do. We want this to be *explainable*, i.e. we want to know exactly in what state is robot currently in, for example, planning, picking, moving to the next goal.
- *Mapping* the environment to create a *map*, with whitch it can then localise itself, and mark other objects of interest, such as shuttlecocks.
- *Planning* of the path subject to constraints of the *imperfect information* robot gets about the world from sensors and thus create plan.
- *Movement* to get from one place where it needs to pick a shuttlecock to the next place.
- *Visualisation* of the map representing the environment that robot had created and give commands and check settings and variables using
- *User interface* with basic ability to start and stop the robot, since robot should be otherwise autonomous.

Figure 1.1: Feathered shuttlecock

Figure 1.2: Multi-shuttle training.

1.1 Goals of the thesis

The aim of our thesis is to create control system for autonomous robot that picks badminton shuttlecocks. This can be divided into following goals:

Goal 1.) detection and recognition of shuttles

Goal 2.) control system

Goal 3.) create internal map of environment and position of shuttles

Goal 4.) plan how to pick shuttles

Goal 5.) movement

Goal 6.) visualisation

Goal 7.) user interface

1.1.1 Detection and recognition of shuttles

We need to sense where shuttles are located relative to the robot, so we can generate movement instructions for the robot to move close enough to the shuttlecocks to pick them up. Because we are using camera as the input, we need to be able detect shuttlecocks in images. This could be done by various approaches, most common and successful nowadays is to train *deep neural network*. For this a lot of training data is needed. Luckily, there exists *pretrained* neural networks. They are trained on large image datasets such as MS COCO: Common objects in context [1].

1.1.2 Control system

We need to be able to tell what the robot is doing at each point in time [2]. This does not mean that software needs to run in one process, on the contrary we need many processes running simultaneously. But just as well human that tells that he is studying for exam, can be holding pen and writing and thinking about math problem, overall state would be studying. Thus if robot state is "moving to shuttle", it could be simultaneously checking sensor inputs for human stepping into his path and stopping to not hit human or plan path around him. Control system would be something that tells us what is the robot doing, but under the hood, multiple processes and programs could be running.

1.1.3 Map

After the robot sensed positions of shuttlecock from visual input, It needs to remember them somewhere. It could be just a list of coordinates (x,y) or shuttles relative to the robot, or to some fixed frame. Since robot moves, it also needs to keep its position in the map, so it needs to have sense of environment around it. We could give robot a man-made map, but since we want to use it at many different courts, we want it to create a map itself. This problem of creating map and localising itself is called *Simultaneous localisation and mapping* or SLAM [3].

It can be solved by various approaches, most commonly probabilistically from sensors, trying to estimate most probable location given previous sensed parts of the environment. Things are easier using simpler sensors like ultrasound or 2D lidar, since there are not many data points to match, and more complicated using cameras, because then it needs extract only some interesting points (landmarks) from images to match between frames, since matching many megapixels would be wasteful and also computationally very hard. Visual SLAM [4] from sequence of images works then by searching interesting points from each image and then trying to match them between successive frames and trying to infer position changed. This is also called visual odometry. Some algorithms also use fact that many cameras nowadays have *inertial measurement unit*-IMU built in, so they can sense direction of where camera had been moving between frames. Many mobile robots have wheel odometry and this can be used in estimating match between frames or helpful in estimation of position. Lastly, there is interesting notion of *loop closure*, where if robot sensed the same scene two or multiple times, its error of localisation should get smaller because it knows where it is more accurately, as opposed thinking of it as new location entirely because it didn't know it is at previously visited location.

1.1.4 Planning

In an ideal setting, we would have knowledge about all coordinates of shuttles relative to some frame (for example map), but since our robot looks at the world from camera mounted on itself, it only sees part of the world at the time. Therefore we do not have perfect information about all the shuttle coordinates, and have to build this knowledge iteratively. We can assume that court is perfectly flat plane and thus we can only care about (x,y) coordinates.

1.1.5 Movement

Badminton courts are flat surfaces, usually made of rubber, wood or plastic materials, and are therefore suitable for *wheeled robot*. We can assume that we have approximate locations of shuttles from vision system, then we need to process them using planner to get sequence of positions to visit, i.e. goals. We could use only one step to plan movement of robot from shuttle positions and map, but robot movement is usually done with respect to map, we also need to take care that robot does not go off somewhere or hit anything. If we have a map, then in this map we can mark safe space for robot, and unsafe. Then we have next goal to go, and we give it to motion planner and it gives commands for wheels to robot platform. We could have feedback from wheel odometry and cameras with respect to map so we can navigate safely.

1.1.6 Visualisation

We would like to visualise input from robot sensors, such as cameras. Images from cameras would be used to build map for the robot, so we could also visualise this map in 3D interactive manner. Robot should also be displayed as 3D model.

Our objective is to pick up badminton shuttlecocks, so they should be visualised as well, for example as their *estimated location* by points or *bounding cubes*. Because we want to pick up shuttles as fast as possible, we need to plan shortest path, this path could be visualised as lines between estimated locations of shuttlecocks. We do not want robot to hit humans or other parts or environment so differentiation of safe and unsafe space by colours would be helpful. Visualisation is also needed for remote control of robot and debugging.

1.1.7 User Interface

For controlling robot, an easy user interface would be necessary. Since our main goal is to create control system with all the necessary software, we are content with creating simple user interface using premade controls such as buttons, sliders, windows and graphs.

1.2 Structure of the thesis

In the second chapter, **Background**, we describe the game of Badminton, its rules and history. We describe environment of where robot would operate.

In the third chapter, **Related work**, we describe already made solutions that employ autonomous robots from different domains such as agriculture and other sports.

In the fourth chapter, **Analysis**, we describe decision process that we used to select architecture of our control system, which frameworks and technologies we used and which algorithms we chose to use.

In the fifth chapter, **Proposed solution**, we present how we designed our control system. We also detail what 3rd party packages we used and how we interconnected them.

In the sixth chapter, **Implementation**, we describe what software and hardware we developed as part of our solution.

In the seventh chapter, **User documentation**, we show how to setup robot for picking shuttlecocks at badminton court.

In the eight chapter chapter, **Results**, we will present results of training neural network on our dataset, shuttlecock picking mechanism and other features of our solution.

In the last, ninth chapter, **Conclusion**, we summarise what we have achieved and we will outline the future work that could be done and was not part of goals of our thesis.

Installation documentation will be presented in the Appendix A. We will describe necessary software requirements for robot and notebook.

2. Background

In this chapter we introduce the game of Badminton. We describe how players train with many shuttles at once and when the problem of picking shuttles arises in training. Robots could up speed training substantially by picking shuttlecocks on the ground instead of humans. We also show what is normal environment for robot for this task.

2.1 Game of Badminton

Badminton is a racket game for two (*singles*) or four (*doubles*) players. It is played by hitting feathered projectile called *shuttlecock* by light racket nowadays made from carbon. Courts have standard dimensions, as shown in Figure 2.3. For singles court is shorter on sides, and for doubles it is wider. We will not be going into details of the rules of the game any further since we are interested in picking up shuttles during **training**.

2.1.1 Shuttlecock

According to *Laws of badminton*, published by Badminton World Federation [5], badminton shuttlecock is made of 16 bird feathers arranged in cone with tips of feathers glued to cork head (Figure 2.1). It can be also made from nylon or other synthetic materials, but flight characteristics are different, and they are not used for serious competition. The tips of the feathers shall lie on a circle with a diameter from 58 mm to 68 mm. Weight of shuttlecock should be between 4.74 to 5.50 grams.

Figure 2.1: Yonex feather shuttlecock

Shuttlecock is aerodynamically different to balls used in other racket sports such as tennis, ping pong, or squash. Because of its conical shape and holes between feathers, it creates small *vortices*, shown in Figure 2.2, that increases air drag as it travels further and abruptly decelerates [6]. This means that it

hard to predict where shuttle lands, and players have to train for many years to develop intuition about this.

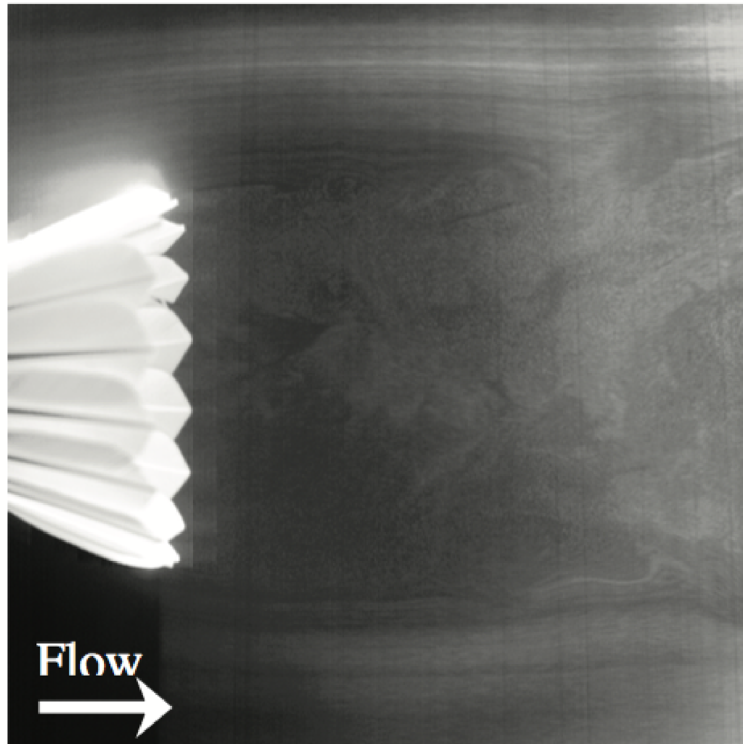


Figure 2.2: Vortices create drag, from [6]

2.1.2 Badminon court

Badminton court is rectangular area marked by perpendicular lines. Court for doubles is slightly larger at the sides, similar to tennis. Total length of court is 13.4m and width 6.1 m [5], shown in Figure 2.3.

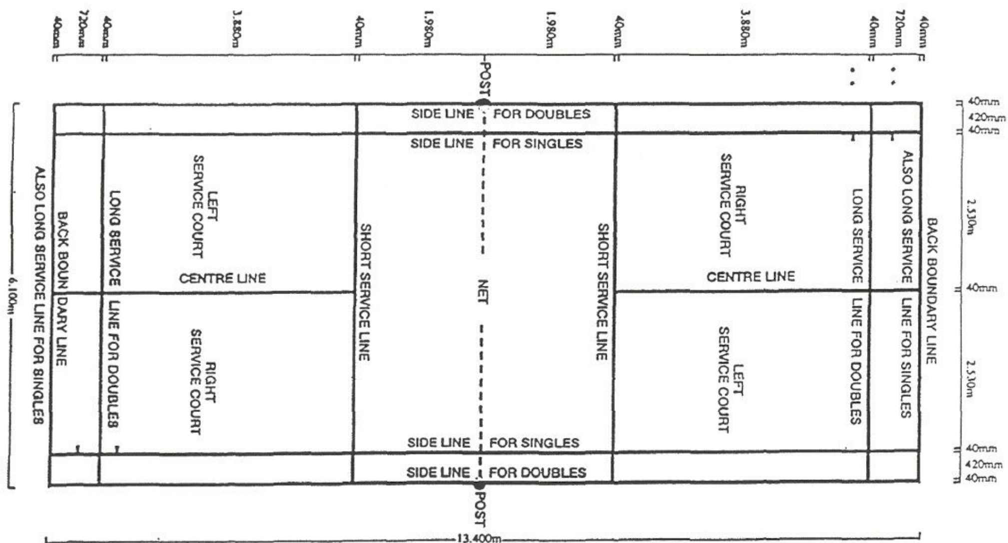


Figure 2.3: Badminton court dimensions

2.2 Training

Badminton is both skill based and fitness based game. Players spend lot of time on court practicing and polishing shots and also spend lot of time in gym working out. Top players at international and national level do *multishuttle training* (Figure 2.4), which consists of using many shuttles instead of just one. This has many benefits such as:

- more pressure to simulate harder opponent,
- more repetitiveness, i.e. player have to smash 20 times, or play netshot 20 times in a row and coach can observe shortcomings or these strokes and correct them,
- longer exercise to increase *stamina*
- train explosiveness,
- random shuttle throwing to train *reflexes* and improve *reaction time.*

Coach can feed shuttles with more frequency than is normally possible and therefore creating **high pressure** situation and increasing players reflexes, fitness and coordination in process.

Players and coaches can see where shuttles actually land, this is not possible with one shuttle since it is constantly in play, and in top level of badminton, centimeters matter. Players think that they are hitting perfect shots to the sideline and in fact they are hitting well into the court, and seeing where the shuttle actually lands helps a lot.

Figure 2.4: Picture of Kento Momota from Japan,currently no.1 player in the world, practicing with former Korean gold olympic medalist, Japan Head Coach Park Joo-Bong

2.3 Shuttle picking

During training, after using many shuttlecocks at once, players have to *manually* pick up shuttles and arrange them into rows to be used again in next exercise. Shuttlecocks are delicate, and since one costs 1-2 euros depending on the quality, they also have to take care to not damage then unnecessarily in the picking process. It also takes lot of *effort* and *time* to pick them up by hand and arrange them into rows (Figure 2.5).

We empirically tested how fast can one person arrange 100 shuttles in rows and it took around 5 minutes. This train-pick pair happen around 4-8 times per hour depending on how fast coach is feeding, thus the overall time can be up to *two thirds* of the actual training time spent.

Players thus spend 2/3 of their time picking shuttles. This time could be spend for more training. Moreover, amateur players usually have to pay for courts and therefore this would also save 2/3 of their money.

Therefore if picking of shuttles could be automated, players and coaches could spend *more time practicing* or take a break for drinking or talk about next exercise.

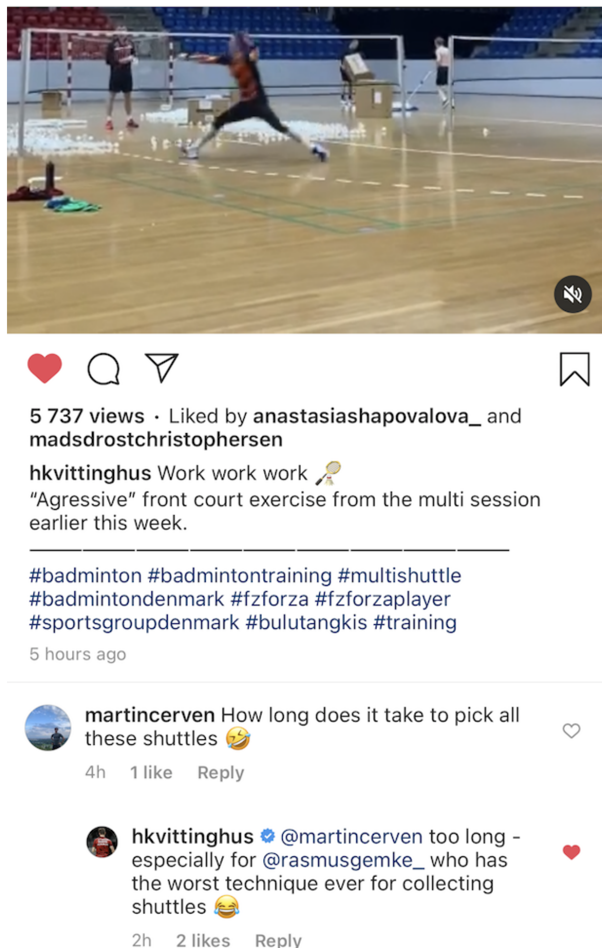


Figure 2.5: Hans-Kristian Vittinghus, no. 20 singles player in the world [7], responds to the author that even pro players like Rasmus Gemke (no. 12) lose time in training due to slow shuttle picking technique

2.4 Environment

Shuttlecock collecting robot will be used on badminton courts, doing its work alongside humans. This is *natural environment* for humans, not environment crafted for robot.

There are environments specifically crafted for robots, such as warehouses as shown in Figure 2.6. The robots and human worker areas are *separated*. Robots thus can roam freely without need of giving care to humans. They navigate themselves by going by QR codes glued to floor. They arrive at current *pod* and deliver it to the humans at the side of warehouse, separated by fence [8].

Figure 2.6: Kiva Robot, now Amazon robotics

Example of robots that works in human/natural environment are robotic vacuum cleaners such as Roomba, shown in Figure 2.7. It does not use any artificial landmarks such as QR codes, but instead it moves randomly or use SLAM.

Figure 2.7: Roomba vacuum cleaner

We will be using our robot on court in the public halls, so we won't be able to install birds eye camera, although in future this could be possibility.
We also want to use robot on different courts as shown in Figures 2.8, 2.9. This means that we cannot use any other cameras stationed on tripod for example, or birds eye view camera on roof. Every sensor should be on robot.

Robot should also adapt to changes in *illumination*, cast shades, or number of light sources. Color of the court also shouldn't pose problem for navigation or visual recognition of shuttlecocks.

Badminton courts are made from rubber, hard wood, soft rubber, or plastic. They can cost up to 20 000 euro, thus robot should not be excessively heavy or make markings when moving or picking shuttlecocks or not damage court surface or equipment otherwise.

Courts are located in large halls with as much as 20 courts in one hall, so robot should not venture from his assigned court, it needs to know which courts he is assigned to, and not disturb players on the other courts.

Figure 2.8: Example of green court

Figure 2.9: Example of hardwood court

3. Related work

In this chapter we present related work. Similar technologies that we listed in goals in introduction are being used in *agricultural robots*. Furthermore, mobile robots were created for other sports similar to badminton. We found that advances in vision correlates to development in robotics. Also cheaper computers and sensors such as stereo cameras are widening the scientific community from large corporations such as automobile industry to smaller companies and universities.

3.1 Fruit picking robots

One field where robots, picking and vision is used, is agriculture, mainly fruit picking. There is need to visually identify fruit from *background* to correctly position the arm with *gripper*. There is also tendency to select fruits by *ripeness*, mainly colour and size is used, for example green tomatoes are not picked, and red are because red colour is associated with ripeness. Similarly, *size* of the fruit could be used, in asparagus, if the sprouts are between some size, they are cut and picked.

3.1.1 Cucumber picking robot

In 2002, researchers in Netherlands developed cucumber picking robot [9], shown in Figure 3.1. It takes long time to pick, around 90s. Robot consists of 6DOF robotic arm on the rails. It has camera mounted on end effector.

- *Detection* of fruit, shown in Figure 3.2, is based on different reflectances of leaves and cucumbers. 3D localisation is made by taking images from different position by sliding robot across rails.
- It moves along greenhouse by rails. This also means that it can safely take pictures from different positions and do 3D reconstruction since it moves only in one known direction.
- Ripeness is estimated by measuring volume and thus weight from images. Authors report 95% accuracy of this method.
- Gripper grips cucumber with suction.
- Cuts cucumber with thermal knife, so transmission of viruses is minimised, and freshness of fruit is preserved.
- *Slow* - 45s on average to pick a cucumber, 10s is required for commercial use.

Figure 3.1: Cucumber robot

Figure 3.2: Segmented cucumbers

3.1.2 Strawberry picking robot

Agrobot E-Series (Figure 3.3) is a robot for **strawberry picking** developed in Spain[10]. There is not much information to find about this robot, but youtube video suggest it works in practice. https://www.youtube.com/watch?v=M3SGScaShhw

Figure 3.3: Agrobot E-Series

Problem of picking strawberries by robotic hands similar to the Agrobot could be solved by method developed by Zhang et.al [11]. They applied R-CNN algorithm for detection of strawberries. They also devised method for estimating **picking points**, shown in Figure 3.4.

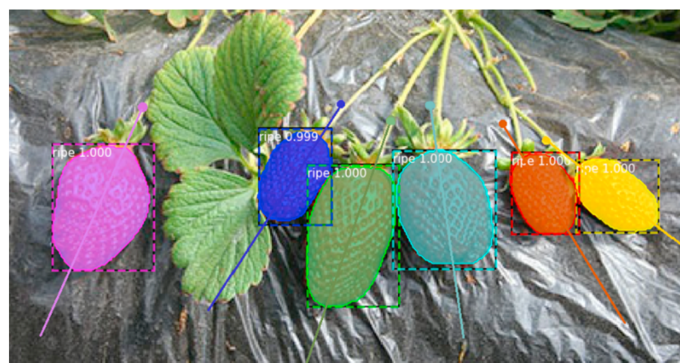


Figure 3.4: Bounding boxes around recongized strawberries with solid points representing picking points, from [11]

Picking point (Figure 3.5) was calculated by taking left and right points of strawberry that collided with bounding box (**a** and **b**), then drawing line **d**, computing barycenter **e**. For each countour point **c** above line they split the image and tested for similarity. This means that if the fruit is bent from its stem, picking point found by this algorithm will be off, like the orange strawberry shown in Figure 3.4. It could be interesting to use deep learning segmentation to also get stems and choose picking point only located on the stem.

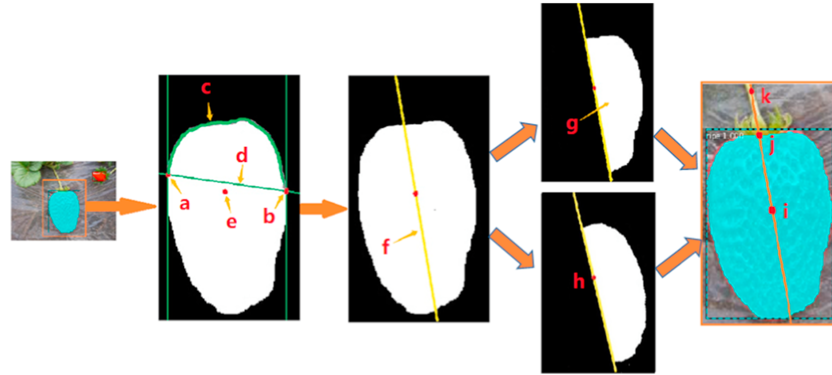


Figure 3.5: Creation of picking points, from [11]

3.1.3 Kiwi picking robot

In 2019 team from New Zealand made **kiwi picking robot**[12] with four grippers[1,2]. It is using ROS to manage messages between four grippers (Figure 3.6) . It is also using R-CNN.

- Robot has four robotic hands, calibrated before running with *Diamond Markers*[3].
- They use one upward looking camera, from it *plan* for four harvesting arms is made so that they don't collide between themselves.
- Rugged base, in development for at least 10 years.
- Fruit recognition done by adapted VGG-net16 network[4]. Trained only on 48 hand labeled kiwi images, network was *pretrained* [5] on PASCAL VOC dataset[13]. After locating fruit, blob detector is run on each fruit to locate center of fruit for grippers (Figure 3.7).

[1] https://www.roboticsplus.co.nz/kiwifruit-picker

[2] https://www.youtube.com/watch?v=b4L-oMd0yVk

[3] https://www.docs.opencv.org/master/d5/d07/tutorial_charuco_diamond_detection.html

[4] https://github.com/shelhamer/fcn.berkeleyvision.org/tree/master/pascalcontext-fcn8s

[5] Further training with custom data with already trained weights: https://stats.stackexchange.com/questions/193082/what-is-pre-training-a-neural-network

Figure 3.6: Kiwi robot with four arms

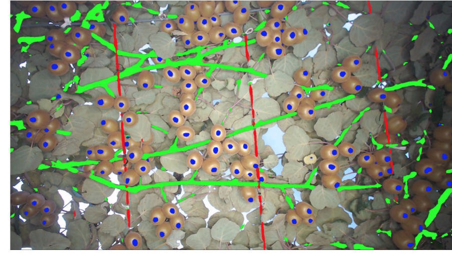


Figure 3.7: Visual output of network and blob detector

3.2 Plant polination

BrambleBee [14] is robot for **plant polination** (Figure 3.8). It is using Clearpath robotics Husky platform [15].

- Interesting approach to recognising flowers, first they ran naive bayes classifier on pixels, and after extracting patches of possible flower areas they run neural network to weed out false positives.
- Motion of arm - used MoveIt ROS package[16] and faster trac_ik [6] library
- Did not solve orientation of flowers, only touched ArUco marker instead of flowers.
- Camera at the end effector to guide visual servoing.
- Velodyne 3D lidar [7] for SLAM inside greenhouse.

Figure 3.8: BrambleBee

[6] https://bitbucket.org/traclabs/trac_ik/src/master/
[7] https://velodynelidar.com/products/hdl-32e/

3.2.1 Pepper picking robot

Arad et.al. developed **sweet pepper** picking robot SWEEPER [17] using *ROS*[8] (Figure 3.9) . They also had 4.3 mil. euro funding from European union [9].

- RGB-D camera Fotonic F80 [10]
- Arm 6DOF FANUC LR Mate 200iD [11]
- It used MoveIt [16] ROS package.
- End effector is shown in Figure 3.10.
- Does not use neural networks for pepper detection, but instead use simpler algorithms that find peppers by color. They rationalize this by faster FPS which is useful for visual servoing [18].

Figure 3.9: Sweet pepper robot

Figure 3.10: Closeup of end manipulator

3.3 Sport mobile robots

Higher level of mobility is needed in sports, because unlike fruit the objects robot wants to pick are not in the same position, and are not constrained by environment, i.e. growing from the same tree. Thus they need to have mobile base, more advanced sensors to accommodate possible dynamic environment, for example players on court.

3.3.1 Tennis ball picking robot

Wang [19] proposed mobile robot acting as tennis ballboy (Figure 3.11). It is using ROS and is build on RC car chassis with stereo cameras.

[8] Robotic operating systemhttps://www.ros.org/
[9] https://cordis.europa.eu/project/id/644313/results
[10] Sweden company, appears to be out of business.
[11] https://www.fanuc.eu/si/en/robots/robot-filter-page/lrmate-series/lrmate-200-id

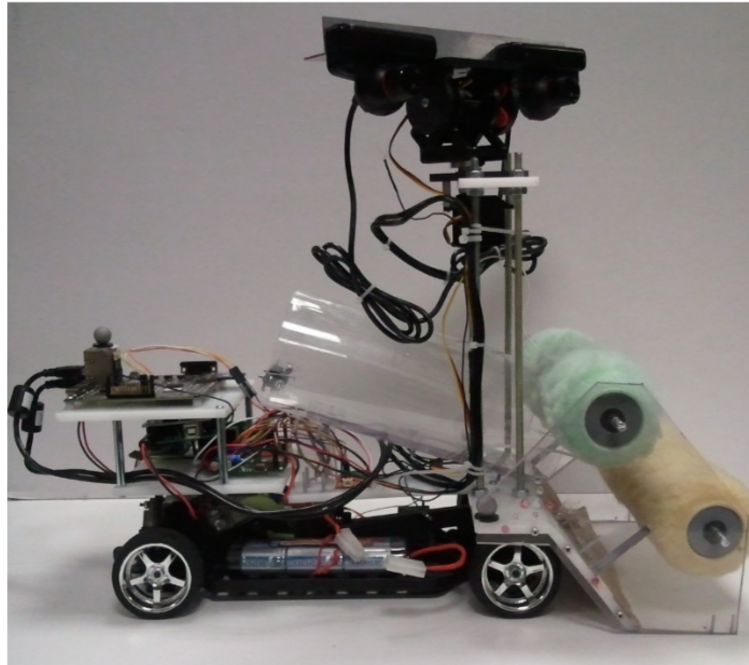


Figure 3.11: Autonomous robotic tennis ball boy

3.3.2 Golf ball picking robot

Yun, Moon, Ko [20] from South Korea developed mobile robot for picking up golf balls at golf driving range (Figure 3.12) . It uses wide view camera mounted on building for getting approximate location of balls, and then computes directions for mobile robot. Robot has GPS and inertial unit and is using MCL to localise itself at the gold range. It then pick ups balls with stereo camera on board into the body of the robot. Several key points: needs supporting infrastructure, does not regard obstacles on golf course, primitive ball detection, does not discriminate between golf balls and other objects.

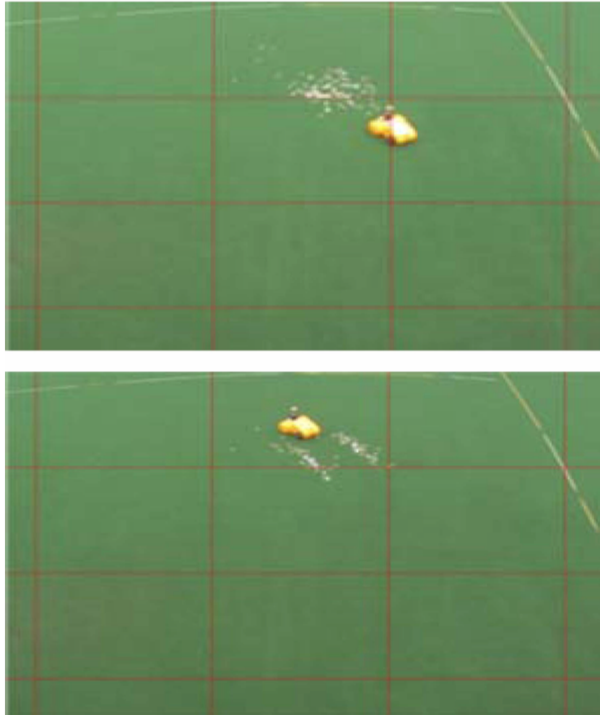


Figure 3.12: Golf ball picking robot from its wide view camera

3.3.3 Autonomous Table Tennis Ball Collecting Robot

In 2017, Yeon et al. [21] developed mobile table tennis robot for picking balls (Figure 3.13). It has camera, lidar and ultrasound sensors, does not use other infrastructure like previous robot, in section 3.3.2. It actively navigates environment to avoid obstacles and can differentiate between table tennis balls and other similarly shaped objects. It uses vacuum cleaner suction mechanism to collect balls and can manipulate nozzle to collect balls from tight spots.

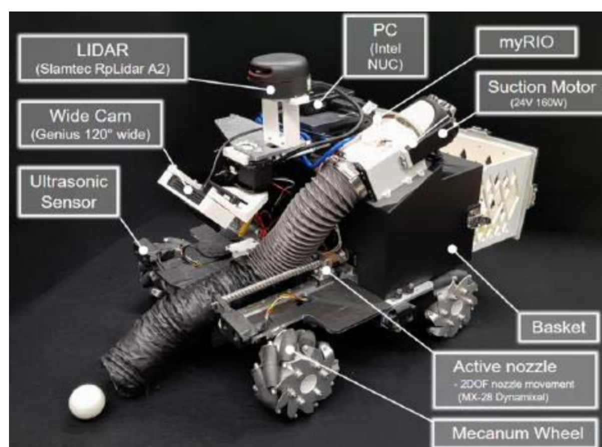


Figure 3.13: Table Tennis Ball Collecting Robot

3.3.4 Badminton

For badminton, only few manual solutions exist. First one is *ProSort CC-60*[12], shown in Figure 3.14, developed by students at Cambridge University[22]. It puts shuttles on strings moving upwards and lets gravity to orient shuttlecock head downward. It then drops them into the tubes under the string mechanism. Energy for carrying shuttlecocks on strings takes from wheels.

Figure 3.14: ProSort CC-60 manual picking mechanism.

Similar manual solutions, such as *Shuttlecock Collector Machine(SCM)*[13], made by students of Politeknik Kuching Sarawak of Malaysia, show in Figure 3.15 and *Shuttlecock Collector / Ballsammler*[14] made by Helmut Siemen of Germany, shown in Figure 3.16.

Figure 3.15: Shuttlecock Collector Machine

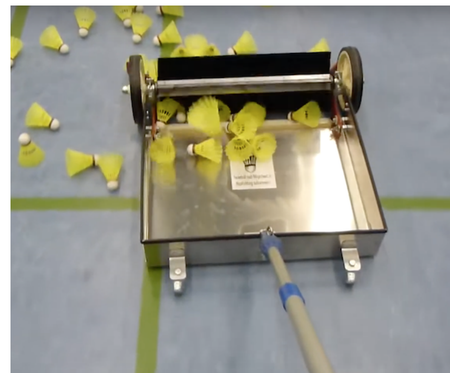


Figure 3.16: Shuttlecock Collector / Ballsammler

We were not able to find any existing autonomous robotic solution that pick shuttles up and arranges them into suitable format for coaches and players to train.

[12]https://www.youtube.com/watch?v=cxVtv0ZTztI
[13]https://www.youtube.com/watch?v=X5Di1ocYQfY
[14]https://www.youtube.com/watch?v=4h0DzsYsZ7A

4. Analysis

In this chapter we analyse goals that we mentioned in introduction. We also discuss what approach should we take in programming the software, and mention what hardware is available for construction of the robot.

In the section 4.1, *Application architecture*, we discuss pros and cons of using robotic middleware. In the following section *Hardware* we mention available hardware for our robot. In the last section 4.3, *Software*, we analyse possible software solutions for goals we listed in introduction.

4.1 Application architecture

Our application can be programmed by different approaches. In this section we describe two types of application architectures, we consider their advantages and disadvantages and describe which one is better suited for our problem.

4.1.1 Monolithic application

Monolithic application is an application performing all the tasks itself. It has these advantages:

- It is fast, because there is no overhead between passing data around such as with messages that need to carry additional data, such as header, time stamp etc.

But it also has disadvantages, such as:

- It is hard to debug, since everything is coupled tighter than modular design.
- It is hard to modify and maintain.
- Most importantly, we would have to reinvent the wheel by programming already available solutions.

4.1.2 Using ROS

ROS is an open source[1] robotic middleware based on distributed computing using interconnected nodes. [23].

Using ROS has several advantages:

- We can use message passing middleware to interconnect components.
- We can use already created packages, such as packages for controlling Turtlebot.
- We can use visualising software RViz for visualising output from cameras.
- We can use packages for creating map of environment.

[1] https://github.com/ros/ros

- Sate of the art (SOTA) algorithms are available as ROS packages.

But it also has disadvantages, such as:

- Since packages are made by different authors, modifying already existing code can be difficult because authors have different coding style, also lack of tutorials for some packages/libraries.
- It is not easy to set up.
- Software has to be in form of packages.
- Passing data by messages can have processing overhead, such as serialisation/deserialisation. Message passing also takes some time, so real-time applications can be affected.

We decided to **use ROS**, mainly because robotic base we have available, Kobuki platform, has ROS package built, and we can leverage already existing packages such as mapping, and deep learning, which we will mention in section 4.3

Our goal will be thus to understand third party packages, and develop control software in form of ROS package, that will:

- recognize shuttlecocks,
- translate them from 2D image space into 3D map positions,
- use knowledge of shuttlecock positions in map to generate a path that robot will take,
- send commands to the Kobuki base that will result in picking up shuttlecocks.

There could be **noise** in visual sensors and also in ability of neural networks to detect shuttlecocks. Additionally, there can be imprecision of shuttlecock positions with respect to the generated map. Another source of noise can be motors of the robot base, that can cause imperfection in the robot base movement. We will have to incorporate the assumption of uncertainty when creating the control software.

4.2 Hardware

We are aiming to use control system on real robot, thus we need to consider which hardware is suitable for out needs.

There are many embedded computers such as Arduino, Raspberry Pi, Nvidia Jetsons, etc.

We need computer that will:

- be small to be installed on top of a Kobuki robot
- be power efficient, so we can run it with external battery and for prolonged time
- be sufficiently powerful to seamlessly run ROS,
- be powerful enough to run visual processing such as neural network segmentation and recognition, visual mapping from stereo cameras

We decided to use Nvidia Jetson platform because of better graphic performance compared to the other options.

At first we developed on Jetson Nano(Figure 4.1), but it proved not powerful enough run simultaneously object recognition by deep neural networks, mapping and other nodes, we acquired Jetson Xavier NX (Figure 4.2), which according to benchmarks[24] is *10x more powerful* in deep learning applications than Nano (Figure 4.3), and empirically suits our needs.

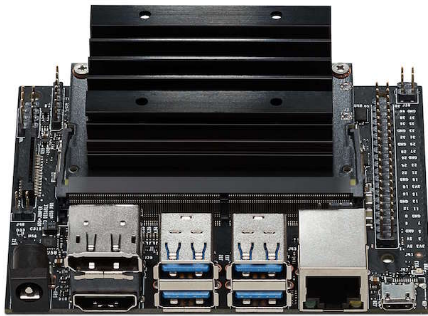


Figure 4.1: Nvidia Jetson Nano

Figure 4.2: Nvidia Jetson Xavier NX

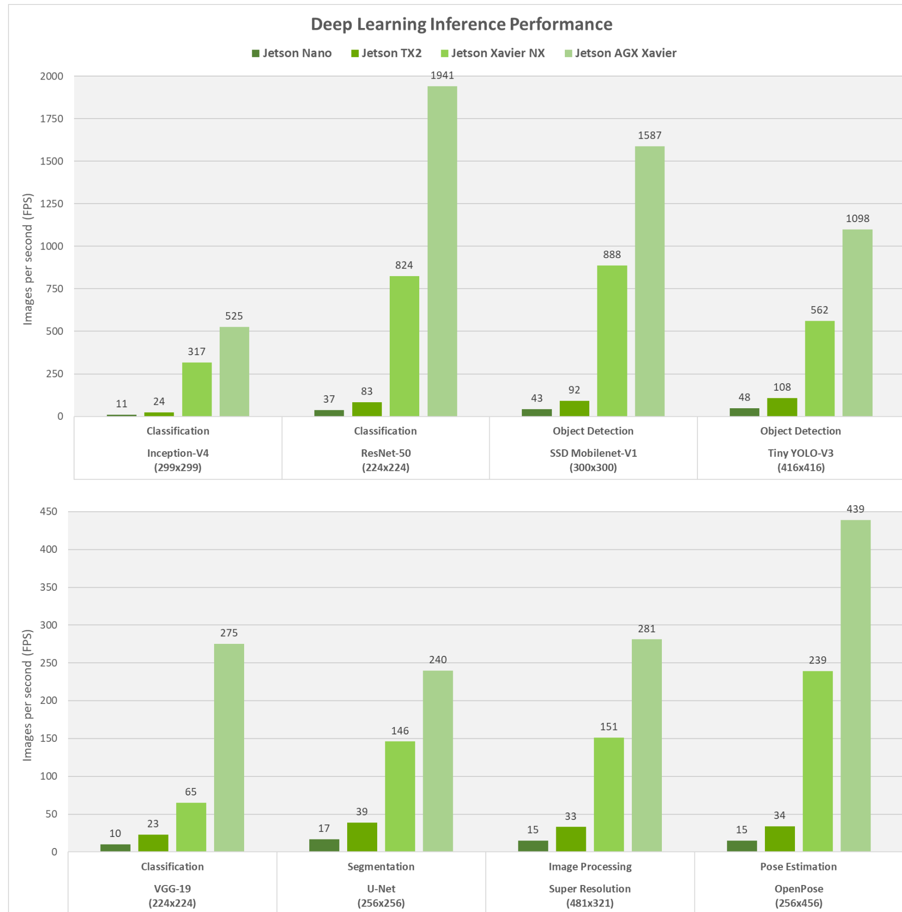


Figure 4.3: Performance comparison of Nvidia Jetson computers, from[24]

We summarised relevant data about Jetson kits [25], from:

Name	Nano B	Xavier NX	AGX Xavier
AI perf.[1,2]	472 GFLOPS	21 TOPS	32 TOPS
GPU	128-core NVIDIA Maxwell GPU	384 Volta CUDA cores and 48 Tensor cores	512-core NVIDIA Volta GPU with 64 Tensor Cores
CPU	Quad-core ARM® Cortex ®-A57 MPCore processor	6-core NVIDIA Carmel ARM®v8.2 64-bit CPU 6 MB L2 + 4 MB L3	8-core NVIDIA Carmel Arm ®v8.2 64-bit CPU 8MB L2 + 4MB L3
Memory	4GB 64-bit LPDDR4 25.6GB/s	8GB 128-bit LPDDR4x 51.2GB/s	32GB 256-bit LPDDR4x 136.5GB/s
Power cons.	10W	15W	30W
Price	99$	399$	699$

[1] GFLOPS = giga floating point operations per second
[2] TOPS = tera operations per second

4.2.1 Sensors

We have available several depth cameras: Stereo Labs ZED[26] camera (Figure 4.4), Intel Realsense D455[27] (Figure 4.5), Astra Orbecc[28] camera (Figure 4.6) and Intel Realsense D455 [27] (Figure 4.5). All three cameras supports Ubuntu 18.04, and have wrappers for ROS Melodic, and their properties are summarised in Table 4.1.

Figure 4.4: ZED stereo camera.

Figure 4.5: Intel Realsense D455

Figure 4.6: Astra camera

Name	ZED 1	D455	Astra
Manufacturer	Stereo Labs	Intel	Orbecc
Year	2017	2020	2017
Shutter[1]	Rolling	Global	Rolling
Illumination	None	Laser	Laser
RGB resolution	2208x1242 @15fps	1280 × 720 @30fps	640 x 480 @30fps
Depth resolution	same as RGB	1280 × 720 @ 90fps	640 x 480 @30fps
FOV[2]	90° x 60°	87° × 58°	60° x 49.5°
Range	0.5 m - 25 m	0.6 m - 6 m	0.6m – 8m
Accuracy	< 2% up to 3m	< 2% at 4m	+/- 3mm @1 m
Power cons.	2W	3W	< 2.4 W
Price	449$ [3]	419$	149$

[1]Rolling shutter reads out pixel values sequentially by rows or columns, resulting in characteristic blur during fast camera motion. Global shutter reads all pixel values at once and does not suffer from this type of blur.

[2]Field of view, Horizontal x Vertical in degrees.

[3]discontinued

Table 4.1: Properties of depth cameras.

4.2.2 Robotic platform

The robotic base we have available, Kobuki (Figure 4.7), is part of the Turtlebot 2 ROS project. It is useful because we can use it in simulation in Gazebo. Turtlebot is long running project of OpenRobotics foundation and ROS community.

Figure 4.7: Kobuki robotic base

4.2.3 Other

Kobuki has innternal battery, but for prolonged use we used 40 000mAh Viking powerbank [29] for notebooks (Figure 4.8) that has suitable adapter for Jetson

Xavier.

Figure 4.8: Viking external battery

4.3 Software

In this section we will analyse what software components we need to successfully implement:

- control system,
- mapping - localisation,
- vision - recognition,
- planning,
- movement,
- visualisation
- user interface,
- picking.

4.3.1 Control system

Simplest control system for robots are *reactive* agents such as Braitenberg's vehicle [30] (Figure 4.9) or line follower robot (Figure 4.10), where inputs are mapped directly or tightly to outputs.

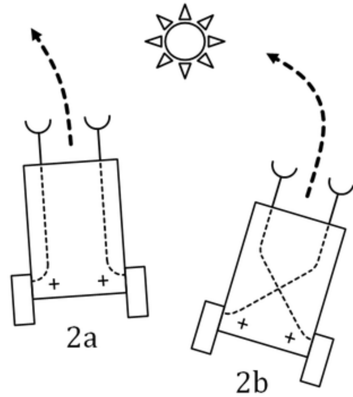


Figure 4.9: Braitenberg vehicle.

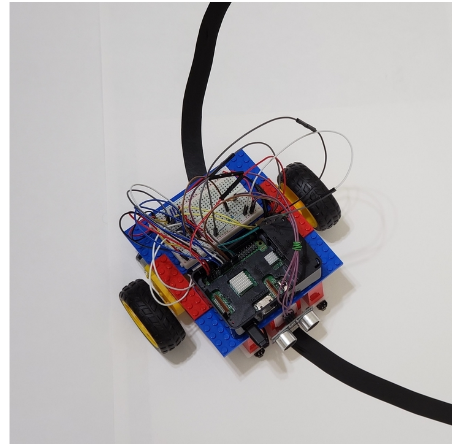


Figure 4.10: Line follower.

More complicated control paradigm is *SPA (Sense, Plane, Act)* implemented in robots such as Shakey [31], developed in 1960s, where robot has some internal representation of the world, and can use it to generate more intelligent actions by reasoning about the world. Problem with robots like Shakey (Figure 4.11) was that they were slow, and did not respond to dynamic changes in environment. After plan was generated, it was carried out without direct feedback from sensors.

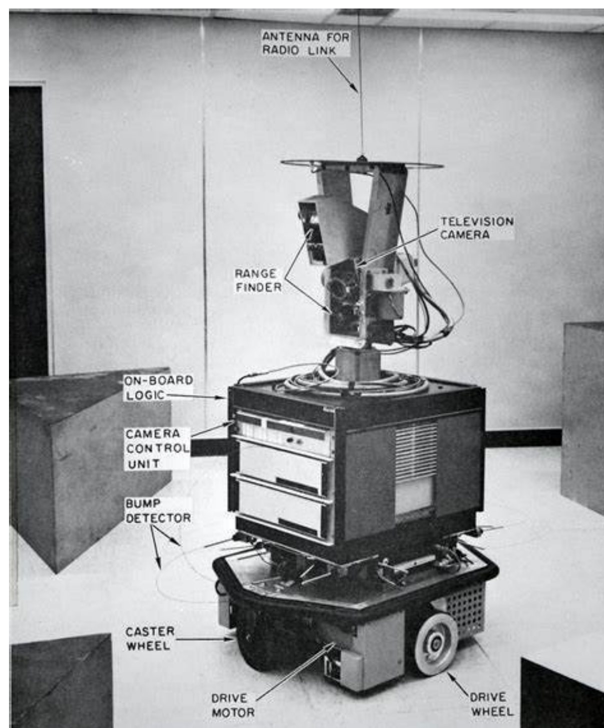


Figure 4.11: Shakey the robot.

Possible data flow from sensors, through planning to actions is shown in Figure 4.12.

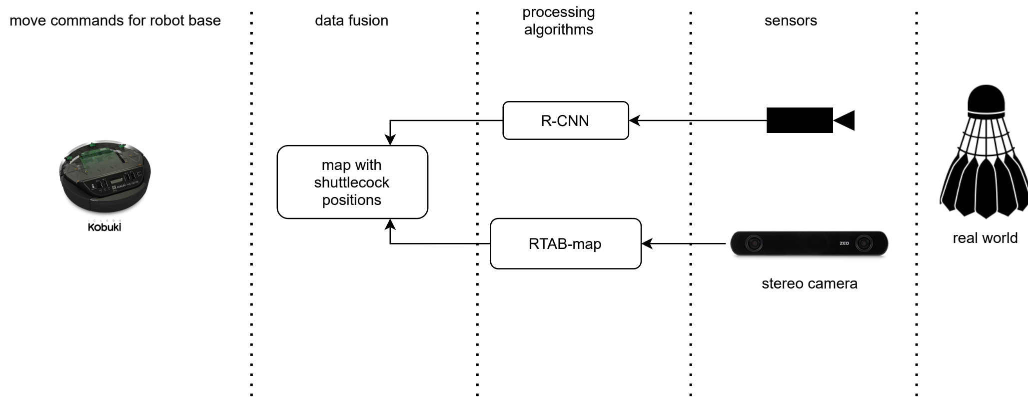


Figure 4.12: Example of data flow of possible robot in a SPA paradigm.

Next progression of robotic control systems was *Subsumption architecture* created by Rodney Brooks[32] in 1986. It was composed of progressively complex control programs (*behaviours*) on top of each other. Higher level behaviour could override lower level behaviour. For example, zeroth level would be collision evasion, first level wandering, and second exploration.

However, control based on behaviors hit its ceiling, because it proved hard to create long lasting goals that were difficult to optimize[33].

Next followed architectures that combined reactivity and planning called *layered* or *hybrid* architectures, such as Firbys[34] *three layered* architecture (Figure 4.13) that was divided into *planning*, *executive*, *behavioral* layers.

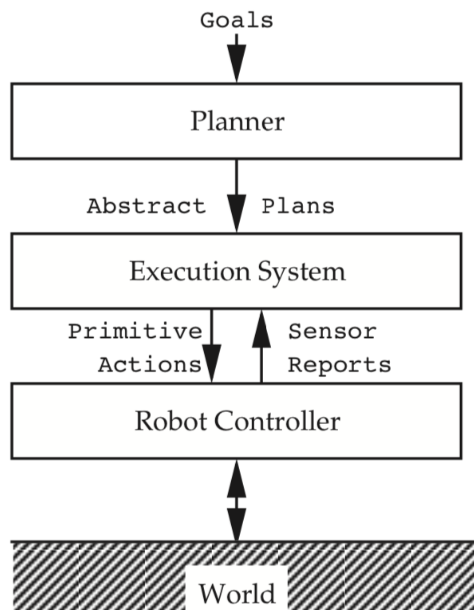


Figure 4.13: Three layered architecture according to Firby, from[34].

4.3.2 Shuttle recognition

We need to detect position of shuttles, so our robot can approach them and pick them up. We will be using visual inputs from camera.

Since shuttles can be scattered across court almost everywhere, we need reliable and fast method to **detect** them.

Then we also need to distinguish object of our interest, shuttles, from background-floor. Our robot could also incorporate human detection to not hit any players that could be in its vicinity.

Since 2000s there has been interest in using neural networks for vision processing, such as classification of numbers by LaCunn[35]. In the 2010s there have beed numerous advances in computer vision using neural networks, such as R-CNN[36] (Figure 4.14).

Figure 4.14: Overview of R-CNN architecture.

Modern state of the art neural networks such as YOLOv5 [37] are composed of many layers, combining convolutional and dense layers (Figure 4.15).

Figure 4.15: Overview of architecture of YOLOv5 neural network, from [37].

4.3.3 Mapping

Badminton courts are flat surfaces with marked lines. We could have two approaches to solving shuttle localization problem. Firstly, we could have up and running mapping at all times, and detect shuttles from point clouds created by mapping algorithm. This is very energy and computationally inefficient, and after the robot picks up shuttle the algorithm need to update place where the shuttle has been.

Secondly, we could run mapping algorithm to create map of the court *without* any shuttles present, and then just mark positions of shuttles as points in map. This of course has few problems, for example we need to keep track of which shuttles are which as to not mark them in our map *more than once*, and we have lot of similar frames from the camera. We could also assume that nobody will shuffle shuttle positions behind robots back since that would complicate matters.

4.3.4 Planning

Another step we need to consider is planning of robot motion. Assume we can give robot coordinates (x, y) where to move, relative to some frame, for example robot's map. In this way, if we had list of coordinates of shuttles $[(x_1, y_1), (x_2, y_2), ..., (x_k, y_k)]$ we could use some TSP solver like integer programming to get shortest path through these coordinates. Of course this does not consider time constraints such as *turning* of the robot base. Another problem is that we have only partial knowledge of the world, i.e. where the shuttles are located because we look at the world from the view of robot (Figure 4.16). This could be alleviated if we had camera at the roof looking at the court below, but this is undesirable because we want to have compact robot that would be usable at many different places without any difficult and time consuming installation of cameras.

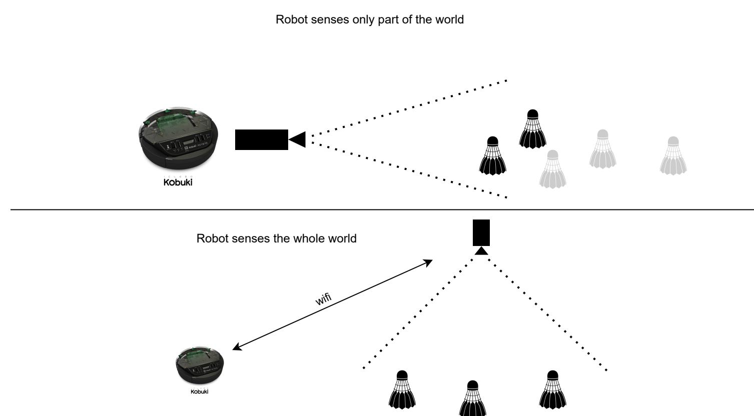


Figure 4.16: Partial vs whole view of the world.

4.3.5 Visualisation

We need to visualise *output* from the sensors of the robot. Mainly for debugging and development, but also during working of the robot. We need to

see *images* from camera, *map* the robot creates, *objects* - shuttlecocks the robot recognizes from environment, *path* or *plan* the robot generates to pick up the shuttles. We would also like to visualize state of the robot, or it's progress.

4.3.6 User interface

We would like control robot during the mapping, for example by keyboard or joystick. During working, use command line to start robot, and modify behavior of robot interactively with mouse or keyboard.

5. Proposed solution

In this chapter we describe how we designed control system, what parts it is composed of, what packages we used. Because our software is implemented as ROS nodes within ROS framework, we will also describe additional files such as configuration, model, launch files, URDF files and world files.

In the section 5.1, *ROS*, we introduce essential ROS concepts, in the section 5.2, *Gazebo* we describe Gazebo simulator and its components. In the following sections we describe goals from introduction, each goal is implemented in as one or more ROS nodes.

5.1 ROS

ROS is an open source robotic middleware used for speeding up robotic development. It provides many useful features as shown in Figure 5.1.

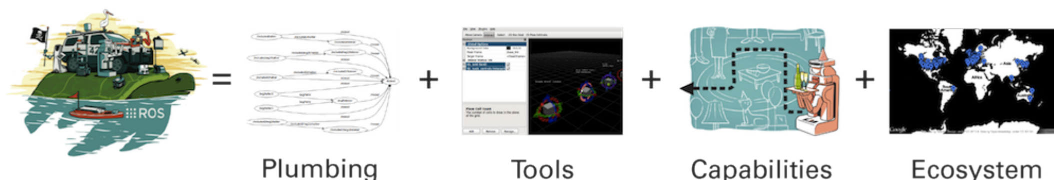


Figure 5.1: ROS equation

Main advantage of using robotic middleware such as ROS is that software is split into interconnected **nodes** that communicate with *messages* over *topics*. It is advantageous because we can swap parts such as camera for another one, the only requirement is that data is still being published on the same topic with same message type. Each node is run in separate process, therefore this approach is inherently using parallelism/concurrency[1]. Nodes are included in packages which are built using **catkin** tool which uses cmake. Nodes use publish subscribe paradigm or service/reply paradigm. **Messages** are defined in the .msg format which are then converted to the Python or C++ classes.

We will use **RViz** 3D rendering program to visualize what robot sees, map, goals, path etc. Nodes can be run from terminal as normal program, or can be run by writing *launchfile* for convenience. Launchfiles can include parameters and themselves can be included in other launchfiles. Last part is **Gazebo**, which we will describe in next section.

[1]In literature, parallelism usually means computation on multiple cores, while concurrency means computation using multiple *threads* running on one core. For example, Jetson Xavier has 6 ARM cores, and since we are running programs as separate nodes, we are using available resources that we get from multi-core CPU.

5.2 Gazebo

Gazebo [38] is a robotic simulator with physics engine (ODE) with 3D rendering capability, now independent of ROS. Robot inputs from simulated environment, such as cameras, odometry, lasers are supplied by writing C++ plugins inside Gazebo, which publishes corresponding messages over topics that we can use for programming the robot. If published topics and simulated physics are somewhat similar to real sensors and real physics, we can use same or slightly modified software for both simulated and real robots.

Environments in Gazebo are called **worlds**[2] and are specified by writing XML files in SDF format.

Because we need to test algorithms before we apply them to a real robot, we will use simulated robot in a simulated environment. For this we will use Gazebo simulator [39], which has ROS integration via **gazebo_ros** [3] package. Gazebo is designed to be separate from ROS, and can be controlled programmaticaly using plugins[4]. Plugins are also used to generate sensor data for robots, and can be used to control the world[5].

5.2.1 Preparing the simulation

We designed few worlds that will serve as a test environment for our simulation. We downloaded free models such as badminton court, shuttlecock and bench from the internet and modified them in Blender to decrease vertex size because our computing platform is very limited. We will describe this more in section 6.6, *Gazebo simulation*. Robots are usually spawned in from launchfile and their models are not included in SDF.

5.3 Control system

For controlling behaviour of the robot, we could use state machine, which are similar to finite state machines [40],[41]. Possible state machine is shown in Figure 5.2.

[2] http://gazebosim.org/tutorials?tut=build_world&cat=build_world
[3] http://gazebosim.org/tutorials?tut=ros_overview&cat=connect_ros
[4] http://gazebosim.org/tutorials/?tut=plugins_hello_world
[5] http://gazebosim.org/tutorials?tut=plugins_world_properties&cat=write_plugin

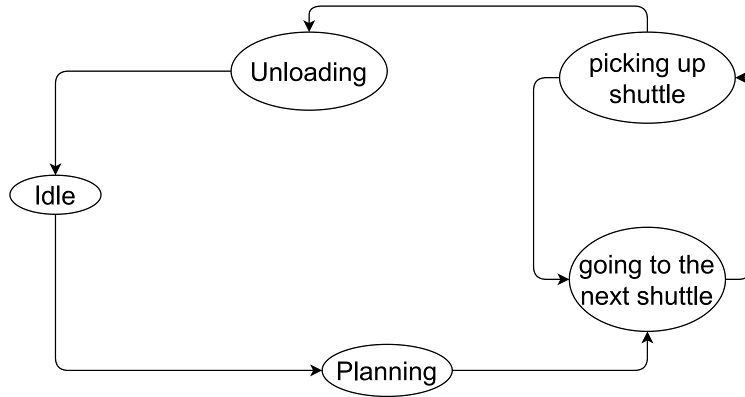


Figure 5.2: Example of possible state machine of the robot.

We could design more complicated state machines, for example: "If you picked 10 shuttles, go to location of unloading".

5.4 Mapping and localisation

For the robot localisation and mapping we chose open source ROS package RTAB-Map[42]. It is a graph-based SLAM approach[43]. It supports inputs from stereo and RGB-D cameras (Figure 5.3) and outputs pose and occupancy grid map which we can use in navigation. To create map, we first need to drive robot around environment to get images from which RTAB-Map constructs graph nodes used in localisation. Images are compared using SIFT or SURF algorithms for matching features. Algorithm also takes input odometry from the robot or can use RGBD visual odometry from camera. If the match between images is found, RTAB-Map creates link - loop closure (Figure 5.4). During the mapping, many loop closures can be found and algorithm tries to minimise error with respect to the measurements - images and their extracted features. In Figure 5.5 is shown setup of RTAB-Map node on a robot. RTAB-Map is widely used by ROS community [42] for mapping and localization with RGBD cameras, and its performance was evaluated experimentally [44]. We found its performance suitable for our solution.

	Inputs							Online outputs			
	Camera				Lidar				Occupancy		Point
	Stereo	RGB-D	Multi	IMU	2D	3D	Odom	Pose	2D	3D	Cloud
GMapping					✓		✓	✓	✓		
TinySLAM					✓		✓	✓	✓		
Hector SLAM					✓			✓	✓		
ETHZASL-ICP					✓	✓	✓	✓	✓		Dense
Karto SLAM					✓		✓	✓	✓		
Lago SLAM					✓		✓	✓	✓		
Cartographer					✓	✓	✓	✓	✓		Dense
BLAM						✓		✓			Dense
SegMatch						✓					Dense
VINS-Mono				✓				✓			
ORB-SLAM2	✓	✓									
S-PTAM	✓							✓			Sparse
DVO-SLAM		✓						✓			
RGBiD-SLAM		✓									
MCPTAM	✓		✓					✓			Sparse
RGBDSLAMv2		✓					✓	✓		✓	Dense
RTAB-Map	✓	✓	✓		✓	✓	✓	✓	✓	✓	Dense

Figure 5.3: Survey of ROS compatibile SLAM packages, table from [42].

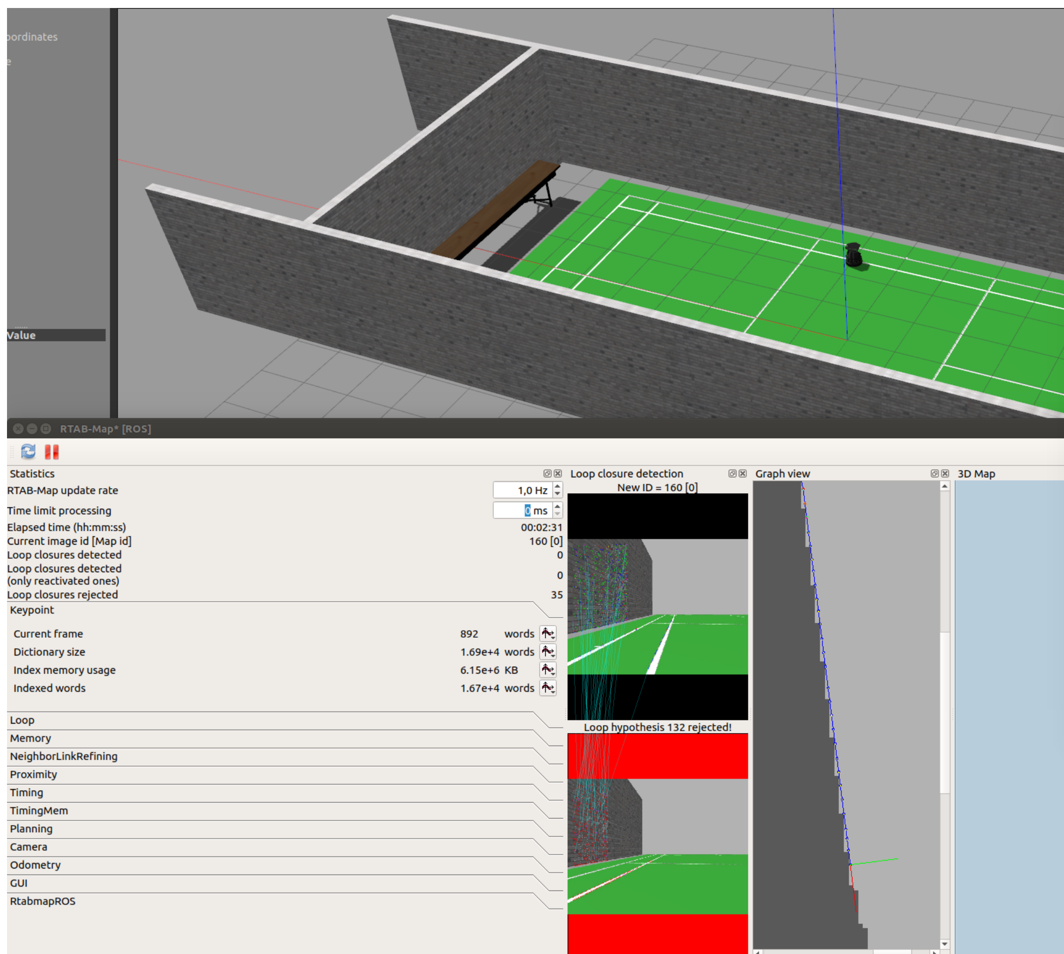


Figure 5.4: Loop closure detection in RTAB-Map viewer

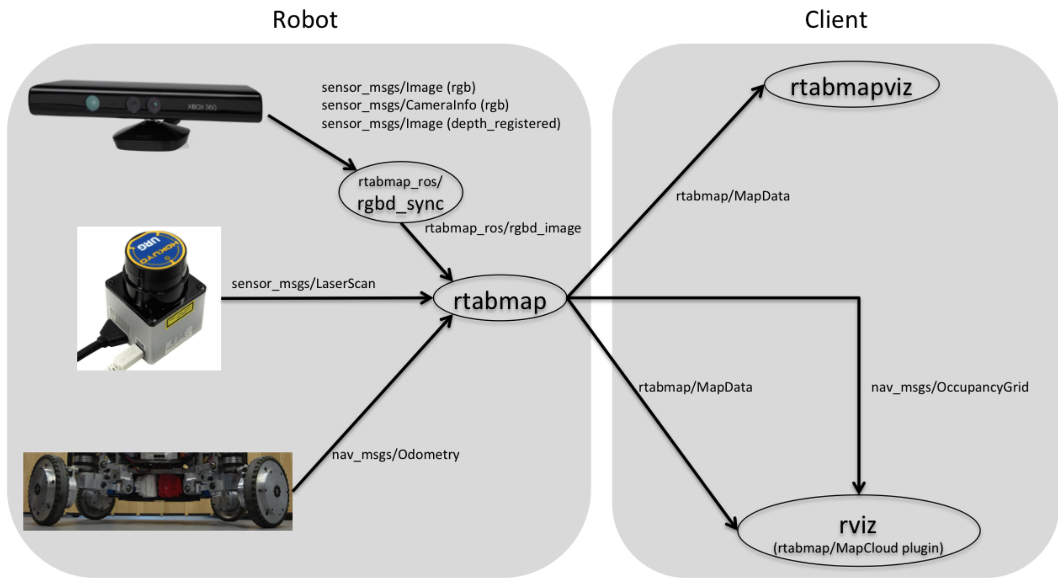


Figure 5.5: Setup of RTAB-Map node on a robot, from [45]

5.5 Navigation and Planning

If we knew position of shuttlecocks in advance, we could plan optimal path between n shuttlecock. This would be equivalent to Travelling salesman problem. Because field of view of camera is limited and we are viewing world from low position instead of birds eye view, we can't plan in advance optimal path of the robot on court. We can only estimate position of shuttlecocks we see in front of the robot. Therefore if we see a shuttlecock, we estimate its position relative to robot and send command to planning node to generate path to it. ROS has already available planning package *move_base* which we will use and which can use data from RTAB-Map mapping package (Figure 5.6).

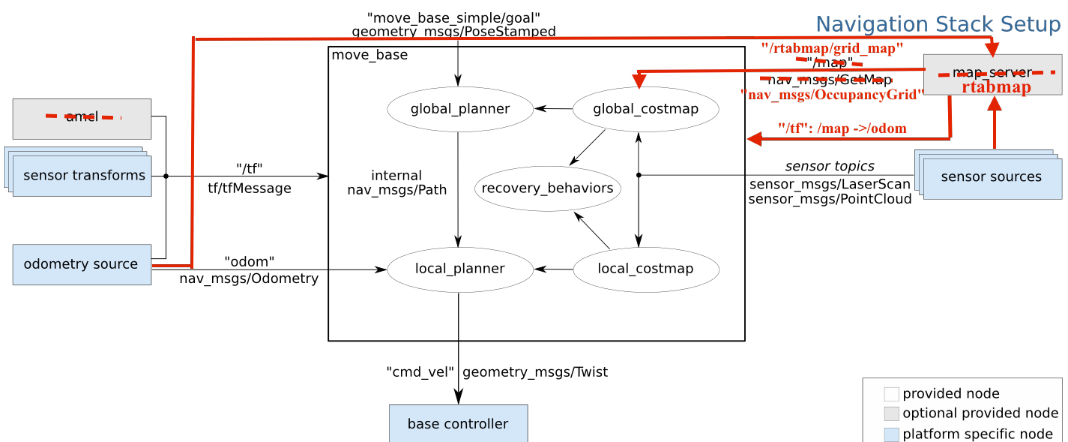


Figure 5.6: Navstack with RTAB-Map.

Global planner (Figure 5.7) plans path from A to B using *Dijkstra algorithm*. This is done on squared grid called occupancy grid. In ROS nav_msgs/occupancy grid has three values, -1, 0 and 100. Unknown space is represented as -1, 0 is free space and 100 is occupied space.

While planning algorithms such as *Dijkstra* works in other domains such as computer games to create path from A to B, in real world robot would need to spin it's motors to follow the path. This is accomplished by *local planner*, in ROS Navstack this is *dynamic window approach* which simulates few trajectories using forward and angular velocities. Output of local planner are velocity commands, which are fed to move_base package that is abstraction of a robot.

Other thing which navstack keeps care of, is that if we have small robot, let's say 10 cm in diameter, and larger robot such as our Kobuki, which has diameter of 40 cm, the paths cannot be the same because smaller robot could pass closer to the wall and between obstacles than Kobuki could. This is solved by using costmaps with *inflation* (Figure 5.8) which inflates larger area around obstacles by robot's diameter. Programmers can also use costmaps to modify the behaviour of planning by altering the cost of cells. We will use this functionality to mark areas of map that robot can use or should avoid, such as not going to another court, more in subsection 6.8.1.

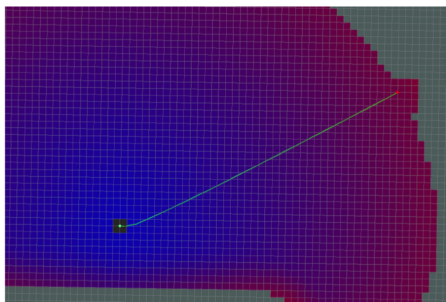


Figure 5.7: Path planning by global planner from package move_base, from [46].

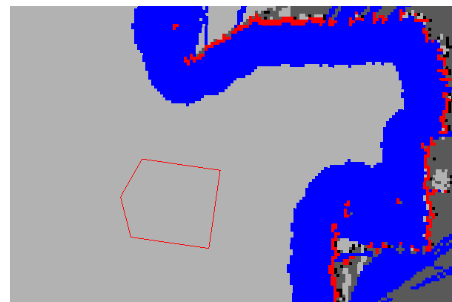


Figure 5.8: Inflation of obstacles (red dots) from Occupancy grid, from [47].

5.5.1 Mapping

If we want to give robot movement position commands such as *go to position (x,y)*, we need a map. Map for given environment can be created with ROS package RTAB-Map. This needs to be done manually before autonomous robot driving on court. After the map is created, we turn off mapping, map will be saved for later use to a database. We then can use this map for localization.

5.6 Movement and shuttle picking

Our robotic base Kobuki has two motors. We could send movement signals directly to the motors, from the output of the camera. This approach is called *reactive* (mentioned in subsection 4.3.1), and does not use map of the environment. In this case it could happen that robot could run off to neighbouring court because he saw shuttlecock there, and since he has no notion of map, nothing would prevent it from doing so.

The other approach is that robot has map of environment and he generates position of the shuttle respective to the map from the camera. This position is then transformed to goal for move_base package. This package generates plan

consisting of velocity commands for robot wheels. This is advantageous because robot can detect obstacles and modify plan so it doesn't hit anything. It has disadvantage that it could identify shuttlecocks as obstacles, therefore avoiding them.

5.7 Computer vision

We need to recognise shuttlecocks from visual input and represent them in a way that robot can generate movement commands to pick them up. We should be able to recognize shuttlecock on different ground colors (Figure 5.9). Our input is simulated RGB and depth cameras in Gazebo simulation. In real life we got depth information from stereo camera that is produced by merging data from left and right cameras. In ROS this information is produced by Gazebo plugin and in real world by camera drivers by publishing *sensor_msgs::Image*[6] for image data and *sensor_msgs::PointCloud2*[7] for depth information, called **point clouds**. Point cloud is an array of n-dimensional points, usually 3 or 6 dimensional such as (x, y, z) for position or (x, y, z, r, g, b) with added colour information[8]. Fortunately, camera drivers output point clouds in *organised* point cloud[9] format, meaning points from depth camera are organised as 2D matrix row-major order in the array, and can be accessed by (x,y) indexing.

[6] http://docs.ros.org/en/api/sensor_msgs/html/msg/Image.html

[7] http://docs.ros.org/en/api/sensor_msgs/html/msg/PointCloud2.html

[8] http://pointclouds.org/documentation/structpcl_1_1_point_x_y_z_r_g_b.html

[9] https://pcl.readthedocs.io/projects/tutorials/en/latest/basic_structures.html

(a) Blue court

(b) Orange court

(c) Green court

(d) Wooden court

Figure 5.9: Example of different court colors and materials.

5.7.1 Object recognition

For shuttle recognition we will use library for neural network inference, developed by Nvidia, jetson-inference[10]. They also developed ROS node for this library [11].

Input for the detection are *sensor_msgs/Image* messages. Neural network outputs detected objects as *vision_msgs/Detection2DArray*[12].

5.7.2 Training neural network

Training consists of feeding data of the form (x_{train}, x_{target}) to the training algorithm and tweaking weights of the neural network by backpropagation. We have several choices for creating datasets. We could use already created network and hope that it generalizes to the new objects, but this usually does not work at all. Another option is to use pretrained network for similar purposes, and then *retrain* it with more examples, this time with shuttlecock images and rectangles by hand.
Third option is to create dataset *synthetically* [48], i.e. in some modelling program, if done correctly, can be huge benefit to training algorithm (Figure 5.10). It is because we could in theory generate large amounts of training data. The

[10] https://github.com/dusty-nv/jetson-inference/

[11] https://github.com/dusty-nv/ros_deep_learning

[12] http://docs.ros.org/en/melodic/api/vision_msgs/html/msg/Detection2DArray.html

problem with this is that we would have to generate training data as closely resembling the real world as possible, in various instances that could arise in the real world.

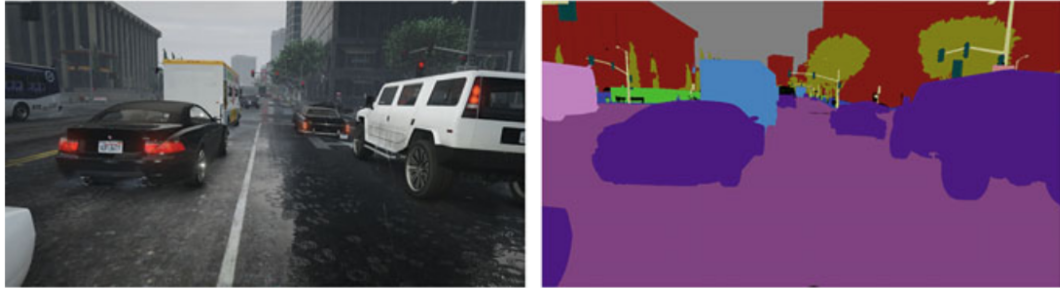


Figure 5.10: Example of synthetic dataset, from [48].

Since generating synthetic datasets is computationally very intensive, we will create smaller datasets by hand i.e. several hundred photos of shuttlecocks and their true positions.

5.7.3 Position estimation

We get bounding box of shuttlecock from neural network node. This is information about position of shuttlecock in 2D space, and we will merge this information with depth points in 3D space, acquired from depth camera. Since depth points are organised into 2D matrix format, we can take rectangle of points corresponding to the shuttlecock from the point cloud message.

5.8 Visualization

5.8.1 RViz

RViz (ROS Visualization) is a 3D visualization tool for displaying data from topics in ROS. It is composed of windows letting us display all necessary messages that robot gets and uses. It can visualise 3D data such as point clouds, 2D image data from camera and image processing nodes, and also display navigation information. It also allows us to visualize data for localisation such as map, robot model, local plan, global plan, and markers for shuttlecock positions.

Specifing robot in URDF

URDF[13],[14] is an XML format for specifying models of robots. We need it for both simulation and using robot in real world. Especially we need to set relationships between robot parts, such as wheels, chassis, camera, or other sensors and actuators. This is essential since we need to know position of data with respect to some origin, such as sensor or centre of robotic base, and we need to establish relationship between parts to easily transform between coordinate frames. This

[13] http://wiki.ros.org/urdf
[14] http://gazebosim.org/tutorials?tut=ros_urdf

is done by specifying links and joints in URDF. For example, if the camera sensor is in front of the robot, distance to objects would be different than to centre of the robot, or its actuators.

Fortunately, there are pre-build URDFs for Kobuki and ZED camera, the only thing we need to specify is relationship between Kobuki and ZED camera. We measured this using meter and found out that ZED origin is offset by 0.15 m in x-axis and 0.15 m in z-axis. This can be set in ZED launch file[15] which passes parameter to xacro macro.

5.9 Shuttlecock picking

Shuttle picking could be done by arm (Figure 5.11) or by **rotary mechanism** (Figure 5.12) as manual solutions mentioned in chapter 3. Robotic arms, such as popular (Figure 5.11) shown in low cost are slow, and are also costly and quite heavy on small robotic platforms such as Kobuki we are using. They also require additional power. They would also increase complexity of the system. This is because they would need precise manipulation (so that it does not damage shuttlecock) based either on visual servoing of the end effector, or planning with software packages such as MoveIt. This would increase complexity of system drastically and would also take it longer to pick shuttles. On the other hand rotary brushes are relatively simple to maintain, does not require any additional planning or AI methods. Biggest advantage of brushing system is that that **we don't need to know position of shuttlecock exactly**, because brush is wide, we can have some *room for error* and thus make whole system more reliable, even if we estimate position of shuttlecock with some error, as long as it is within width of the brush, we can still pick it up. After considering all above mentioned points, we chose simpler solution based on rotating brushes.

Figure 5.11: PincherX 100 Robot Arm by Trossen robotics

Figure 5.12: Example of brushing mechanism

[15] https://github.com/stereolabs/zed-ros-wrapper/blob/master/zed_wrapper/launch/zed.launch

5.10 User interface

Because the robot is autonomous, user interface is using commands in command line. Another option is to monitor robot outputs in RVIZ and optionally set commands and goals through clicking on map or other UI elements.

6. Implementation

In this chapter we describe parts of proposed solution that we developed ourselves, that is visual processing, visualisation for RViz, control system, models needed to run simulation and other files, such as training data for neural network and .stl files for 3D printer, and how to assemble them for working picking mechanism, and programs used for manipulation of allowed and detection areas in RViz.

6.1 Launchfiles

Launchfiles are XML files in ROS ecosystem used for running nodes, or recursively other launchfiles. After we type following command into terminal:

```
roslaunch shuttlebot_control gazebo_all.launch
```

Roslaunch command finds *gazebo_all.launch* roslaunch file inside package *shuttlebot_control*, and runs in order launchfiles *shuttlebot_gazebo.launch*, *dl_gazebo.launch* and nodes *image_processing* and *point_draw.py*.

```
<launch>
    <include file="$(find
     ↪  shuttlebot_control)/launch/shuttlebot_gazebo.launch" >
    </include>

    <include file="$(find shuttlebot_control)/launch/dl_gazebo.launch"
     ↪  >
    </include>

    <node pkg="shuttle_distance_estimation" name="image_processing"
     type="image_processing"/>
    <node pkg="shuttlebot_control" name="point_draw"
     ↪  type="point_draw.py"/>
</launch>
```

6.2 Neural network

Neural networks need data for training. In this section we describe process of acquiring data from camera mounted on moving robot, cleaning and annotating of data, and lastly deployment of trained model using ROS node within docker container.

6.2.1 Data acquisition

Data acquisition was done on badminton courts with camera mounted on robot with position that would be similar on a robot during deployment. We acquired some data with ZED camera (Figure 6.1), and rest of the data with Intel D455(Figure 6.2) because it has better image quality, and also has *global*

shutter, which means that fewer images are blurred while the robot is moving. Both cameras have 1280×720 resolution. After we recorded data from camera, we used free and open source photography program Darktable[1] to remove duplicated and other low quality images. First dozens of images (recorded with ZED camera) were created by putting shuttles at various positions by hand with static robot, but after looking at images, we came to conclusion it would be more natural if the robot was moving and shuttles would fly and drop around robot, similarly at what would happen during training with players and trainer. Thus, we let the robot move and threw shuttles it its field of view, while we had running scrip that would capture and save images every few seconds to disk. After a dozen of passes robot around court we stopped robot, and analysed and annotated pictures. This process was repeated on multiple days with hundreds of photos recorded each time. We also used different courts, to capture more variety for background for training. As for court color, we recorded around 70% of photos of blue court, and rest on the green court to show that neural network trained on one court colors can detect shuttlecocks independently of court color, but that it is not hard to add more data of different courts, and we can still utilize neural network we trained before to help us with new data annotation.

Figure 6.1: Image from ZED camera.

Figure 6.2: Image from D455 camera.

6.2.2 Dataset creation

We acquired data for neural network training by taking images of shuttlecocks from multiple angles and under different lightning conditions. We created simple shell script that would take pictures at determined intervals and drove robot around. Another method that we employed and can be used successfully is to record *rosbag* and extract images from it. We gathered data over multiple days, on various blue and green courts on different times of day. Although courts are usually indoor, there could be (and are) reflections from sunlight from windows that are not present at other times (night). Since we want to get rectangle of where shuttle is in the image, we have to provide a rectangle ourselves with which we train the network, as shown in Figure 6.3.

For annotation of data we firstly used simple tool provided with jetson-inference[2], but we quickly found out that it is too limiting for annotation of larger datasets.

[1] https://www.darktable.org/

[2] https://github.com/dusty-nv/jetson-inference/blob/master/docs/pytorch-collect-detection.md

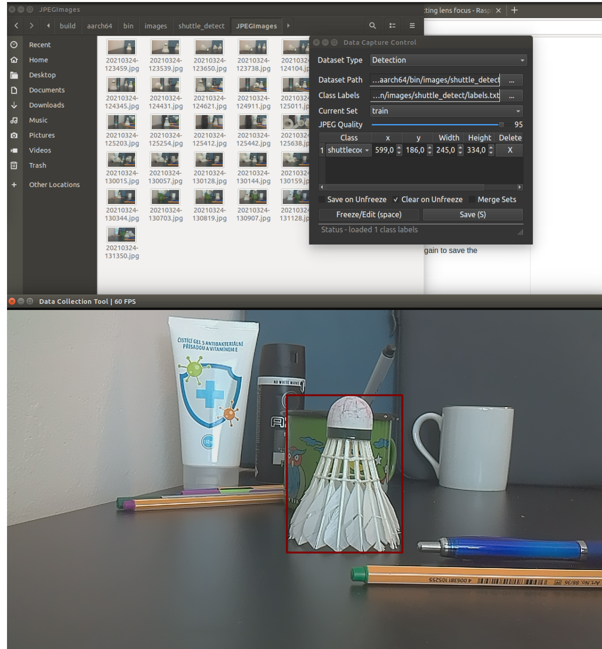


Figure 6.3: Creating dataset manually for object detection by simple annotation tool.

More sophisticated tool that we used later on for annotating, is free and open sourced *CVAT*-Computer Vision Annotation Tool developed by Intel [49]. CVAT is a web based tool, but publicly available online version limits usage to 500mb of data and 10 tasks[3], which we eventually exceeded. Therefore we installed CVAT locally on our machine (Figure 6.4). Another useful feature of CVAT is that we can use various models for aiding of dataset creation. We used MIL tracker, SIAM mask and lastly, we used our own neural network trained on shuttlecocks to detect shuttlecocks. Although CVAT is able to export annotated datasets in various formats, we needed to create a script that would transform exported dataset from similar format to a specific format that neural network training programs accepts. In figure Figure 6.5 we shows unnanotated picture with multiple shuttlecocks, figure Figure 6.6 shows annotated picture.

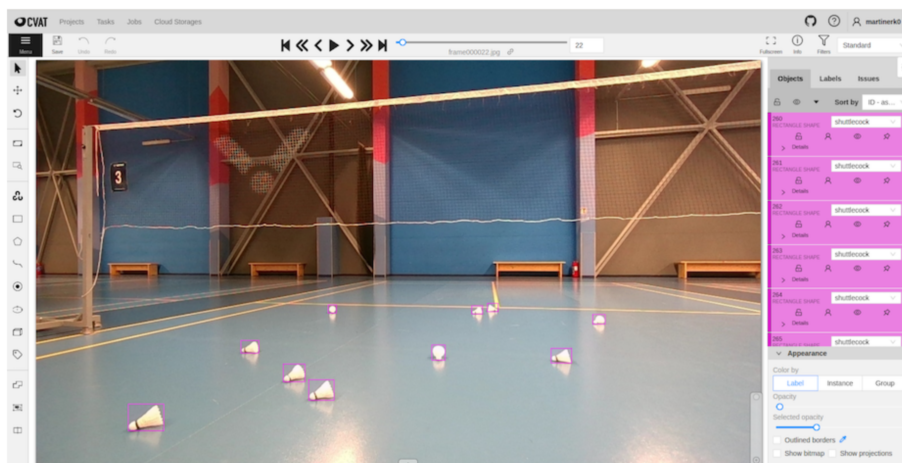


Figure 6.4: Self-hosted version of CVAT.

[3]Task is a set of images, we structured tasks to correspond to different days and courts.

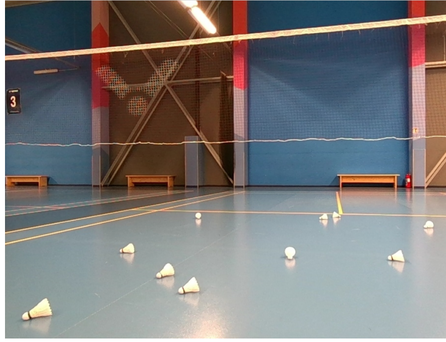
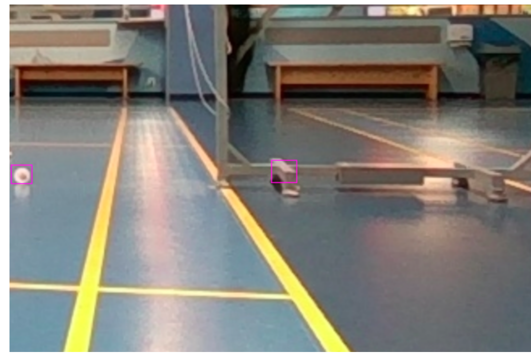


Figure 6.5: Picture taken from camera on real court, not yet annotated.

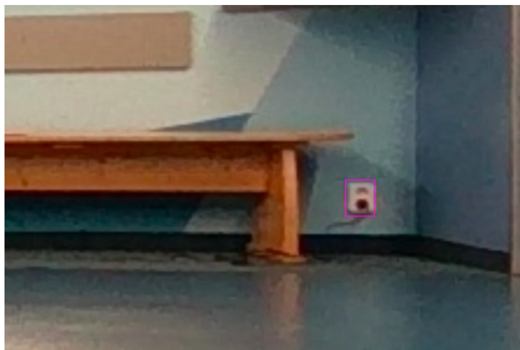
Figure 6.6: Same picture, manually annotated in CVAT.

After training first NN model on images, we deployed and run it on robot. We quickly noticed that model detected shapes that looks like shuttlecock, but are not, such as white circular reflections (Figure 6.7a), badminton net holder (Figure 6.7b), white electrical socket (Figure 6.7c) or author's white socks (Figure 6.7d). After gathering more data with these *false positive* examples, we trained neural network again, this time adding these edge cases to the training set.

(a) Reflections of lights (right)



(b) Net holder.



(c) White power socket.

(d) Bright white socks.

Figure 6.7: Examples of false positives in CVAT after autolabelling with neural network trained on first few hundreds of images.

6.2.3 Deployment on robot

We are using Ubuntu 18.04 with ROS Melodic Morenia which has packages built with Python version 2.7. Because we want to use state of the art libraries for training and inference of neural networks which run on Python 3, we have two options. We could either try to port all the other packages that we developed and used before to Python 3 and use ROS Noetic (which natively uses Python 3), or use Python 2.7 as same as before and isolate neural network with Python 3. We chose latter variant, because lot of packages were not yet available for ROS Noetic during development of the robot, such as packages for Kobuki, SMACH, and others.

One type of *"isolation"* we chose, is to use Docker containers. This is relatively streamlined approach, because all the code we developed and tested before stays intact running natively on machine, and new code that needs Python 3 and other dependencies is neatly isolated in a container. This way, we can run our ROS Node with Python 3 inside container, with all the dependencies it needs. Nvidia Jetson *allows using underlying CUDA cores* from the docker containers with NVIDIA Container Runtime on Jetson[4].

Docker containers are build from **Dockerfiles**, which are text documents describing insctructions to build an **image** that is run as *container*. Following is excerpt from dockerfile we used to build image with ROS Noetic, Pytorch, YOLOv5, our ROS node and other necessary libraries:

```
ARG BASE_IMAGE=nvcr.io/nvidia/l4t-pytorch:r32.7.1-pth1.10-py3
 FROM ${BASE_IMAGE}
 ARG ROS_PKG=ros_base
ENV ROS_DISTRO=noetic
ENV ROS_ROOT=/opt/ros/${ROS_DISTRO}
ENV ROS_PYTHON_VERSION=3
 ENV DEBIAN_FRONTEND=noninteractive
 WORKDIR /workspace
```

6.3 Visual processing

Visual processing is implemented by *image_processing* node in image_processing.cpp. It has three subscribers on:

- */detectnet/detections* topic, which listens to messages of type vision_msgs::Detection2DArray
- */camera/rgb/image_raw* which listens to messages of type *sensor_msgs* :: *Image*
- */camera/depth/points* which listens to messages of type *sensor_msgs* :: *PointCloud2*

[4]It is installed by default on Jetson devices. https://github.com/NVIDIA/nvidia-docker/wiki/NVIDIA-Container-Runtime-on-Jetson

Normally, every subscriber has its own callback function, but since we want to combine them, we use Synchronizer from *message_filters*[5] package. Since messages have different time arrivals, we combined them into one callback function using message filter Time Synchronizer[6] with *ApproximateTime*[7] policy.

Since these are two independent messages and are likely to have different timestamps, we will use time synchronizer to combine them into one callback. We than cut off points that are far behind shuttlecock (Figure 6.8), and compute average of the points using centroid[8] method of the pcl[9] library that we are using for manipulating with point clouds.

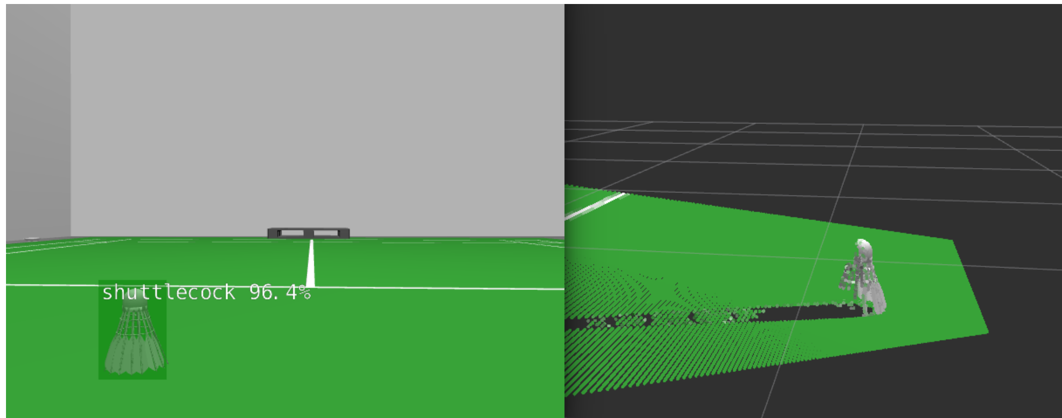


Figure 6.8: Neural network bounding box vs. point cloud. Some points inside bounding box are far behind shuttlecock.

This gives us relatively accurate estimate of shuttlecock's position, shown in Figure 6.10, compared with Figure 6.9. The point we got is in optical frame of the camera, we only need to set header of the point to this frame, and ROS tf system will compute the position in map frame for us. We can then set z value to 0, as to project it to the ground.

Figure 6.9: Shuttlecock in front of robot, inside Gazebo.

Figure 6.10: Estimated position of shuttlecock, from RViz

[5] http://wiki.ros.org/message_filters

[6] https://docs.ros.org/en/api/message_filters/html/c++/classmessage__filters_1_1TimeSynchronizer.html

[7] http://wiki.ros.org/message_filters/ApproximateTime

[8] https://pointclouds.org/documentation/classpcl_1_1_centroid_point.html

[9] https://pointclouds.org/

Positions of multiple shuttlecocks estimated on a real court (Figure 6.11).

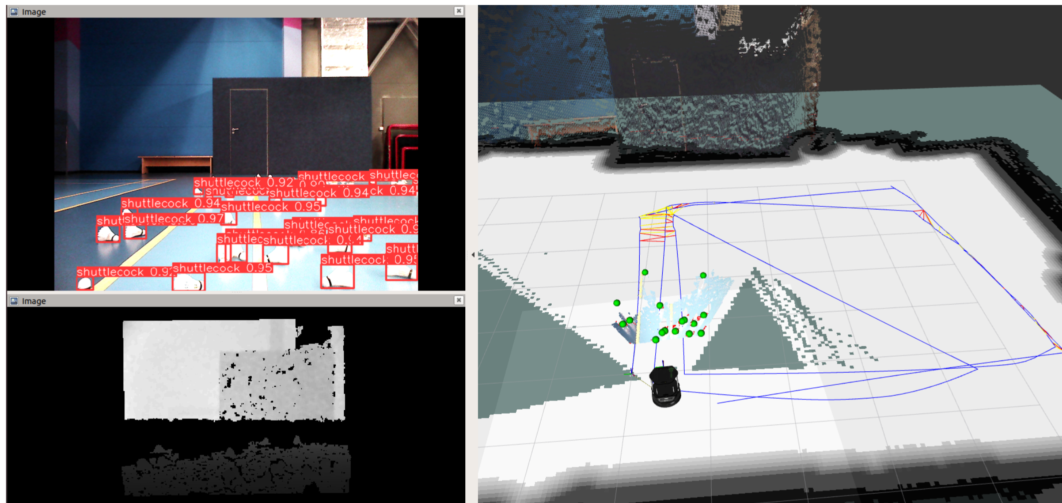


Figure 6.11: Multiple detected shuttlecocks on real court (top left) with estimated positions (right).

6.4 Visualisation

Shuttlecock position visualisation is done by *point_draw* ROS node in point_draw.py file.

```
class Visualize_node:
    def __init__(self):
        self.node = rospy.init_node('point_draw', anonymous=True)
        self.sub =
         ↪  rospy.Subscriber('/marker',Point,self.visualize_point)
        self.pub = rospy.Publisher('shuttlebot_points', MarkerArray,
         ↪  queue_size=10)
        rospy.spin()

    def visualize_point(self,point):
        marker_array = MarkerArray()
        marker_array.markers.append(create_rviz_marker(point))
        self.pub.publish(marker_array)
```

Where *create_rviz_marker(point)* method takes Point and outputs array of Markers[10] which are RViz visualisation objects.

6.5 Control system

Our control system is composed as a state machine, using of SMACH[11], a library for task-level execution and coordination in ROS.

[10] http://docs.ros.org/en/api/visualization_msgs/html/msg/Marker.html
[11] http://wiki.ros.org/smach

SMACH state is a Python class. We can specify inputs and outputs of a state. Transition between states is done by implementing *execute* method. In comparison to finite state machines from Automata theory, SMACH states are not fixed description of the world, but can do any computation inside them [50].

For picking one shuttle, we can design following state machine. It consists of states IDLE[12] (Shown in green in Figure 6.12) and COLLECTING and outcomes[13] *failed_picking* and *picked* (Shown in red in Figure 6.12).

At start, state machine is in state IDLE, and waits for messages from vision system. When it gets message about detected shuttlecock, consisting of (x,y) position in map frame, it passes this information to the next state COLLECTING using transition *got_msg*.

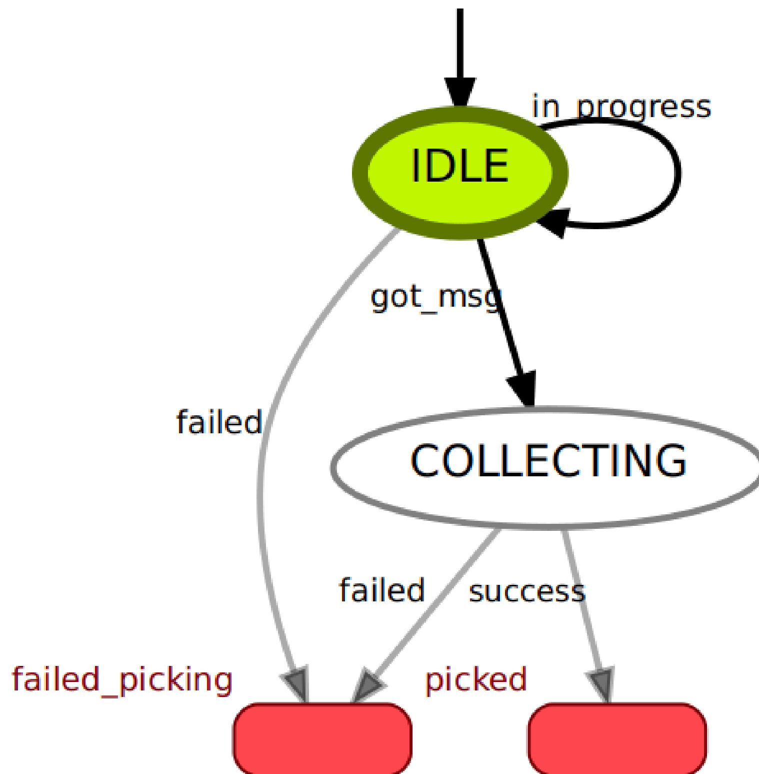


Figure 6.12: System is in first state, IDLE.

In the state COLLECTING (Shown in green in Figure 6.13), (x,y) coordinate of shuttlecock is transformed into the move_base[14] action and system waits for the result. If the robot is successful and picks the shuttle up by moving the robot base, state machine uses transition *success* and goes to the *picked* outcome.

[12] According to SMACH convention, states are named with uppercase.

[13] In SMACH, state machines can be nested, outcomes can serve as transition from sub state machine to higher level machine.

[14] http://wiki.ros.org/move_base

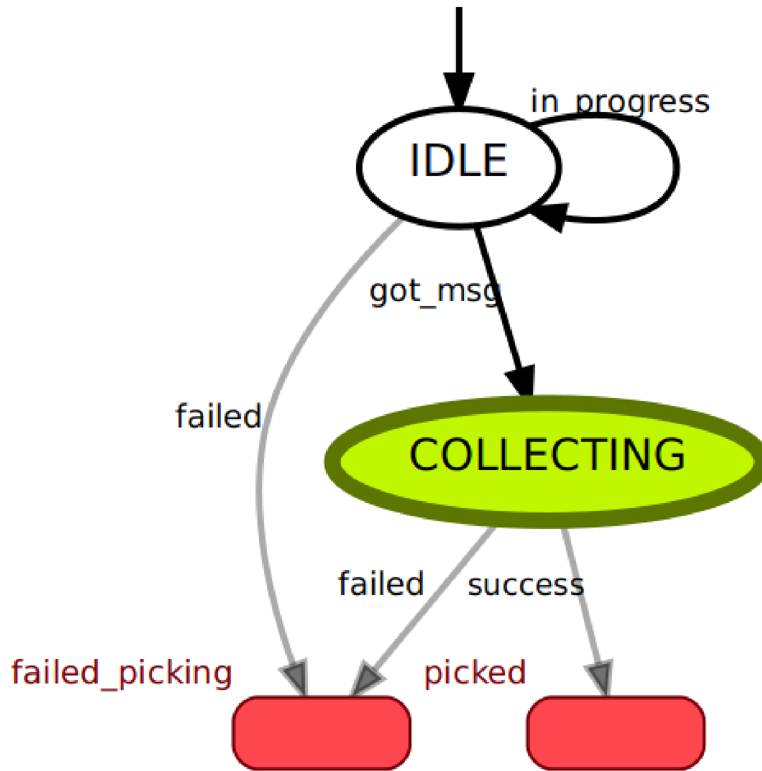


Figure 6.13: System is in second state, COLLECTING.

6.5.1 Concurrent container in SMACH

In SMACH, we can simulate concurrent run of few states using concurrent container. We can use this to check for detections while keeping track of navigation. Of course this is only formalism of this particular implementation of state machines, and there are *more* processes running in background such as navigation, neural networks, visual processing, allowed area node, detection area node etc. We could also create our own large state encompassing all the functionality of the few states that we want to run concurrently, but we think it is better to use multiple small states in concurrent container that one large state that could be run without it.

In this way, we can listen for commands in one state, while communicating with ROS nodes in other state. Of course, we don't want and need all functionality to be inside SMACH states. Nodes such as RTAB-Map, YOLO, rosserial, and Kobuki nodes will be running at all times outside of the state machines. Ephemeral nodes, such as blinking LEDs on Kobuki or playing sound can happen inside SM states.

6.5.2 Control System

Shuttlecock picking by state machines in SMACH can be done as shown in (Figure 6.14). Robot starts at IDLE state, and when it gets command to work, it enters concurrency state called SM_PATROL. In this concurrency state are three states SM_STOP which listens for user commands, SM_DETECT which

listens for shuttlecock detections, and SM_NAV which keeps track of waypoint navigation. If SM_DETECT gets message from visual processing with estimated point of shuttlecock, it terminates with outcome 'got_msg' and passes this point to the COLLECTING state which starts brushes of picking mechanism and sends goal with passed position to move_base node. After move_base report that goal has been reached, it turns off brushes and goes back to concurrency SM_PATROL state which resumes waypoint navigation and listneing for detections in SM_DETECT state. After we collect given amount of shuttlecocks, we use state SM_STOP to go to HOME_POSITION which again sends goal to move_base and upon reaching waits for emptying by human by going to state IDLE. This process repeats until all the shuttlecocks are collected or we turn off the control system.

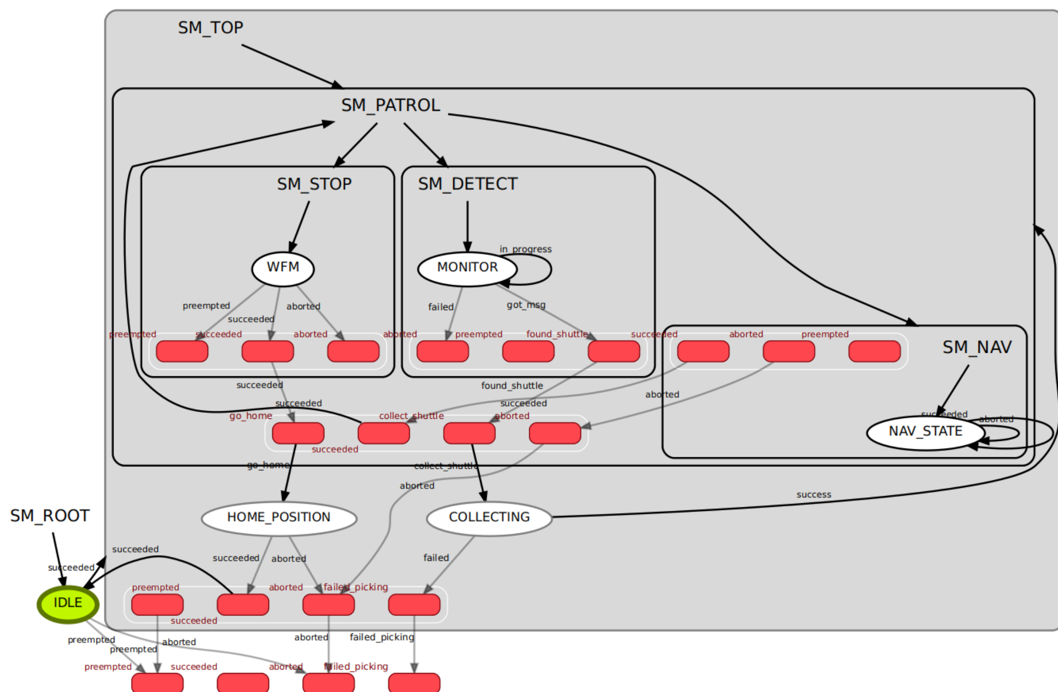


Figure 6.14: State machine of shuttlecock picking.

6.6 Gazebo simulation

For development and testing of our solution, we created multiple worlds for our simulation in the Gazebo simulator (Figure 6.15). Simulation consists of building a world populated with models with visual and physical attributes such as mesh, mass, inertia, etc. Gazebo uses physics engines such as ODE[51] to simulate physics, and OGRE[52] engine to draw 3D graphics. We already mentioned parts of gazebo in section 5.2. *World* is a XML file in SDF [53] format. It is specified by world tag. Inside it we can place *models* which are also files in SDF format.

```
<sdf version='1.6'>
  <world name='default'>
  <!-- populated with  models -->
```

```
  </world>
</sdf>
```

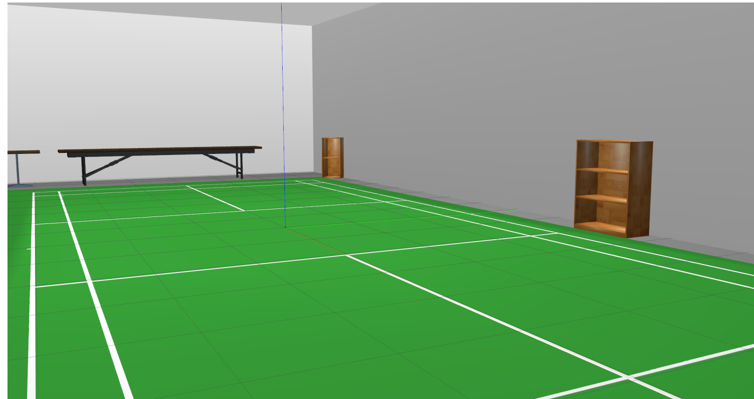


Figure 6.15: Gazebo world.

Models are also specified in SDF format. Necessary parts for Gazebo simulator models are physical properties such as inertia, mass, staticness.

Mesh is the actual 3D data of the model, i.e. information about vertices, edges, faces, textures etc. It is in Collada .dae [54] format.

6.6.1 Gazebo Plugins

Gazebo plugins[15] are shared libraries written in C++ used to control parts of simulation. There are 6 types of plugins, we will use *world plugin* that attaches to the world, and *sensor plugin* which attaches to link of a model and produces data that can be further used in simulation.

6.6.2 Sensor plugin for picking shuttlecocks

To simplify picking shuttles in simulation, as mentioned in *section 5.9*, we created a sensor plugin for Gazebo. It's function is, when robot touches the shuttle, we delete it from simulation and count it as picked up, for simplicity.

We do this by creating *contact plugin*[16] for contact sensor[17]. Gazebo is using similar message system as ROS. Messages are published on topics, and are using *publish/subscribe* paradigm. Contacts are published on *shuttle_contact* topic.

This is done by detecting collisions between models.

One thing we need to be wary of is that robot should not go into some weird state when it touches shuttle. i.e. shuttle needs to disappear before it triggers any Kobuki's collision avoidance system (for example by touching bumpers). To do this, we added collision cylinder (Figure 6.16) to Kobuki's model by modifying original URDF.

[15] http://gazebosim.org/tutorials?tut=plugins_hello_world&cat=write_plugin

[16] http://gazebosim.org/tutorials?tut=contact_sensor

[17] http://osrf-distributions.s3.amazonaws.com/gazebo/api/9.0.0/classgazebo_1_1sensors_1_1ContactSensor.html

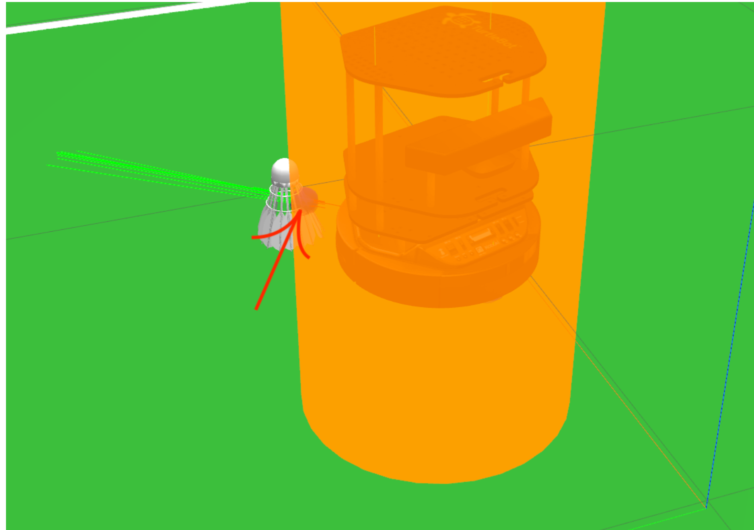


Figure 6.16: Contacts (pointed by arrow) detected between shuttle model and collision element (in orange).

We then attached this cylinder to base link of the robot.

```
<joint name="bounding_joint" type="fixed">
    <parent link="base_link"/>
    <child link="cylinder_link"/>
    <origin xyz="0.00 0.0 0.0" rpy="0 0 0"/>
    <axis xyz="0 0 0"/>
</joint>
```

Because we want to use this bounding cylinder to detect collisions but be *"contact-free"* to remove shuttle before it touches the robot, we use *collide_without_contact* property of *contact* tag. We care only about collisions of cylinder with shuttlecocks, so we set same *collide_bitmask*[18] on both models[19].

```
<contact>
    <collide_without_contact>1</collide_without_contact>
    <collide_bitmask>0xf000</collide_bitmask>
</contact>
```

6.7 Picking system

In this section we describe development of picking mechanism used on real robot and what materials and parts we used during development. Fully working version which we will describe in this section is shown in Figure 6.18.

[18] http://gazebosim.org/tutorials?tut=collide_bitmask&cat=physics

[19] Links to be precise

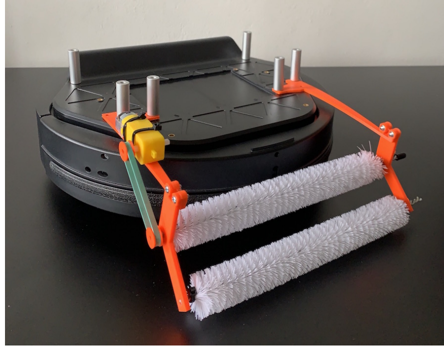


Figure 6.17: Early version of picking mechanism.

Figure 6.18: Finished, working prototype of picking mechanism.

Our first version, (Figure 6.17), was using only 3D printed parts which proved too weak, especially parts extending forward from robot and holding the weight of brushes. Since 3D printing large parts is time consuming, there is also limit to strength of parts in particular dimensions and also space limit imposed by 3D printing bed of 3D printer. We therefore used L aluminium profile for holding other, 3D printed parts. Aluminium is strong, compared to PLA[20] plastic used for 3D printing, and is not prone to bending. We used the L profile as *scaffold* to support 3D printed parts holding brushes, which we had to design from scratch. We used Kobuki hardware drawings[55], shown in Figure 6.19, to drill holes into L profiles and attached them to Kobuki frame, (Figure 6.20).

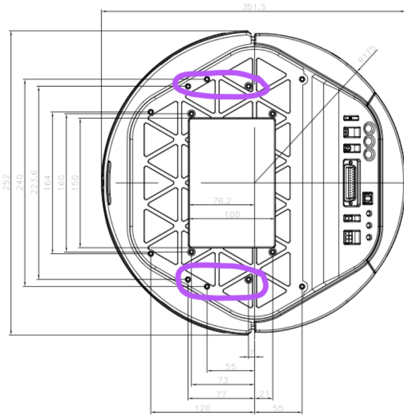


Figure 6.19: Kobuki drawing, with marked holes used for attaching L profiles, image from [55].

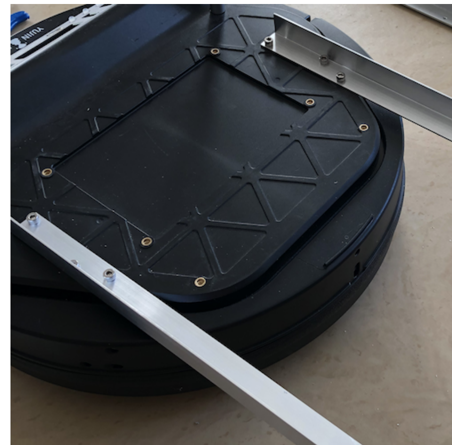


Figure 6.20: Aluminium L profiles with drilled holes, attached to Kobuki robot.

Rotating brushes for our picking system are commonly used for cleaning radiators at homes[21]. We chose these because they are somewhat available at common hardware stores, they are cheap, and we could try multiple versions. We settled for *small diameter* brush with *harder bristles* for lower brushes and brushes with *softer bristles* for *upper brushes*. Brush availability at hardware stores proved to be essential, because from 20+ brushes we examined, only 2 were of sufficient

[20] https://en.wikipedia.org/wiki/Polylactic_acid

[21] https://www.obi.cz/cistici-pristroje/kartac-na-radiatory-dlouhy/p/2718716

quality, i.e were not bent and had fairly straight steel core that can be used as axle for rotating by motors.

We adapted brushes in following way. Firstly, we cut steel core to suitable length with steel cutting wheel[22], then we cut off bristles with scissors. Next, we sanded off remaining hairs with sanding drum[23] and then polished it with sanding disc[24], resulting axle shown in Figure 6.21. Lastly, we applied thin layer of hot glue so that axle would fit the ball bearing, (Figure 6.22).

Figure 6.21: Cut and sanded steel core of the brush.

Figure 6.22: Brush after applying hot glue to the core.

6.7.1 3D printed parts

In second version of picking mechanism we used aluminium L profiles instead of 3D printing. This allows us to create smaller modular 3D printed parts that can be affixed to L profile. Repository for all 3D printed parts we created is at: https://github.com/martinerk0/shuttlebot_3dprints. We created parts that can be **moved** along the L profile, allowing us to experiment with various brush configurations and to fine tune picking mechanism. We call them *L slider*, shown in Figure 6.23 and Figure 6.24.

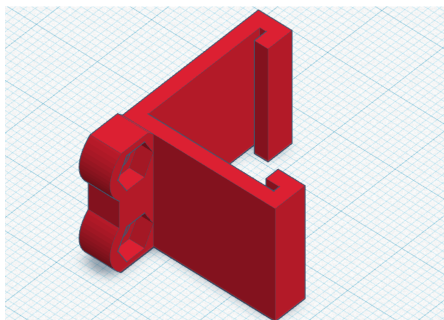


Figure 6.23: Slim L slider, used connected to bottom double brush holder.

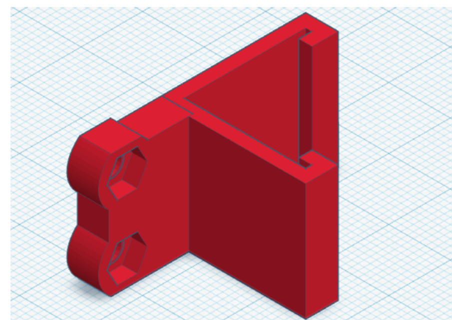


Figure 6.24: Longer L slider, connected to two upper holders.

[22]https://www.dremel.com/us/en/p/456-01-26150456aa
[23]https://www.dremel.com/us/en/p/407-2615000407
[24]https://www.dremel.com/us/en/p/412-2615000412

Next 3D printed parts are *holders* - part that holds brush through the bearing to the L slider with M3 nuts and bolts. We created four variants. First two holds one large upper brush each, shown in Figure 6.25 and Figure 6.26.

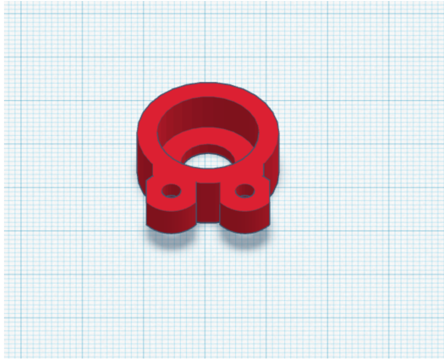


Figure 6.25: Single large brush holder, holds first upper brush.

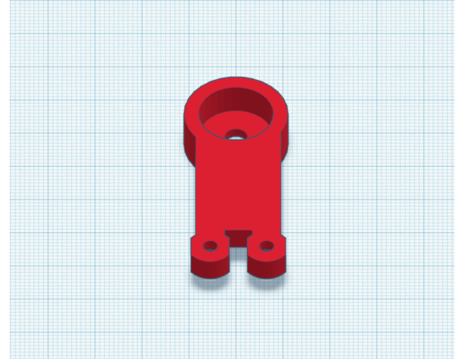


Figure 6.26: Single upper brush, holds middle upper brush.

Third holder, (Figure 6.27), is closest to the robot and holds one large and one small brush. Last holder, (Figure 6.28), supports two lower brushes, it is connected for stability with *M3 rod* to opposing holder.

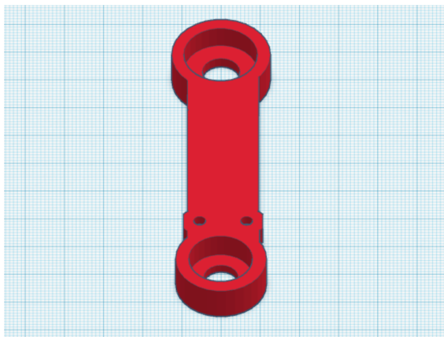


Figure 6.27: Double holder, holds third upper and lower brushes.

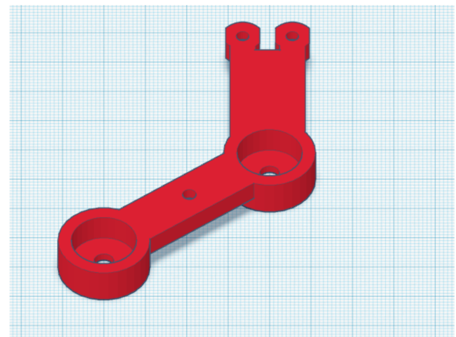


Figure 6.28: Bottom double holder, holds first and second lower brushes.

Figure 6.29 shows all sliders and holders put together on a real robot. Each holder is connected to slider with two M3 bolts and nuts. Nuts are attached to sliders from inner side, as visible on Figure 6.23 and Figure 6.24.

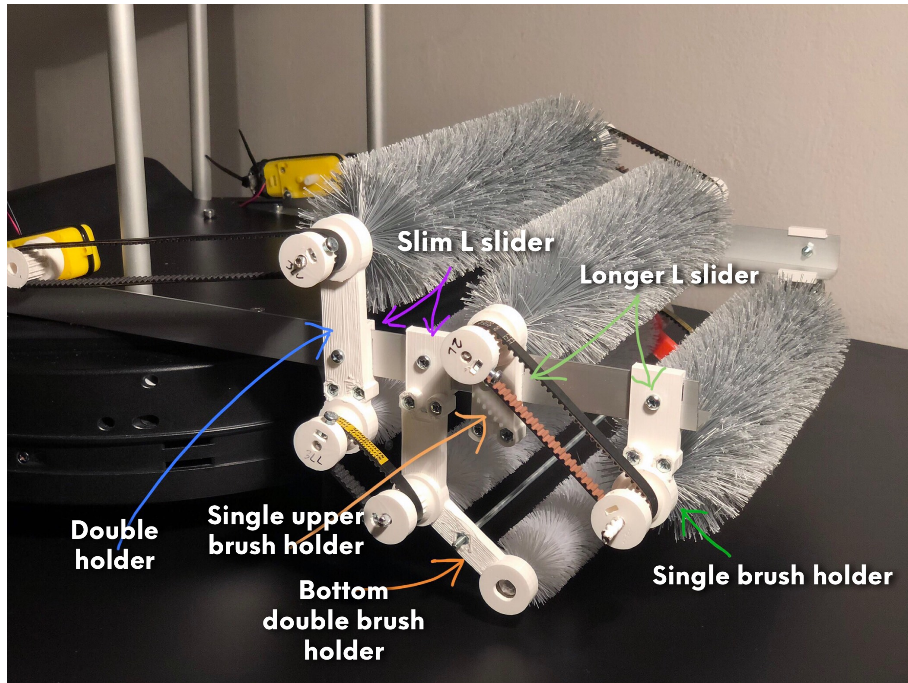


Figure 6.29: All sliders and holders put together on a real robot.

Last 3D printed part is pulley, but we will describe it in the subsection 6.7.2.

6.7.2 Other parts

Torque between motors and brushes is transferred by *timing belt* or *toothed belt*. To ensure that belts don't slide, we use *pulleys*, (Figure 6.30), which are gears fixed on the axle of a brush. There are various *profiles* of the tooth of the belts, and also distance between teeth. We decided to use HTD 3M timing belt, which has *curvilinear profile*, (Figure 6.31). 3M in name means distance between centers of neighbouring teeth in millimeters.

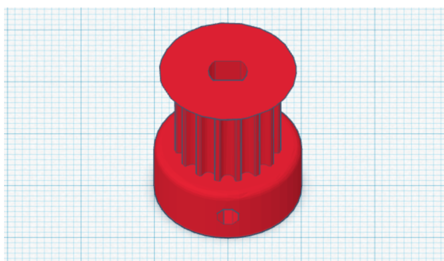


Figure 6.30: 3D model of pulley, with HTD tooth profile.

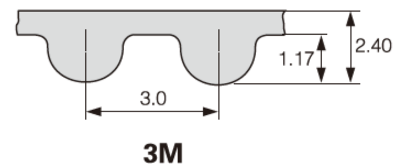


Figure 6.31: HTD 3M tooth profile, with pitch shown.

To minimize friction between brush axle and holder, we put axles into *ball bearings*. This ensures minimal friction and no degradation of 3D printed parts.

6.7.3 Iterative design

While designig the picking system, we came across several issues. First issue was that two lower brushes were not enough to bring shuttlecock safely to the

top of Kobuki robot (Figure 6.32). We solved this by adding third brush near the upper level of Kobuki, as shown in Figure 6.33.

Figure 6.32: Picking mechanism with two lower brushes.

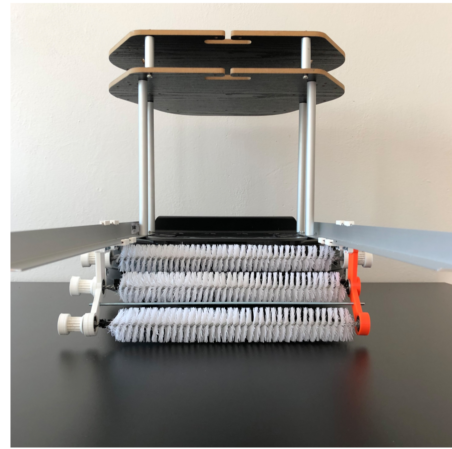


Figure 6.33: Picking mechanism with three lower brushes.

Second issue was that shuttlecock was sometimes oriented with feathers towards robot, and feathers struck upper frame of robot, (Figure 6.34). This was resolved by adding **smooth plastic layer**, as shown in Figure 6.35 spanning most of the robot's top frame and coming down below the closest small brush, therefore creating surface where feathers are not likely to stuck but are glided upwards.

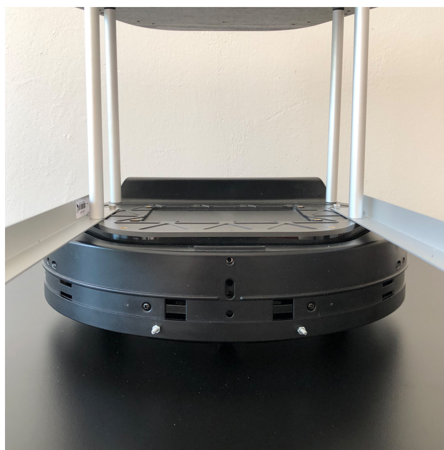


Figure 6.34: Kobuki's front top, without plastic layer.

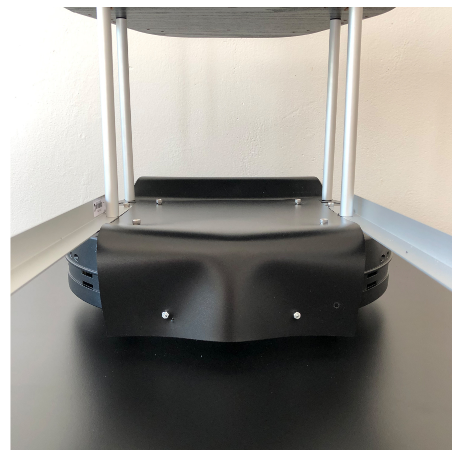


Figure 6.35: Kobuki's front top with added smooth plastic.

Another problem was that mechanism did not *catch*[25] shuttles onto the brushes, this was because first upper brush was too close above first lower brush, therefore not creating any pressure on feathers, but instead skidding on the featherless part of the shuttlecock. This was resolved by extending the upper brush more forward until it caught shuttlecocks reliably.

[25]Shuttle was not pulled upwards by first upper and first lower brush.

Lastly, on rare cases, shuttlecock could be stuck between four brushes, shown in Figure 6.36, *gliding* on two lower brushes. This was resolved by putting upper brushes *more closely* together, so that upper brush could push shuttlecock upwards with more pressure, (Figure 6.37).

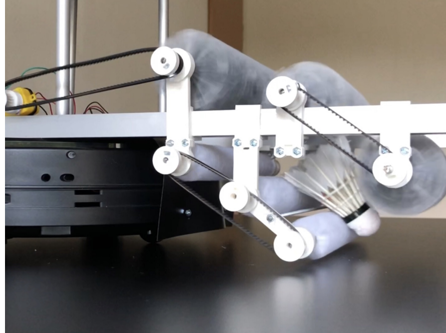


Figure 6.36: Larger distances between axles caused shuttles to stuck.

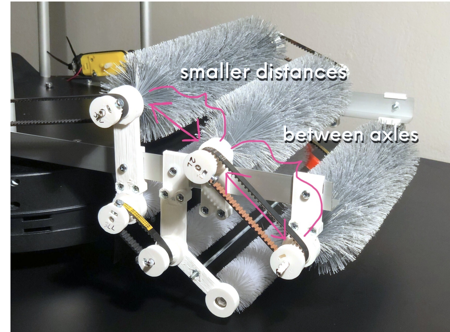


Figure 6.37: Smaller distances between axles of upper brushes.

After solving these issues, we did not observe any problems with picking mechanism. After fine tuning positions of brushes, we drilled holes through holders, shown in Figure 6.38, and L profile, and fixed it with M3 bolts, (Figure 6.39).

Figure 6.38: Drilled holes for fixed position.

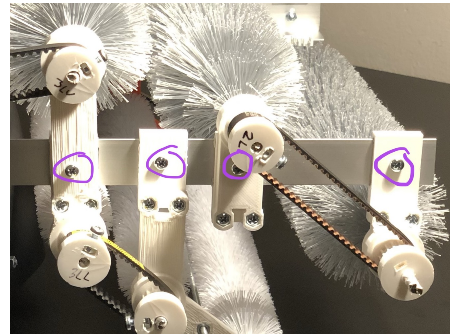


Figure 6.39: Fixed position of brushes.

6.7.4 Motor control

For motor control, we used *L298N*[56] motor module, shown in Figure 6.41. It can drive two DC motors and can be controlled by standard TTL[26] levels. It can be also configured to use PWM signal on both motors by removing jumpers on *ENA* and *ENB* pins. We control motor module using *Arduino Uno*[57] (Figure 6.40), popular and cheap microcontroller board. Schematic of wiring between Arduino, motor module, motors and battery is shown in Figure 6.42.

[26] https://learn.sparkfun.com/tutorials/logic-levels/all

Figure 6.40: Arduino Uno microcontroller board.

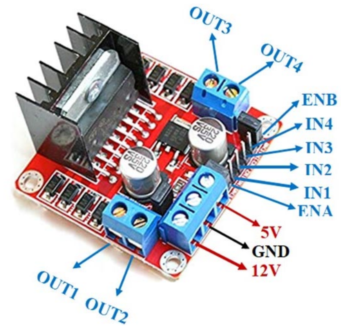


Figure 6.41: L298N module, image from [58].

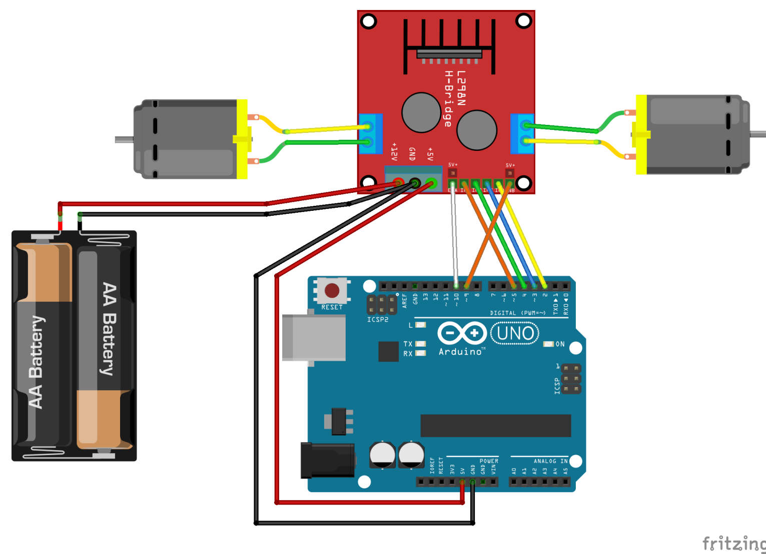


Figure 6.42: Wiring of the L298N module to the Arduino Uno and motors.

6.7.5 Arduino ROS node

For seamless communication between Arduino and ROS, we used ROS rosserial package[27]. We then created ROS node on Arduino that accepts *std_msgs/String* messages and turns on/off particular motor.

6.7.6 Arduino Leonardo

After using Arduino Uno microcontroller, we found out that it takes too much space on the Turtlebot top platform. We therefore searched for another suitable microcontroller that would be smaller, ideally could be attached directly on the L298N motor module. Because L298N is using 5V we were limited to boards with 5V logic, and we chose Beetle CM-32U4 Leonardo[28] developed by DFRobot. It has *ATmega32U4* chip and our ROSSerial program can be adapted with minimal changes. We soldered female pins on bottom of the board so that it would fit on top of the motor modules directly without any cables, as shown in Figure 6.44.

[27] http://wiki.ros.org/rosserial
[28] https://www.dfrobot.com/product-2475.html

It can be seen that it takes much less space than previous solution with Arduino Uno, (Figure 6.43).

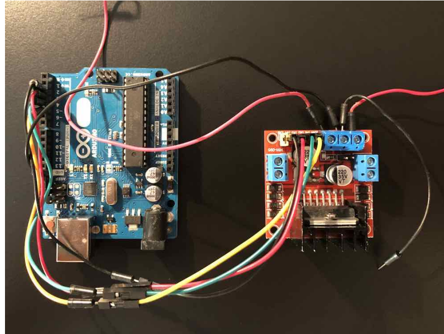


Figure 6.43: Old arrangement with Arduino Uno.

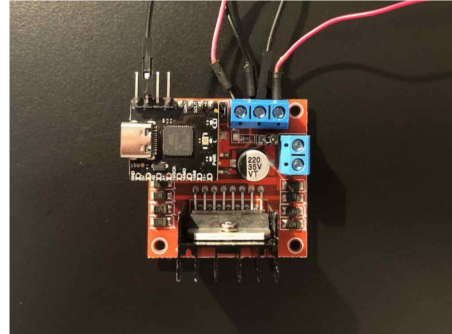


Figure 6.44: CM-32U4 on top of the motor module.

6.8 Working area customization

6.8.1 Allowed area

Badminton courts are often positioned side by side in a big hall, often as much as 4-6 courts, (Figure 6.45). We don't want the robot to wander to neighbouring courts while picking up shuttles, potentially disrupting play of other people. To do this, we need to make sure that the robot have to make path plans only in a specified area we want. We do this by marking area as *allowed*, as shown in Figure 6.46, and the rest as *disallowed* or *prohibited*. Since we are using navigation stack of ROS, and we are using global/local planners on a map made by mapping node, we can modify the already made map, so that robot won't create plans going to the neighbouring courts.

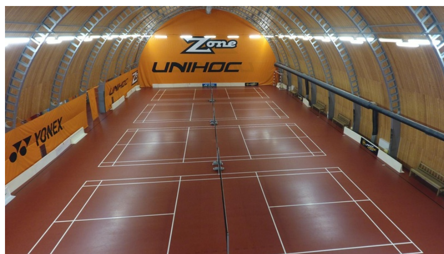


Figure 6.45: Multiple badminton courts side by side

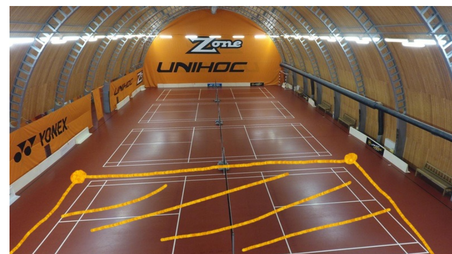


Figure 6.46: One court with allowed area.

Part of an occupancy grid map, (Figure 6.47), that we want to mark as allowed, can be specified by a polygon. We specify positions of vertices of the polygon, marking the allowed area, (Figure 6.48). To make it easier for users of our robot, we created node that uses *interactive markers*[29] of RViz. This way, user can pull the vertices around in XY plane, and our node will re-create the

[29] https://wiki.ros.org/interactive_markers

polygon marking the area. This polygon will be used as *static layer*[30] in the global costmap, with *max merging*.

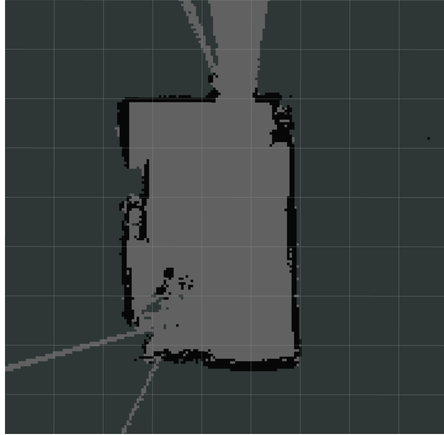


Figure 6.47: An occupancy grid map.

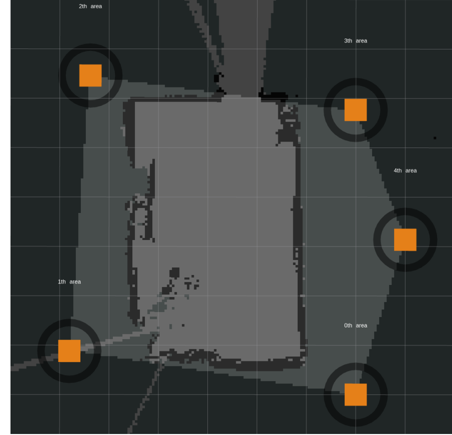


Figure 6.48: Allowed area markers (orange) with delineated area (light grey).

6.8.2 Allowed detection area

Another problem we can solve with the interactive markers, is specifying the area where shuttles can be picked, (Figure 6.49). This is different from allowed area for planning, because we want robot to *be able* to make a path to the home position, described in detail in subsection 6.8.4, where shuttles will be offloaded, but we *don't want* these same shuttles be picked up again. We do this similarly to allowed area, where we create new occupancy grid map from original /map that we get from mapping node and publish it on */map_darea*[31] topic.

[30] http://wiki.ros.org/costmap_2d/hydro/staticmap

[31] As for "detection area".

Figure 6.49: Detection area (white) between detection area markers (blue).

6.8.3 Waypoints

Next, we can use interactive markers for waypoints the robot will move between, while not picking any shuttles. We represent waypoints by black cubes (Figure 6.50). They can be moved around in plane by mouse similarly to area markers. Since RGBD cameras have relatively small field of view (Table 4.1), we can use waypoint markers to *cover more area*. Because robot is moving between waypoints, and rotating after it arrives to and when it exits waypoints, we can use this functionality to *cover larger area* for detections.

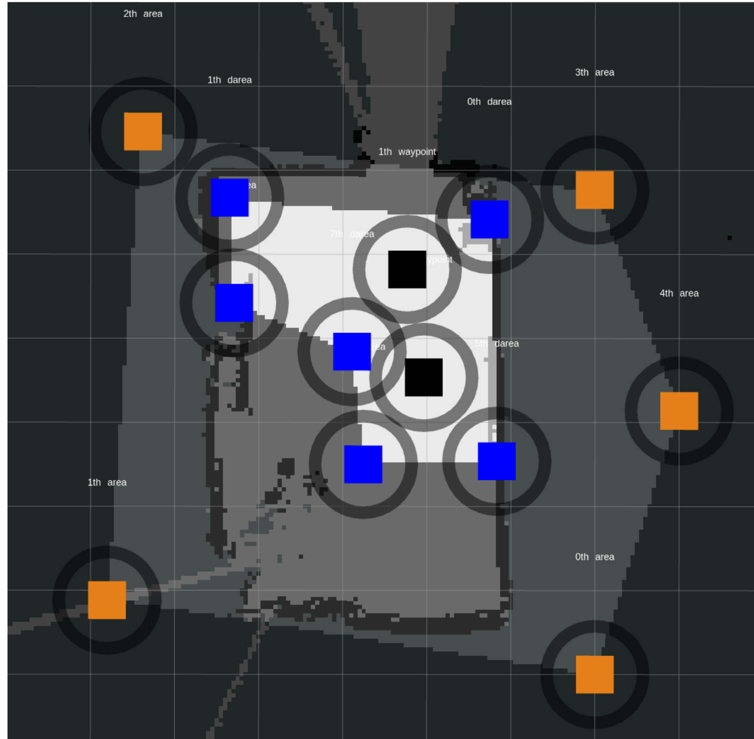


Figure 6.50: Movable waypoints (black).

6.8.4 Home position

Home position, (Figure 6.51), is a place where robot will drive off when it's full of shuttlecocks. We represent home position with RViz interactive marker with *green* color to distinguish it from other markers. Allowed area, Detection area and waypoints have ability to *add/remove* markers with right mouse clicking on marker and then selecting *add* or *remove*. Since home point marker is only one, we won't use adding/removing functionality and instead we can add useful commands for robot, such as *go to home position* or *return to work*. When user clicks on a menu selection, publisher inside callback sends a message on a given topic (to trigger a state change for example).

Figure 6.51: Home position marker (green) with menu with commands for robot.

6.8.5 Detection area visualization

To visualize correctness of the detection area delineated by markers, we created a test launchfile:

```
roslaunch shuttlebot_control marker_test.launch
```

It creates points colored green if they are inside detection area and blue if they are outside. Because home point is draggable, it can be interactively visualized in RViz that the detection area works correctly, without real robot or simulation, (Figure 6.52).

Figure 6.52: Points in a grid around home position, queried if they lie in the detection area.

7. User documentation

In this chapter we describe how to set up the robot on a badminton court. In the first section, we list what we need to successfuly operate the robot. In section mapping we show how to create map that will be subsequently used in the autonomous phase. In the autonomous working section we describe how to modify the behaviour of robot to suit specific needs and how to interact with the robot when it is working.

7.1 Setup

We need two parts, assembled robot and notebook. After turning on the robot, access point with name "xavier" will be created, to which we can connect with a notebook. Necessary software requirements for robot and notebook and are described in Appendix, in section A.1 and section A.2. Notebook will serve for visualization of robot data and giving robot commands (such as: go home, stop, start, etc..) and interactively setting area the robot can go to, and where it can detect shuttlecocks and setting the position where it would drop off collected shuttlecocks.

7.2 Mapping the court

When we bring robot to a completely new badminton court, we need to create a map of the court and surroundings. We cannot reuse the same map since every hall and building is different and we want to have optimal performance in each hall, therefore we will create new map for each different hall. After map is created, we can reuse it as many times as we like, on condition that the surroundings don't change, i.e. If we come to a same hall after a year with a benches or chairs in front of a wall, we would need to create a new map again. We start mapping by running following roslaunch on the robot:

```
roslaunch shuttlebot_control shuttlebot_mapping_total.launch
 ↪  localization:=false
```

On notebook we open rviz with for visualizaton and rqt for driving:

```
roslaunch shuttlebot_control shuttlebot_rviz.launch
```

We then create map by driving around court until a map is created. We can see progress in the RViz, as seen in Figure 7.1. After the map is saved, we are ready for the autonomous phase.

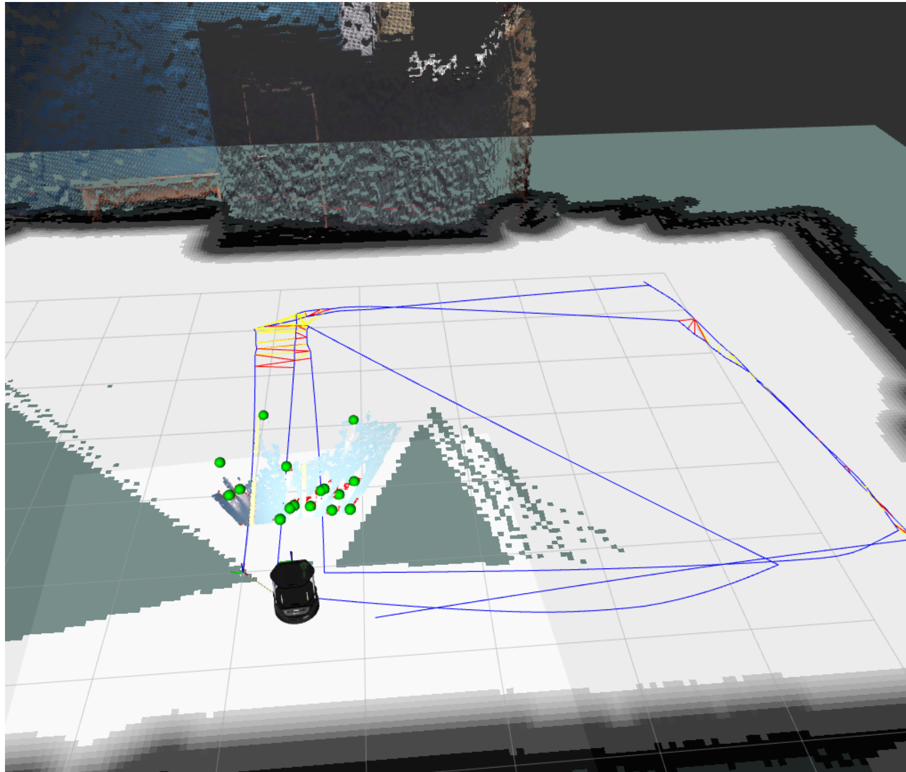


Figure 7.1: Blue lines is the path robot took during mapping.

7.3 Autonomous working

After we created map, we should run following roslaunch to start the the picking process.

```
roslaunch shuttlebot_control shuttlebot_working.launch
```

We can run robot with a list of waypoints between which it will drive, and home position where robot will "empty" itself of the shuttles. If we don't know the number of the waypoints beforehand, we can add them interactively in RViz. We can then drag them on needed locations on court, as shown in video[1].

Next, we give robot signal to start picking shuttles. If there are not any shuttles present, it will just drive between waypoints. As soon as shuttles starts arriving, it will continuously start to pick them up. After determined amount of shuttles, (for example 10), robot will enter emptying phase, which consists of driving to home position, and waiting for the shuttles to be taken off the robotic base. After we took the shuttles of the robot, we press the button to activate autonomous phase once more. This will happen repeatedly until we tell the robot to pause or stop.

[1]https://www.youtube.com/watch?v=acHEjFpJ6vw

8. Results

In this chapter, we present results of our proposed solution. Robot should be able to recognize shuttlecocks from 2D image camera data reliably. We show this by training neural network and evaluating it on previously unseen images with shuttlecocks. Next, we show successful collection of shuttlecocks by our picking mechanism with rotary brushes. Important part of our solution, is using interactive allowed area markers to delimit working area of robot, and interactive detection area markers to delimit detection area of robot.

8.1 Shuttlecock detection

Our created dataset of real shuttlecocks on badminton courts contains total of 2476 images, with 18618 total objects (shuttlecocks). Positions of shuttlecocks across dataset we collected and created is symmetrical around vertical axis (Figure 8.1, left) with most of the shuttlecocks located below the half of the image and more shuttlecocks located in the bottom of the image (y=1, x=1 is bottom right corner of the images). This means that when we created dataset from camera attached on moving robot, the shuttlecocks were positioned randomly on both sides of the images. We can also see that *size* of the majority of bounding boxes is relatively small (objects are further away) compared to the width and height of the image, with larger objects also present.

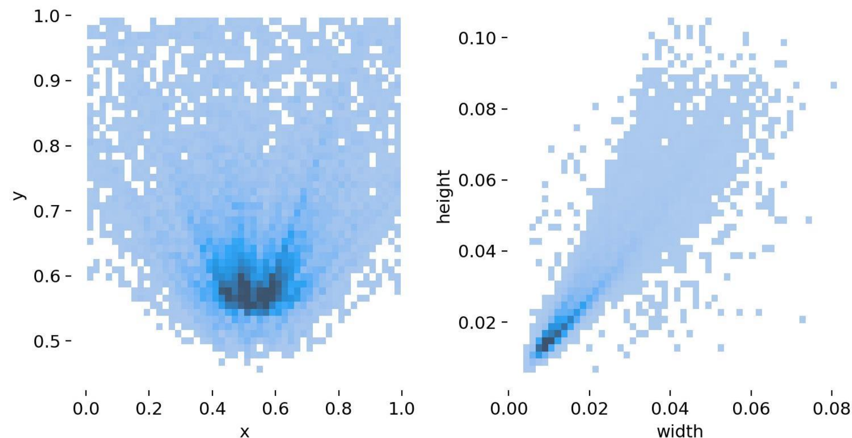


Figure 8.1: Position and size of bounding boxes across dataset we created.

During training and evaluation of machine learning models such as neural networks, datasets are usually split into three disjoint sets: *train*, *validation*[1], and *test*. Model is then trained only on train set. Training is done in epochs, where each epoch is one pass over training data, and after each pass, the model's performance is evaluated on validation set. As we observe performance of model after each epoch, evaluated on unseen validation data, this gives us insight if the model's performance is increasing with more time spent on training. Validation set can also be used for tunning different hyperparameters of model or training,

[1] Or development set.

such as parameters of optimizer, or regularization techniques such as dropout. Test set is then used as final evaluation of a best model, trained both on train and validation sets. Since we want to evaluate performance of model on *previously unseen* data, if we didn't use validation set, we could try many different hyperparameters to try to get as high accuracy on test set using our knowledge of previously tried hyperparameters.

8.1.1 Evaluation metrics

In this subsection we list several metrics used for evaluation of object detection models [59]. When we evaluate classification for two classes 1 and 0, we compare prediction of model with ground truth value. Therefore we can get four results:

model predicted 1 and ground truth is 1, that is *True positive*
model predicted 1 and ground truth is 0, that is *False positive*
model predicted 0 and ground truth is 1, that is *False negative*
model predicted 0 and ground truth is 0, that is *True negative*[2]

Precision is a metric comparing correctly predicted positive examples to all positive predictions:

$$\text{Precision} = \frac{TP}{TP + FP}$$

Recall is a metric comparing correctly predicted positive examples over all positive examples:

$$\text{Recall} = \frac{TP}{TP + FN}$$

Metric used in evaluation of object detection is Intersection over Union or IoU (Figure 8.2). We say that object is correctly predicted if the predicted class matches ground truth class and bounding box of prediction intersects bounding box of ground truth at least partially. We can then set threshold for IoU as a cutoff, for example IoU needs to be greater than 0.5. Higher IoU thresholds therefore means better predictions in terms of alignment of bounding boxes of predictions and ground truth examples.

[2] This is usually ignored in object detection metrics, since there is infinitely many true negatives. (i.e. model could predict 1000s of boxes with negative predictions where there is nothing) [59].

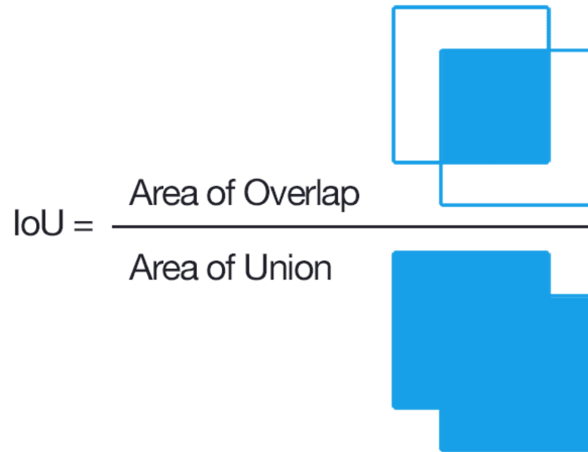


Figure 8.2: Intersection over union in object detection, from [60].

Lastly, two used metrics are mean average precision $mAP_{0.5}$[13] and $mAP_{0.50:.05:.95}$[61]. Mean average precision is computed as area under the precision recall curve, an example of precision recall curve is shown in Figure 8.4. Number in subscript of mAP is IoU threshold and $mAP_{0.50:.05:.95}$ means averaging mAP over IoU thresholds from 0.5 to 0.95 by 0.05 steps.

8.1.2 Training and results

We split our dataset of 2476 images randomly into test, val and train set by 8:1:1 ratio, that is 1982 images were used for training, 247 for validation and 247 for final evaluation. We trained YOLOv5s[3] model which has 7.2 million parameters, and used batch size 16 and trained for 200 epochs.

As we can see in the Figure 8.3 the losses are decreasing with number of epochs, while the precision mAP and $mAP_{0.50:.05:.95}$is increasing.

[3]There are larger versions of YOLOv5 models with more parameters, but they also have slower inference speeds, therefore we choose middle ground between performance and inference speed.

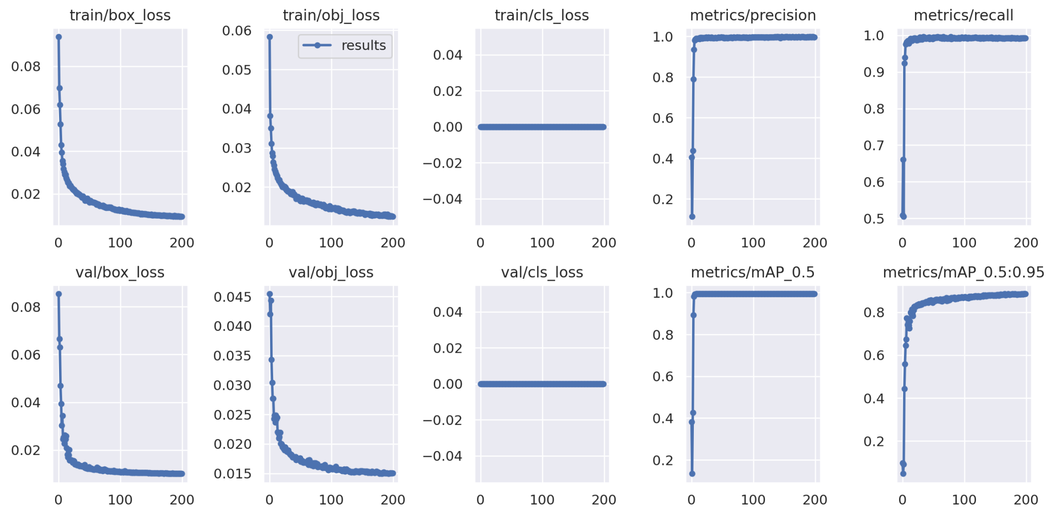


Figure 8.3: Losses, precision, recall and mAP as function of number of epochs trained.

Precision - recall curve (Figure 8.4) and F1 curve (Figure 8.5) of the first model.

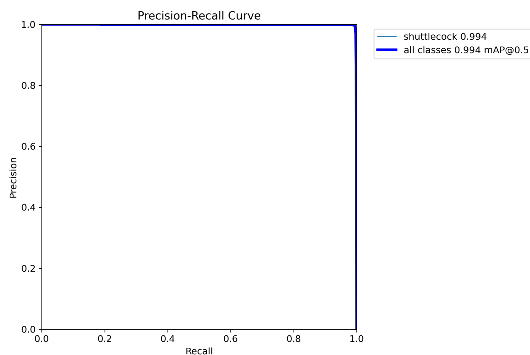


Figure 8.4: Precision - recall curve.

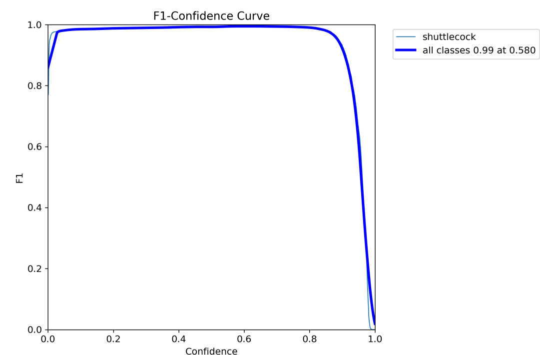


Figure 8.5: F1 curve.

Next, we trained model with batch size 16 and number of epochs 200 on test+val set and then evaluated on test set. Although we did not tune any hyperparameters on val set previously, we think it is interesting to observe increased performance with 10% more data available for training. We evaluated first model's performance against second model on the same test data, and while $mAP_{0.5}$ stayed same at 0.949, we observed modest increase of $mAP_{0.50:.05:.95}$from 0.853 to 0.858.

Following two pictures are examples of predictions of previously unseen test data, that contains objects similar to shuttlecocks. Our model correctly ignored these similar objects, such as white shuttlecock tube cap (Figure 8.6) and power socket (Figure 8.7), which was one of the objects that we mentioned in subsection 6.2.2 when we used model trained on several hundred images to help us with automatic dataset annotation[4].

[4]Although we still had to inspect each automatic annotation by hand, tweaking bounding

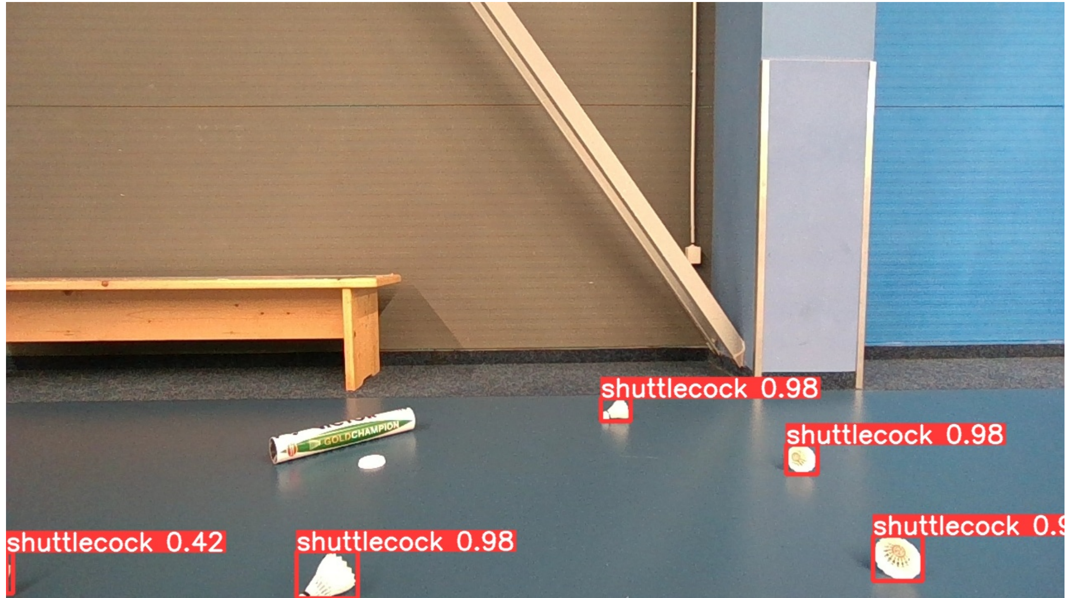


Figure 8.6: Example of object detection of previously unseen data.

Figure 8.7: Another example of prediction on previousy unseen data.

Lastly, we created small testing dataset of shuttlecocks located on a *green* court to test detection ability of trained model on different court colors. Dataset contains 26 images with 103 objects. Our model achieved $\text{mAP}_{0.5}$=0.995 and $\text{mAP}_{0.50:.05:.95}$=0.892 . Our model correctly predicted even hard object detections such as shuttlecock in direct reflection of sunlight, shown at Figure 8.8.

---

boxes by little, but we at least found several false positives that we could include to our training set.

Figure 8.8: Model trained on shuttlecocks on blue court can also make accurate predictions on other court colors such as green. Notice correct prediction of shuttlecock in reflection of sunlight.

8.2 Picking mechanism

We showcased working picking mechanism (Figure 8.9 , Figure 8.10) to visitors at day of open doors at Faculty of Mathematics and Physics[5]. Videos of working mechanism can be also seen on Youtube[6,7].

Figure 8.9: Idle robot waiting for command to pick the shuttle.

Figure 8.10: Succesfully picked shuttlecock.

[5] On 22.11.2022, report from event at: https://www.mff.cuni.cz/cs/verejnost/aktuality/matfyz-se-otevrel-uchazecum-o-studium

[6] https://www.youtube.com/shorts/kckWgyIh_Yo

[7] https://www.youtube.com/shorts/uJs3kkeHSKo

8.3 Allowed area

Allowed area demarcation for robot planning was tested on a real robot. We put waypoints on a map in RViz and let the robot move between them. Then we placed markers representing allowed area so that robot's planned path would have to intersect it if they were not there. Robot successfully planned its path between the waypoints, avoiding the the parts of the map outside allowed area as can be seen in Figure 8.11.

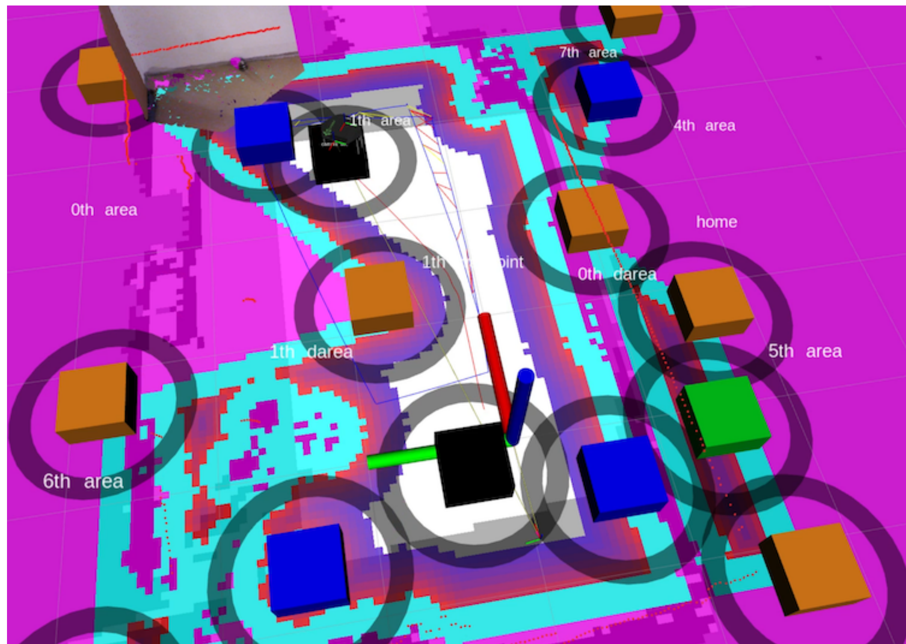


Figure 8.11: Robot moving between waypoints (black), avoiding outside of the allowed area (pink), represented by allowed area markers (orange).

We then put the allowed marker from the other side to the center of the map, so that robot would have to make a plan from the other side, and it did so, as can be seen on Figure 8.12. This process happened in real-time and interactively. After moving markers to the new location, new allowed area was redrawn for the robot to use, which can be seen on *video on youtube*[8]. Difference compared to allowed area shown in subsection 6.8.1 is that ROS navstack automatically *inflates* the area around obstacles by robot's footprint, which was mentioned in section 5.5.

[8] https://www.youtube.com/watch?v=acHEjFpJ6vw

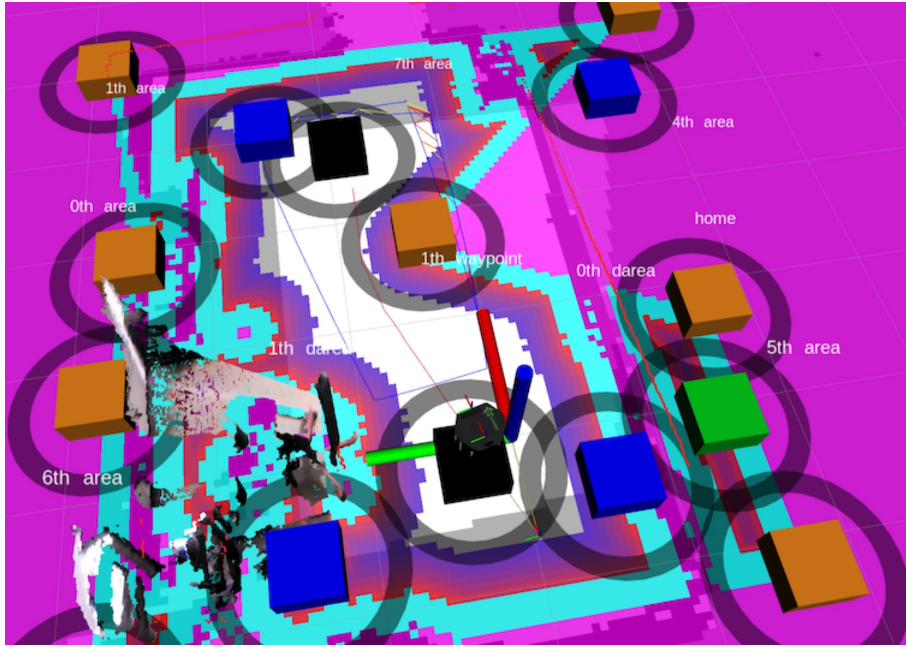


Figure 8.12: After we put allowed area markers from the other side of the map, robot had to take other path (thin red line between waypoints).

Lastly, we tested localization on a real robot in presence of allowed area markers. Natural question could be: *"Can robot relocalize in presence of custom virtual obstacles such as our allowed area?"*. As can be seen in Figure 8.13 and on video[9] robot successfully does relocalization in presence of virtual obstacles of allowed area, and that is because RTAB-Map mapping and localization node (section 5.4) does not use our virtual obstacles, but they are used as static layer by planners in navstack later on during path planning.

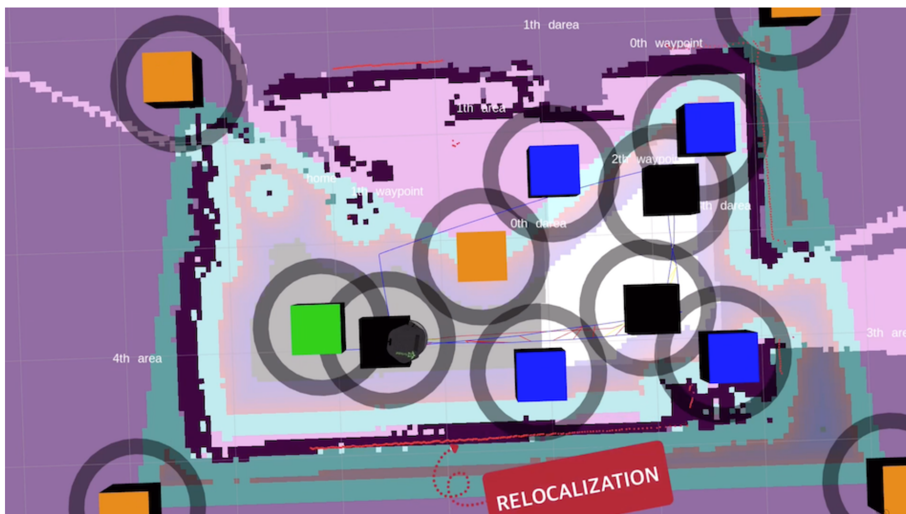


Figure 8.13: Our allowed area does not prevent robot from relocalization.

[9]https://www.youtube.com/watch?v=SsB-IQQB1ZY

8.4 Detection area

We tested detection area demarcation in a similar way to section 8.3. We put shuttlecocks in front of the robot. Robot detected shuttlecocks with neural network, and estimated locations of shuttlecocks. Then it filtered these positions using ROS node with respect to the *detection area* we created from detection area markers, colored *green* (Figure 8.14) if they are inside detection area and *blue* (Figure 8.15) if they are outside. This was done in interactively in real-time, as demonstrated on video[10].

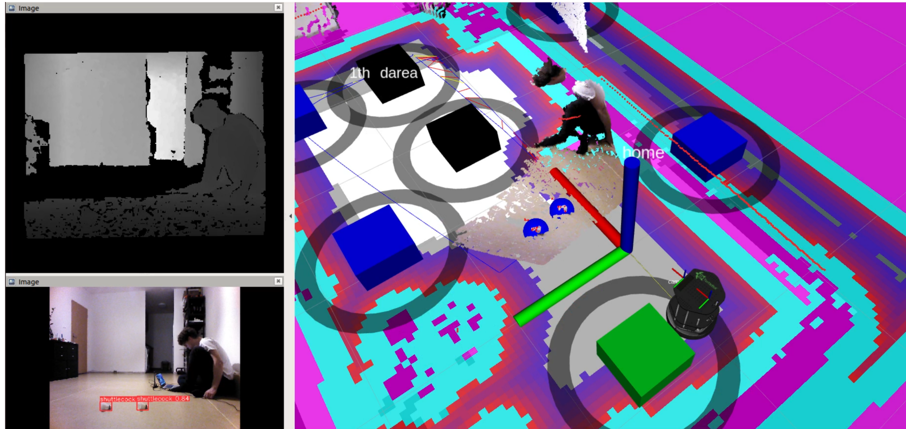


Figure 8.14: Shuttlecocks are detected (bottom right) and are regarded as outside of the detection area (blue spheres in centre of the image).

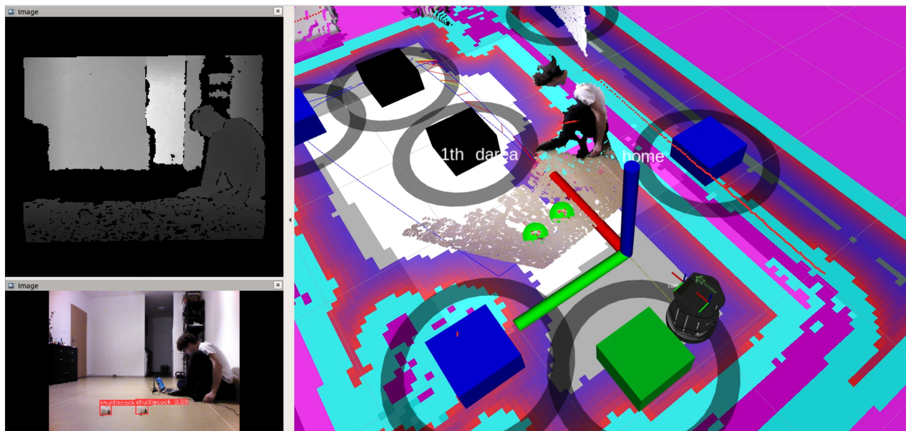


Figure 8.15: After we moved detection area markers so that shuttlecocks would be inside detection area, filter node classified them as inside the area and colored them green.

8.4.1 Discussion

During development and testing, we tried to increase performance of our system by all the means. Our solution contains many programs developed by us, but also necessary third party ROS packages that we use, such as RTAB-Map

[10] https://www.youtube.com/watch?v=E_6Vl6_PnLQ

for mapping and localization or navstack for planning, and these have high number of parameters that could affect performance of robot in real life. After we successfully tested our part of the solution on a real robot with shuttlecocks on various positions around robot, we turned our attention to trying to tune various parameters of the third party packages.

For example, there are several parameters of *DWA local planner* that we tried to modify, that would hopefully increase our performance. We tried several suggestions from navstack tunning guide [62] and although some weren't useful, others increased ability of shuttle picking. Increased planning window does not help, since longer window means robot can plan more smooth and curved local plans, which does not help us at all, since we want robot to go to shuttle straight on, and not by a curve because that decreases chance of picking the shuttlecocks by brushes from front of the robot. Conversely, smaller windows makes robot not plan ahead too much and therefore it has to adhere more to global plan which results in straighter path to the goal.

Next, we tried increasing number of vx and vtheta samples, this helped a bit. Decreasing xy_goal_tolerance from default value was catastrophic, because robot arrives at vicinity of goal and then wiggles around until it satisfies the small tolerance, this often resulted in not picked shuttle because it was pushed by the sides when robot wiggled closer and closer to the goal. This is exactly where we benefit from choosing wide rotary brushes for picking shuttlecocks, since we can use default value of goal position tolerance.

While before we used only RGBD camera for mapping and localization, the laser scans used to construct occupancy grids for navigations were constructed by sampling the small horizontal section of the camera itself, which has limited accuracy and field of view. We then tried to use real LIDAR as laser scan in stead of fake laser scan generated from depth data or pointclouds from camera. This improved accuracy of generated map. Another thing that was improved was clearing of obstacles, since LIDAR that we used have 270 degree field of view, compared to 70 of the RGBD camera. This means, that robot can clear obstacles also on his sides, which we found very helpful. We also found that quality and frequency of laser scans is improved since it is using TOF technology instead of constructing points from stereo camera and then sampling this data to get fake laser scans.

Lastly, we observed that while robot is moving and rotating at high speeds, the rolling shutter of the camera can generate blurred images which are very hard for neural network to detect shuttlecocks, and also for mapping algorithm to localize. Also moving and rotating at high speeds rapidly increase odometry drift, which is again negative thing.

We therefore tried to limit working speeds, mainly rotation velocity since that produced most of the blurry images.

Lastly, we were very satisfied with the allowed area demarcation, since it guarantees that robot won't move from the court we are training on, as long as robot relocalizes itself from time to time using RTAB-Map package.

Although we confirmed high performance of detection of shuttlecocks by neural network, the detection area that we can interactively set in RViz acts yet as another protection from false positives that could otherwise happen, which is very welcomed contribution to reliability of our solution.

9. Conclusion

In our thesis we have developed control system based on state machines for the robot that pick shuttlecocks located on a badminton court, by using robotic middleware ROS Melodic - Robot Operating System and creating our programs as ROS nodes.

To accomplish goal of shuttlecock detection, we collected and annotated over 2500 images, which contained over 18500 objects.

We trained state of the art neural network to recognise shuttlecocks under various lightning conditions and court colors. We showed that neural network is able to make precise detections of shuttlecocks even on badminton courts of different color.

To use this state of the neural network with older robotic base Kobuki which was using older version of ROS, Melodic, we deployed it inside docker container running as a newer ROS Noetic node.

To estimate position of shuttlecock from 3D data of depth camera relative to the robot, we created program that integrates depth information from camera and neural network detections from ROS node running inside the container.

We integrated and tested multiple third-party ROS packages such as RTAB-Map for creating maps of the environment and navstack for planning and then used for mapping, localization and planning.

During development of our solution, we created virtual badminton court simulation and simulated Kobuki robot in Gazebo. We created plugin that simulates shuttle picking, which helped us with development of solution for a real robot.

Next, we designed and created fully working shuttlecock picking mechanism using 3D printer based on rotating brushes, that successfully and repeatedly picks up badminton shuttlecocks.

As part of our solution we created ROS nodes that interactively creates allowed area for robot by moving markers in RViz visualization program. This ensures robot does not wander to neighbouring courts during autonomous working. User can specify where robot should work by interactively moving waypoints.

User can also specify area where robot would drop off collected shuttlecocks and then return to work. As part of our solution we also created filtering node that filters detected shuttlecock positions based on detected area that user can specify.

Future work

It could be useful to test out other types of sensors such as RGB-D cameras with wider field of view.

Our solution could be adapted in other domains, such as picking of other objects. Because our solution is fairly general, only things that would have to be modified would be training neural network on another dataset to recognize objects in the other domains and picking mechanism.

Bibliography

[1] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European conference on computer vision*, Springer, 2014, pp. 740–755.

[2] D. Gunning, "Explainable artificial intelligence (xai)," *Defense Advanced Research Projects Agency (DARPA), nd Web*, vol. 2, no. 2, 2017.

[3] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J. J. Leonard, "Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age," *IEEE Transactions on robotics*, vol. 32, no. 6, pp. 1309–1332, 2016.

[4] G. Bresson, Z. Alsayed, L. Yu, and S. Glaser, "Simultaneous localization and mapping: A survey of current trends in autonomous driving," *IEEE Transactions on Intelligent Vehicles*, vol. 2, no. 3, pp. 194–220, 2017.

[5] BWF. (2019). Bwf statutes, section 4.1: Laws of badminton, BWF, [Online]. Available: https://extranet.bwfbadminton.com/docs/document-system/81/1466/1470/Section%204.1%20-%20Laws%20of%20Badminton.pdf (visited on 02/06/2021).

[6] S. Kitta, H. Hasegawa, M. Murakami, and S. Obayashi, "Aerodynamic properties of a shuttlecock with spin at high reynolds number," *Procedia Engineering*, vol. 13, pp. 271–277, 2011.

[7] BWF. (Jul. 20, 2021). Bwf world rankings. BWF, Ed., [Online]. Available: https://bwfbadminton.com/rankings/2/bwf-world-rankings/6/men-s-singles/2021/29/ (visited on 07/21/2021).

[8] D. B. Poudel, "Coordinating hundreds of cooperative, autonomous robots in a warehouse," *Jan*, vol. 27, no. 1-13, p. 26, 2013.

[9] E. J. Van Henten, J. Hemming, B. Van Tuijl, J. Kornet, J. Meuleman, J. Bontsema, and E. Van Os, "An autonomous robot for harvesting cucumbers in greenhouses," *Autonomous robots*, vol. 13, no. 3, pp. 241–258, 2002.

[10] (2021). Agrobot, [Online]. Available: https://www.agrobot.com/e-series (visited on 01/05/2023).

[11] Y. Yu, K. Zhang, L. Yang, and D. Zhang, "Fruit detection for strawberry harvesting robot in non-structural environment based on mask-rcnn," *Computers and Electronics in Agriculture*, vol. 163, p. 104 846, 2019.

[12] H. A. Williams, M. H. Jones, M. Nejati, M. J. Seabright, J. Bell, N. D. Penhall, J. J. Barnett, M. D. Duke, A. J. Scarfe, H. S. Ahn, *et al.*, "Robotic kiwifruit harvesting using machine vision, convolutional neural networks, and robotic arms," *biosystems engineering*, vol. 181, pp. 140–156, 2019.

[13] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International journal of computer vision*, vol. 88, no. 2, pp. 303–338, 2010.

[14] N. Ohi, K. Lassak, R. Watson, J. Strader, Y. Du, C. Yang, G. Hedrick, J. Nguyen, S. Harper, D. Reynolds, *et al.*, "Design of an autonomous precision pollination robot," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2018, pp. 7711–7718.

[15] (2021). Husky robot, [Online]. Available: https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/ (visited on 01/05/2023).

[16] ROS. (Feb. 15, 2021). Moveit! ros package, [Online]. Available: https://moveit.ros.org/ (visited on 01/05/2023).

[17] B. Arad, J. Balendonck, R. Barth, O. Ben-Shahar, Y. Edan, T. Hellström, J. Hemming, P. Kurtser, O. Ringdahl, T. Tielen, *et al.*, "Development of a sweet pepper harvesting robot," *Journal of Field Robotics*, vol. 37, no. 6, pp. 1027–1039, 2020.

[18] B. Arad, P. Kurtser, E. Barnea, B. Harel, Y. Edan, and O. Ben-Shahar, "Controlled lighting and illumination-independent target detection for real-time cost-efficient applications. the case study of sweet pepper robotic harvesting," *Sensors*, vol. 19, no. 6, p. 1390, 2019.

[19] J. Wang, "Ballbot: A low-cost robot for tennis ball retrieval," *Electrical Engineering and Computer Sciences University of California at Berkeley, Berkeley, CA, USA, Tech. Rep. No. UCB/EECS-2012-157*, 2012.

[20] C. H. Yun, Y.-S. Moon, and N. Y. Ko, "Vision based navigation for golf ball collecting mobile robot," in *2013 13th International Conference on Control, Automation and Systems (ICCAS 2013)*, IEEE, 2013, pp. 201–203.

[21] S. H. Yeon, D. Kim, G. Ryou, and Y. Sim, "System design for autonomous table tennis ball collecting robot," in *2017 17th International Conference on Control, Automation and Systems (ICCAS)*, IEEE, 2017, pp. 909–914.

[22] B. Sarah, L. Tianyi, L. Thomas, and M. Patel. (2016). Prosort cc-60. U. of Cambridge 2016, Ed., [Online]. Available: https://www.ifm.eng.cam.ac.uk/education/met/a/design/design-show-2015/#ProSort%20CC-60 (visited on 01/05/2023).

[23] J. M. O'Kane, *A gentle introduction to ros*, 2014.

[24] D. Franklin. (Nov. 6, 2019). Introducing jetson xavier nx, the world's smallest ai supercomputer. D. Franklin, Ed., [Online]. Available: https://developer.nvidia.com/blog/jetson-xavier-nx-the-worlds-smallest-ai-supercomputer/ (visited on 07/18/2021).

[25] Nvidia. (Jul. 27, 2021). Developer kit technical specifications. Nvidia, Ed., [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/ (visited on 01/05/2023).

[26] S. Labs. (2018). Zed camera and sdk overview, [Online]. Available: https://www.stereolabs.com/assets/datasheets/zed-camera-datasheet.pdf (visited on 01/05/2023).

[27] Intel. (2022). Intel realsense d455, [Online]. Available: https://store.intelrealsense.com/buy-intel-realsense-depth-camera-d455.html (visited on 01/05/2023).

[28] Orbecc. (2022). Astra camera, [Online]. Available: https://shop.orbbec3d.com/Astra (visited on 01/05/2023).

[29] czc.cz. (2022). Viking notebooková powerbanka, [Online]. Available: https://www.czc.cz/viking-notebookova-powerbanka-smartech-ii-quick-charge-3-0-40000mah-cerna/227132/produkt (visited on 01/05/2023).

[30] V. Braitenberg, *Vehicles: Experiments in synthetic psychology.* MIT press, 1986.

[31] N. J. Nilsson, "A mobile automaton: An application of artificial intelligence techniques," Sri International Menlo Park Ca Artificial Intelligence Center, Tech. Rep., 1969.

[32] R. Brooks, "A robust layered control system for a mobile robot," *IEEE journal on robotics and automation*, vol. 2, no. 1, pp. 14–23, 1986.

[33] E. Gat, R. P. Bonnasso, R. Murphy, *et al.*, "On three-layer architectures," *Artificial intelligence and mobile robots*, vol. 195, p. 210, 1998.

[34] R. J. Firby, "Adaptive execution in complex dynamic worlds," PhD thesis, Yale University, 1989.

[35] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[36] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[37] G. Jocher, A. Chaurasia, A. Stoken, J. Borovec, NanoCode012, Y. Kwon, TaoXie, J. Fang, imyhxy, K. Michael, Lorna, A. V, D. Montes, J. Nadar, Laughing, tkianai, yxNONG, P. Skalski, Z. Wang, A. Hogan, C. Fati, L. Mammana, AlexWang1900, D. Patel, D. Yiwei, F. You, J. Hajek, L. Diaconu, and M. T. Minh, *ultralytics/yolov5: v6.1 - TensorRT, TensorFlow Edge TPU and OpenVINO Export and Inference*, version v6.1, Feb. 2022. DOI: 10.5281/zenodo.6222936. [Online]. Available: https://doi.org/10.5281/zenodo.6222936.

[38] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, IEEE, vol. 3, 2004, pp. 2149–2154.

[39] (2020). Gazebo website, [Online]. Available: http://gazebosim.org/tutorials (visited on 01/05/2023).

[40] M. Sipser, *Introduction to the Theory of Computation.* Cengage learning, 2012.

[41] R. Balogh and D. Obdržálek, "Using finite state machines in introductory robotics," in *International Conference on Robotics and Education RiE 2017*, Springer, 2018, pp. 85–91.

[42] M. Labbé and F. Michaud, "Rtab-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation," *Journal of Field Robotics*, vol. 36, no. 2, pp. 416–446, 2019.

[43] G. Grisetti, R. Kümmerle, C. Stachniss, and W. Burgard, "A tutorial on graph-based slam," *IEEE Intelligent Transportation Systems Magazine*, vol. 2, no. 4, pp. 31–43, 2010.

[44] B. M. da Silva, R. S. Xavier, T. P. do Nascimento, and L. M. Gonçalves, "Experimental evaluation of ros compatible slam algorithms for rgb-d sensors," in *2017 Latin American Robotics Symposium (LARS) and 2017 Brazilian Symposium on Robotics (SBR)*, IEEE, 2017, pp. 1–6.

[45] M. Labbe. (Mar. 1, 2021). Setup rtab-map on your robot! [Online]. Available: http://wiki.ros.org/rtabmap_ros/Tutorials/SetupOnYourRobot (visited on 01/05/2023).

[46] D. Lu, D. Hershberger, and E. Marder-Eppstein. (Feb. 12, 2021). Global_planner, [Online]. Available: http://wiki.ros.org/global_planner (visited on 01/05/2023).

[47] ——, (). Costmap2d, [Online]. Available: http://wiki.ros.org/costmap_2d (visited on 01/05/2023).

[48] S. I. Nikolenko *et al.*, "Synthetic data for deep learning," *arXiv preprint arXiv:1909.11512*, vol. 3, 2019.

[49] cvat.ai. (Mar. 2, 2019). Computer vision annotation tool: A universal approach to data annotation, [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/computer-vision-annotation-tool-a-universal-approach-to-data-annotation.html (visited on 01/05/2023).

[50] Isaac I. Y. Saito. (2022). Smach, [Online]. Available: http://wiki.ros.org/smach (visited on 01/05/2023).

[51] R. Smith. (2022). Open dynamics engine, [Online]. Available: https://www.ode.org/ (visited on 01/05/2023).

[52] The OGRE Team. (2022). Object-oriented graphics rendering engine, [Online]. Available: https://www.ogre3d.org/ (visited on 01/05/2023).

[53] (2021). Sdformat (simulation description format), [Online]. Available: http://sdformat.org/ (visited on 01/05/2023).

[54] (Aug. 1, 2021). Collada dae format, [Online]. Available: https://docs.fileformat.com/3d/dae/ (visited on 01/05/2023).

[55] Yuijin Robot. (Oct. 30, 2012). Kobuki hardware drwaing, [Online]. Available: https://raw.githubusercontent.com/kobuki-base/kobuki_resources/devel/hardware/drawings/pdf/kobuki_base_01.pdf (visited on 01/05/2023).

[56] STMicroelectronics. (Jul. 18, 2022). L298 dual full-bridge driver, [Online]. Available: https://www.st.com/resource/en/datasheet/l298.pdf (visited on 01/05/2023).

[57] Arduino. (Aug. 4, 2022). Arduino uno r3, [Online]. Available: https://docs.arduino.cc/hardware/uno-rev3 (visited on 01/05/2023).

[58] Components101. (Jul. 19, 2022). L298n module, [Online]. Available: https://components101.com/modules/l293n-motor-driver-module (visited on 01/05/2023).

[59] R. Padilla, S. L. Netto, and E. A. B. da Silva, "A survey on performance metrics for object-detection algorithms," in *2020 International Conference on Systems, Signals and Image Processing (IWSSIP)*, 2020, pp. 237–242.

[60] A. Rosebrock. (). Intersection over union (iou) for object detection, [Online]. Available: https://pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/ (visited on 01/05/2023).

[61] (2022). Coco metrics, [Online]. Available: https://cocodataset.org/#detection-eval (visited on 01/05/2023).

[62] K. Zheng, "Ros navigation tuning guide," in *Robot Operating System (ROS)*, Springer, 2021, pp. 197–226.

[63] ROS. (2021). Ros installation, [Online]. Available: http://wiki.ros.org/melodic/Installation/Ubuntu (visited on 01/05/2023).

[64] D. Franklin. (2021). Nvidia jetson inference, [Online]. Available: https://github.com/dusty-nv/jetson-inference (visited on 01/05/2023).

List of Figures

1.1 Feathered shuttlecock . . . 4
1.2 Multi-shuttle training. . . . 4

2.1 Yonex feather shuttlecock . . . 8
2.2 Vortices create drag, from [6] . . . 9
2.3 Badminton court dimensions . . . 9
2.4 Picture of Kento Momota from Japan,currently no.1 player in the world, practicing with former Korean gold olympic medalist, Japan Head Coach Park Joo-Bong . . . 10
2.5 Hans-Kristian Vittinghus, no. 20 singles player in the world [7], responds to the author that even pro players like Rasmus Gemke (no. 12) lose time in training due to slow shuttle picking technique 11
2.6 Kiva Robot, now Amazon robotics . . . 12
2.7 Roomba vacuum cleaner . . . 12
2.8 Example of green court . . . 13
2.9 Example of hardwood court . . . 13

3.1 Cucumber robot . . . 15
3.2 Segmented cucumbers . . . 15
3.3 Agrobot E-Series . . . 15
3.4 Bounding boxes around recongized strawberries with solid points representing picking points, from [11] . . . 15
3.5 Creation of picking points, from [11] . . . 16
3.6 Kiwi robot with four arms . . . 17
3.7 Visual output of network and blob detector . . . 17
3.8 BrambleBee . . . 17
3.9 Sweet pepper robot . . . 18
3.10 Closeup of end manipulator . . . 18
3.11 Autonomous robotic tennis ball boy . . . 19
3.12 Golf ball picking robot from its wide view camera . . . 20
3.13 Table Tennis Ball Collecting Robot . . . 20
3.14 ProSort CC-60 manual picking mechanism. . . . 21
3.15 Shuttlecock Collector Machine . . . 21
3.16 Shuttlecock Collector / Ballsammler . . . 21

4.1 Nvidia Jetson Nano . . . 24
4.2 Nvidia Jetson Xavier NX . . . 24
4.3 Performance comparison of Nvidia Jetson computers, from[24] . . 25
4.4 ZED stereo camera. . . . 26
4.5 Intel Realsense D455 . . . 26
4.6 Astra camera . . . 26
4.7 Kobuki robotic base . . . 27
4.8 Viking external battery . . . 28
4.9 Braitenberg vehicle. . . . 29
4.10 Line follower. . . . 29
4.11 Shakey the robot. . . . 29

4.12 Example of data flow of possible robot in a SPA paradigm. 30
4.13 Three layered architecture according to Firby, from[34]. 30
4.14 Overview of R-CNN architecture. 31
4.15 Overview of architecture of YOLOv5 neural network, from [37]. . 31
4.16 Partial vs whole view of the world. 32

5.1 ROS equation 34
5.2 Example of possible state machine of the robot. 36
5.3 Survey of ROS compatibile SLAM packages, table from [42]. . . . 37
5.4 Loop closure detection in RTAB-Map viewer 37
5.5 Setup of RTAB-Map node on a robot, from [45] 38
5.6 Navstack with RTAB-Map. 38
5.7 Path planning by global planner from package move_base, from [46]. 39
5.8 Inflation of obstacles (red dots) from Occupancy grid, from [47]. . 39
5.9 Example of different court colors and materials. 41
5.10 Example of synthetic dataset, from [48]. 42
5.11 PincherX 100 Robot Arm by Trossen robotics 43
5.12 Example of brushing mechanism 43

6.1 Image from ZED camera. 46
6.2 Image from D455 camera. 46
6.3 Creating dataset manually for object detection by simple annotation tool. 47
6.4 Self-hosted version of CVAT. 47
6.5 Picture taken from camera on real court, not yet annotated. . . . 48
6.6 Same picture, manually annotated in CVAT. 48
6.7 Examples of false positives after gathering first set of data. . . . 48
6.8 Neural network bounding box vs. point cloud. Some points inside bounding box are far behind shuttlecock. 50
6.9 Shuttlecock in front of robot, inside Gazebo. 50
6.10 Estimated position of shuttlecock, from RViz 50
6.11 Multiple detected shuttlecocks on real court (top left) with estimated positions (right). 51
6.12 System is in first state, IDLE. 52
6.13 System is in second state, COLLECTING. 53
6.14 State machine of shuttlecock picking. 54
6.15 Gazebo world. 55
6.16 Contacts (pointed by arrow) detected between shuttle model and collision element (in orange). 56
6.17 Early version of picking mechanism. 57
6.18 Finished, working prototype of picking mechanism. 57
6.19 Kobuki drawing, with marked holes used for attaching L profiles, image from [55]. 57
6.20 Aluminium L profiles with drilled holes, attached to Kobuki robot. 57
6.21 Cut and sanded steel core of the brush. 58
6.22 Brush after applying hot glue to the core. 58
6.23 Slim L slider, used connected to bottom double brush holder. . . 58
6.24 Longer L slider, connected to two upper holders. 58

6.25 Single large brush holder, holds first upper brush. 59
6.26 Single upper brush, holds middle upper brush. 59
6.27 Double holder, holds third upper and lower brushes. 59
6.28 Bottom double holder, holds first and second lower brushes. . . . 59
6.29 All sliders and holders put together on a real robot. 60
6.30 3D model of pulley, with HTD tooth profile. 60
6.31 HTD 3M tooth profile, with pitch shown. 60
6.32 Picking mechanism with two lower brushes. 61
6.33 Picking mechanism with three lower brushes. 61
6.34 Kobuki's front top, without plastic layer. 61
6.35 Kobuki's front top with added smooth plastic. 61
6.36 Larger distances between axles caused shuttles to stuck. 62
6.37 Smaller distances between axles of upper brushes. 62
6.38 Drilled holes for fixed position. 62
6.39 Fixed position of brushes. 62
6.40 Arduino Uno microcontroller board. 63
6.41 L298N module, image from [58]. 63
6.42 Wiring of the L298N module to the Arduino Uno and motors. . . 63
6.43 Old arrangement with Arduino Uno. 64
6.44 CM-32U4 on top of the motor module. 64
6.45 Multiple badminton courts side by side 64
6.46 One court with allowed area. 64
6.47 An occupancy grid map. 65
6.48 Allowed area markers (orange) with delineated area (light grey). . 65
6.49 Detection area (white) between detection area markers (blue). . . 66
6.50 Movable waypoints (black). 67
6.51 Home position marker (green) with menu with commands for robot. 68
6.52 Points in a grid around home position, queried if they lie in the detection area. 69

7.1 Blue lines is the path robot took during mapping. 71

8.1 Position and size of bounding boxes across dataset we created. . . 72
8.2 Intersection over union in object detection, from [60]. 74
8.3 Losses, precision, recall and mAP as function of number of epochs trained. 75
8.4 Precision - recall curve. 75
8.5 F1 curve. 75
8.6 Example of object detection of previously unseen data. 76
8.7 Another example of prediction on previousy unseen data. 76
8.8 Model trained on shuttlecocks on blue court can also make accurate predictions on other court colors such as green. Notice correct prediction of shuttlecock in reflection of sunlight. 77
8.9 Idle robot waiting for command to pick the shuttle. 77
8.10 Succesfully picked shuttlecock. 77
8.11 Robot moving between waypoints (black), avoiding outside of the allowed area (pink), represented by allowed area markers (orange). 78
8.12 After we put allowed area markers from the other side of the map, robot had to take other path (thin red line between waypoints). . 79

8.13 Our allowed area does not prevent robot from relocalization. . . . 79
8.14 Shuttlecocks are detected (bottom right) and are regarded as outside of the detection area (blue spheres in centre of the image). . . 80
8.15 After we moved detection area markers so that shuttlecocks would be inside detection area, filter node classified them as inside the area and colored them green. 80

A. Installation documentation

This chapter describes software installation on the robot and notebook.

A.1 Robot

A.1.1 Jetson Xavier NX

To install Linux operating system with preinstalled Nvidia software, use steps described in:
https://developer.nvidia.com/embedded/jetpack-sdk-462

A.1.2 ROS

Install ROS Melodic according to [63]

A.1.3 Jetson - inference

Next we install pretrained neural networks for Nvidia Jetson platform [64] from [1]

```
sudo apt-get update
sudo apt-get install git cmake libpython3-dev python3-numpy
git clone --recursive https://github.com/dusty-nv/jetson-inference
cd jetson-inference
mkdir build
cd build
cmake ../
make -j$(nproc)
sudo make install
sudo ldconfig
```

A.1.4 Kobuki base

We have to build Kobuki packages from source for ROS Melodic, because official Kobuki release at https://github.com/yujinrobot/kobuki depends on packages that did not work with Melodic, we used script from:
https://github.com/gaunthan/Turtlebot2-On-Melodic
that downloads all dependencies. Lastly, there was need to downgrade OpenCV on Jetson platform so packages would succesfsully compile.
https://github.com/ros-perception/vision_opencv/issues/329

To test that everything is installed correctly, use:

```
source ./devel/setup.bash
roslaunch turtlebot_bringup minimal.launch
```

[1] https://github.com/dusty-nv/jetson-inference/blob/master/docs/building-repo-2.md

A.1.5 Other dependencies

RTABmap mapping package: http://wiki.ros.org/rtabmap_ros

PCL library : https://pointclouds.org/downloads/#linux

Rosserial package: http://wiki.ros.org/rosserial

SMACH package: http://wiki.ros.org/smach

All above-listed packages should be *melodic* versions.

A.1.6 Cameras

To install ZED ROS wrapper and SDK, follow instructions at:
https://www.stereolabs.com/docs/ros/
Software for Astra camera can be installed from:
https://github.com/orbbec/ros_astra_camera
Lastly, software for Intel realsense can be installed from:
https://github.com/IntelRealSense/realsense-ros/tree/ros1-legacy

A.1.7 Vision

YOLOv5: https://github.com/ultralytics/yolov5

CVAT: https://github.com/opencv/cvat
With self-hosted version instructions at: https://opencv.github.io/cvat/docs/administration/basics/installation/

A.2 Notebook

Notebook serves as controlling and visualization platform. We have installed Ubuntu 18.04 with ROS Melodic with latest packages, same as on Jetson. We connect to access point *xavier* running on Jetson. Jetson will have IP adress *10.42.0.1* and we set up static IP address of notebook to *10.42.0.176*. Networking in ROS is done by setting linux environmental variables in *~/.bashrc*. Since we changed these variables often (on different networks, or switched master IP to localhost), we created convenience scripts for setting these variables, they are namely:

```
export ROS_MASTER_URI=http://10.42.0.1:11311
export ROS_IP=10.42.0.1
export ROS_HOSTNAME=10.42.0.1

export GAZEBO_MASTER_URI=http://10.42.0.1:11345
export GAZEBO_IP=10.42.0.1
```

scripts are:

```
/shuttlebot/misc/set_master_ip.sh
/shuttlebot/misc/set_my_ip.sh
/shuttlebot/misc/switch_ip.sh
```

and can be called when installed like this:

```
set_master_ip 10.42.0.1
```

After setting IP of notebook correctly, we open RViz and can begin controlling robot with RViz UI and visualize data we get from the robot.

A.3 Source code, dataset and other files of our solution

A.3.1 Source code for robot

Source code for robot is organized as catkin packages, which is hosted at Github. To download packages use:

```
cd ~/catkin_ws/src
git clone https://gitlab.mff.cuni.cz/cervema/shuttlebot_public
cd ..
```

Then use:

```
catkin_make
```

to compile packages. Gazebo plugins in *badminton_court* directory are not a ROS package.

A.3.2 Shuttlecock dataset

Images for shuttlecock dataset (subsection 6.2.1) and annotations (subsection 6.2.2) can be downloaded from: https://gitlab.mff.cuni.cz/cervema/shuttlecock_dataset

A.3.3 Dockerfile

Docker file we used to build image that we deployed on the robot can be downloaded from:
https://gitlab.mff.cuni.cz/cervema/shuttlebot_docker

A.3.4 Yolo node

Because Yolo node was meant to run inside docker, and is using ROS Noetic, we put it in a separate repository. https://gitlab.mff.cuni.cz/cervema/yolo_node_public

Attachments

The attachment contains 3 directories. Firstly, directory **shuttlebot**, which contains:

- **badminton_court**, directory containing Gazebo worlds, models, meshes, Gazebo plugin code and original Blender files
- **misc**, directory containing convenience scripts and other miscellaneous files.
- **shuttle_distance_estimation**, directory containing shuttle_distance_estimation package
- **shuttlebot_control**, directory with shuttlebot_control package
- **README.TXT**

Secondly, directory **other** containing other two repositories.

Lastly, attachment contains directory **thesis**, which contains PDF document of this thesis.