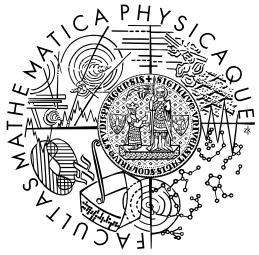

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

DIPLOMA THESIS

Martin Červeň

Control system for badminton shuttlecock collecting robot

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the diploma thesis: David Obdržálek
Study programme: Computer Science
Study branch: Artificial Intelligence

Prague 2021

I declare that I carried out this diploma 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: David Obdržálek, 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 hundreds of shuttles at once which are fed by coach from hand. After a short training period there are hundreds of shuttles scattered around the court, which need to be arranged in rows 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 simulator where we tested our solution.

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

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 *hundreds* of feathered shuttlecocks which are fed by coach from hand. After a short training period there are hundreds of shuttles 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, can 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 aim to create autonomous robot. It needs 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 goal of our thesis is to create control system for autonomous robot that picks badminton shuttlecocks and to test it on a real hardware.
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 visual input 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. If we had list of all positions, we could use some TSP solver with robot position as start and although TSP is NP-complete problem, it can be solved approximately for our purposes of hundreds of shuttles.

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, it's 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 algorithm we chose.

In the fifth chapter, **Proposed solution**, we present how we developed 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 we developed as part of our solution.

In the seventh chapter, **Results**, we will present results of running robot in real life collecting shuttlecocks and comparison to human performance.

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

User documentation will be presented in the Appendix A. We will describe how to setup robot, how to setup software and our control system.

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.

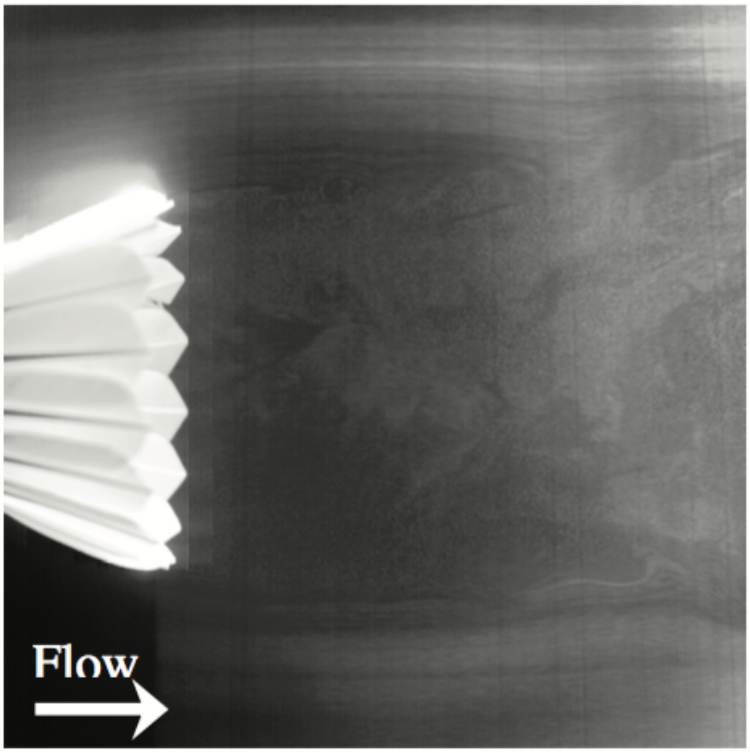

Figure 2.2: Vortices create drag, from [6]

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.

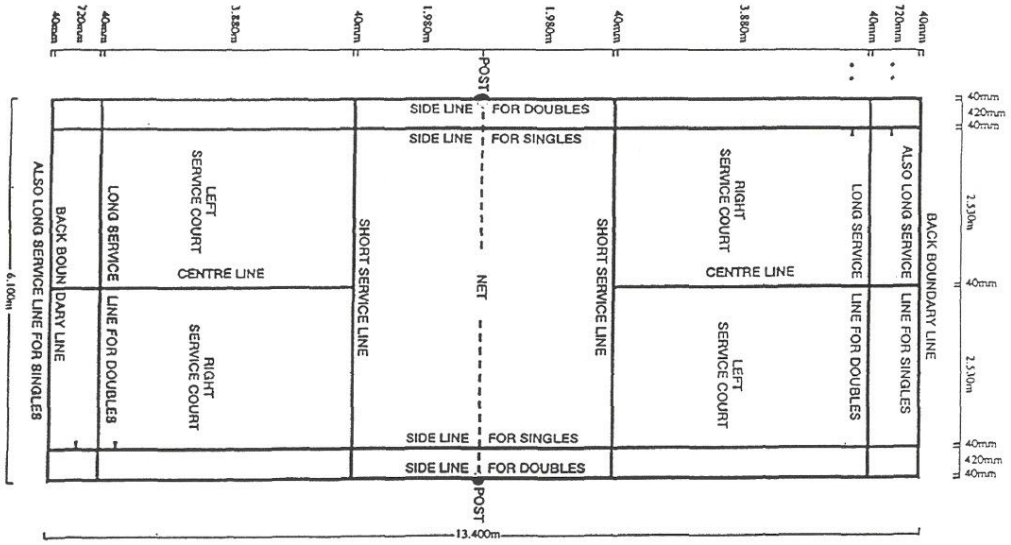

Figure 2.3: Badminton court dimensions

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.

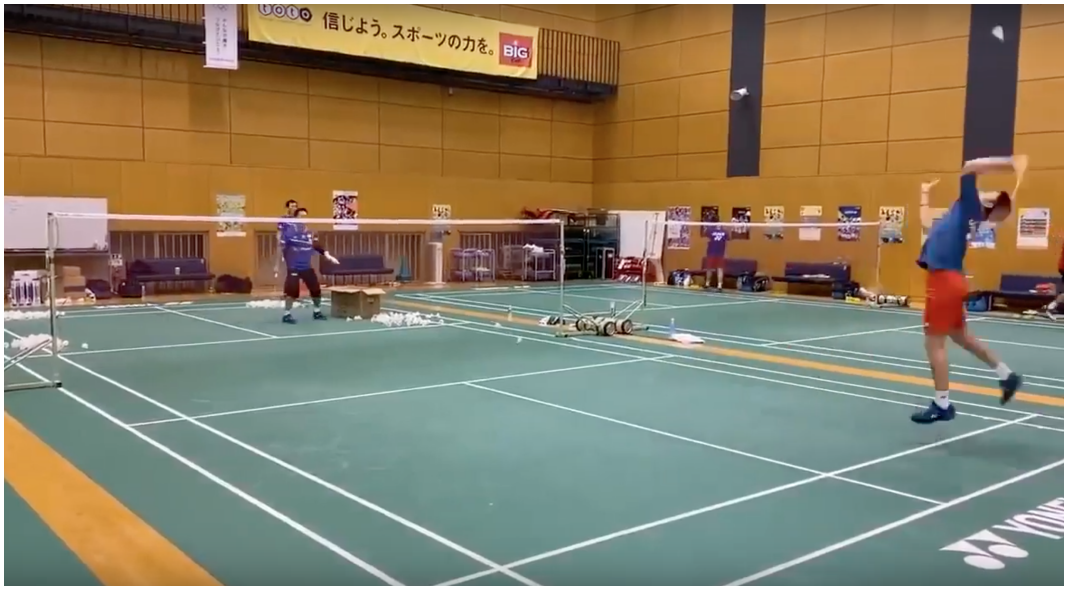

Figure 2.4: Picture of Kento Momota from Japan,currently no.1 player in the world, practicing with former Korean gold olympic medalist, Japan Head Coach Park Joo-Bong

2.3 Shuttle picking

During training, after using 100-1000 shuttles at once, players have to *manually* pick up shuttles and arrange them into rows to be used again in next excercise. 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.

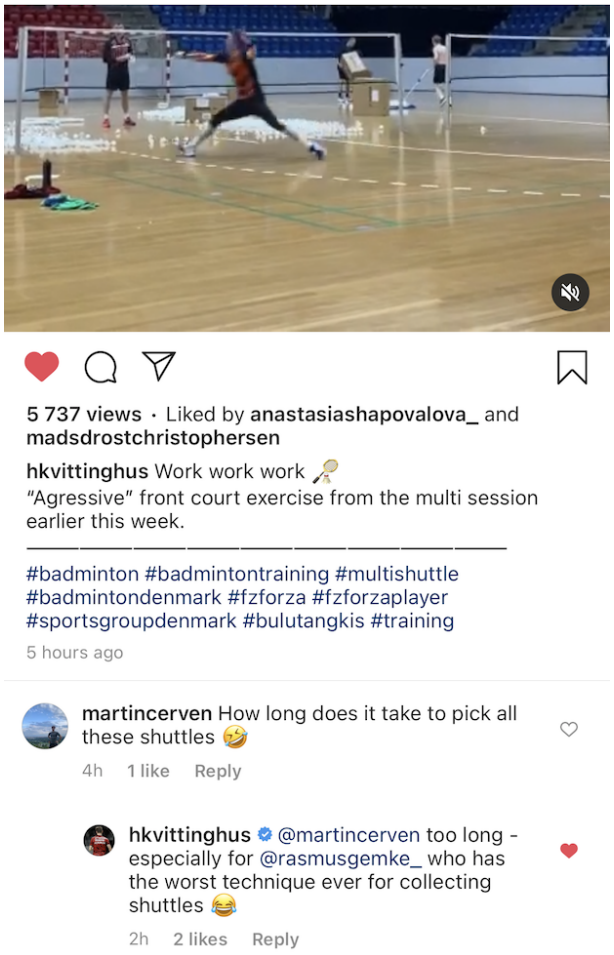

Figure 2.5: Hans-Kristian Vittinghus, no. 20 singles player in the world [7], responds to the author that even pro players like Rasmus Gemke (no. 12) lose time in training due to slow shuttle picking technique

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.

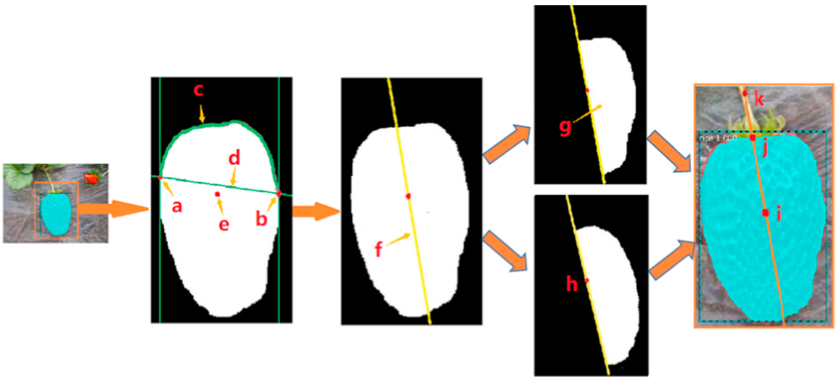

Figure 3.5: Creation of picking points, from [11]

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] https://stats.stackexchange.com/questions/193082/what-is-pre-training-a-neural-network

Figure 3.6: Kiwi robot with four arms

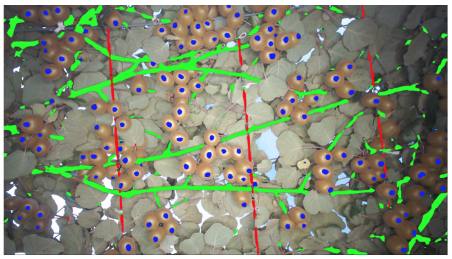

Figure 3.7: Visual output of network and blob detector

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

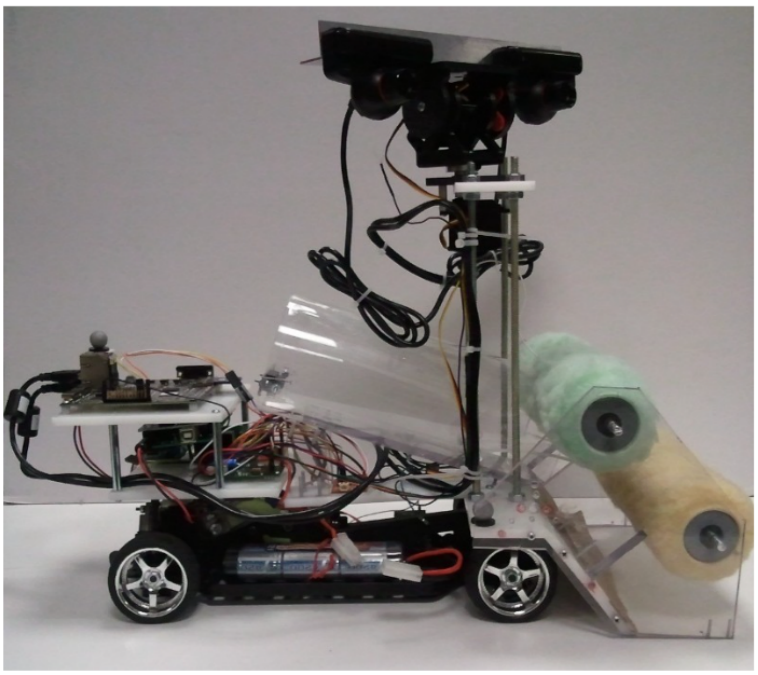

Figure 3.11: Autonomous robotic tennis ball boy

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.

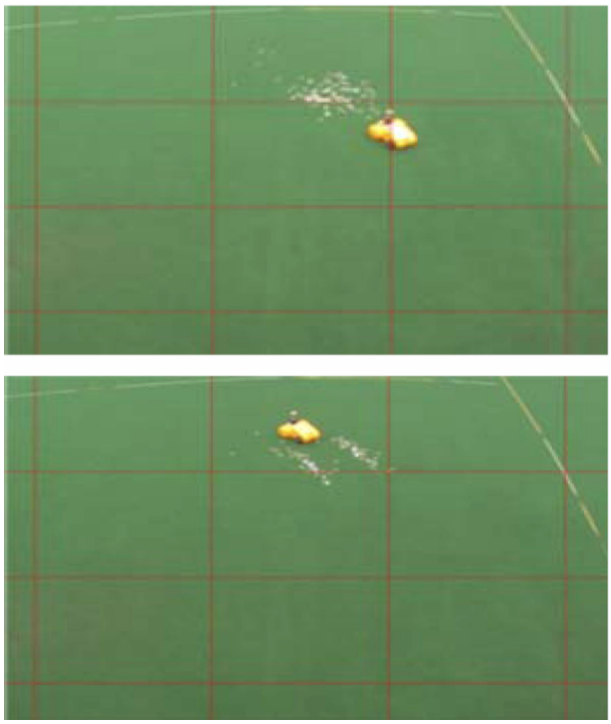

Figure 3.12: Golf ball picking robot from its wide view camera

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.

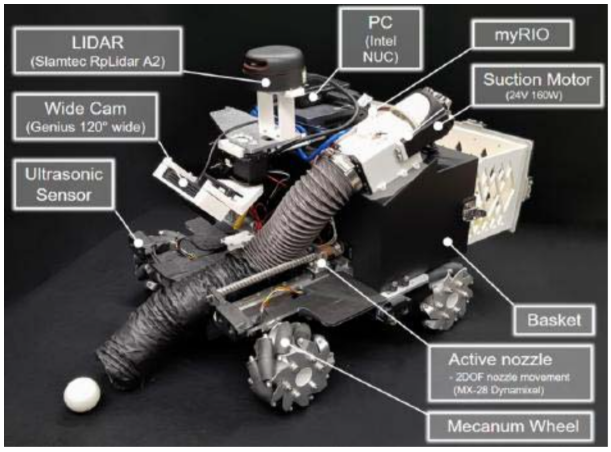

Figure 3.13: Table Tennis Ball Collecting Robot

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

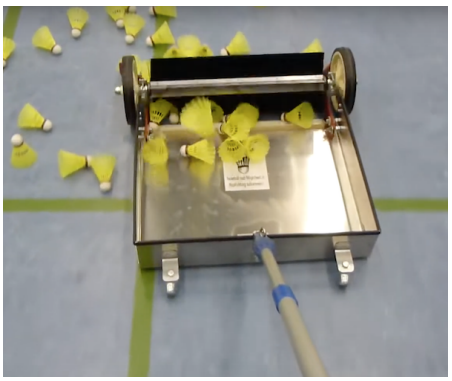

Figure 3.16: Shuttlecock Collector / Ballsammler

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=4hODzsYsZ7A

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.

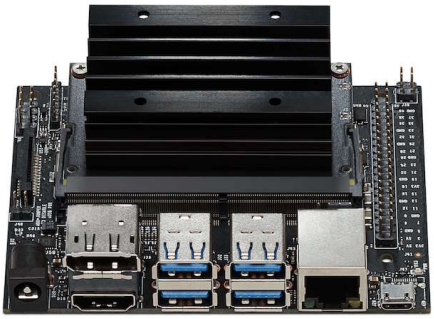

Figure 4.1: Nvidia Jetson Nano

Figure 4.2: Nvidia Jetson Xavier NX

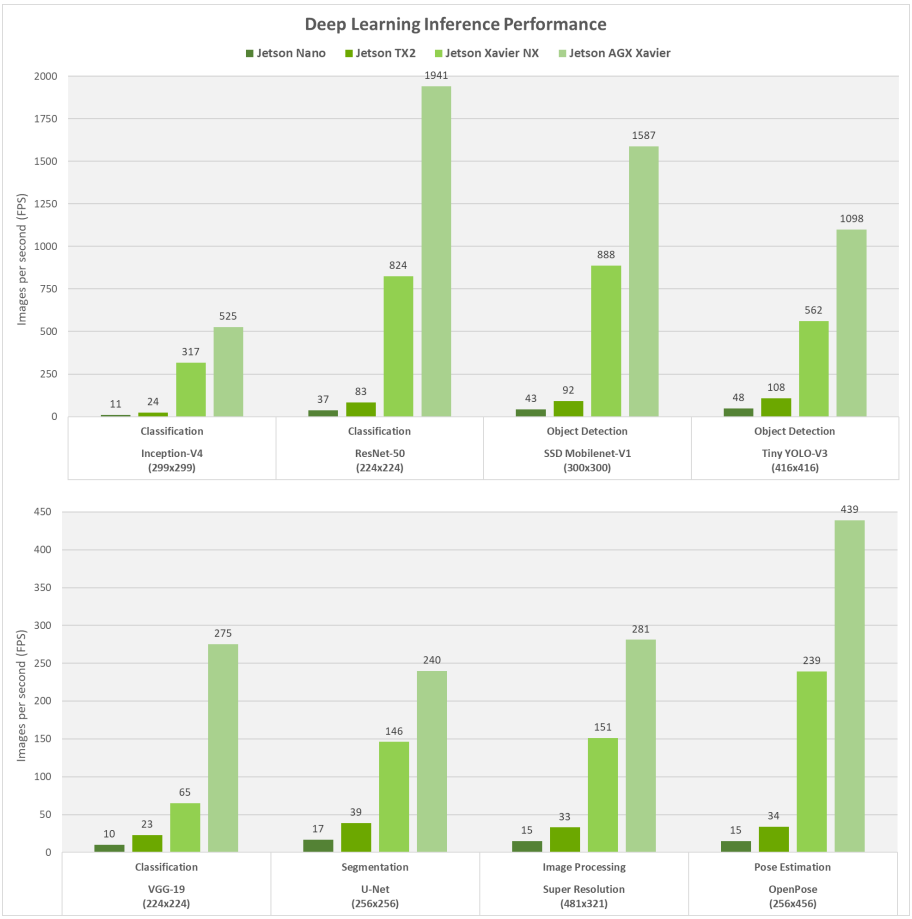

Figure 4.3: Performance comparison of Nvidia jetson cameras, from[24]

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

The only hardware sensor we have available is ZED stereo camera, shown in figure 4.4. It supports ROS.

Figure 4.4: ZED stereo camera.

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 [26] (Figure 4.5) or line follower robot (Figure 4.6), where inputs are mapped directly or tightly to outputs.

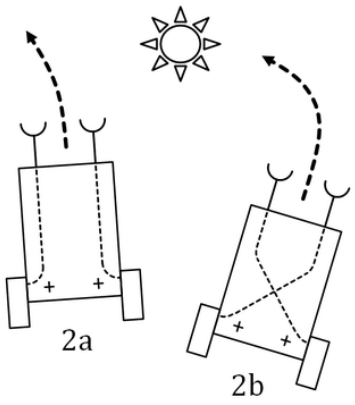

Figure 4.5: Braitenberg vehicle.

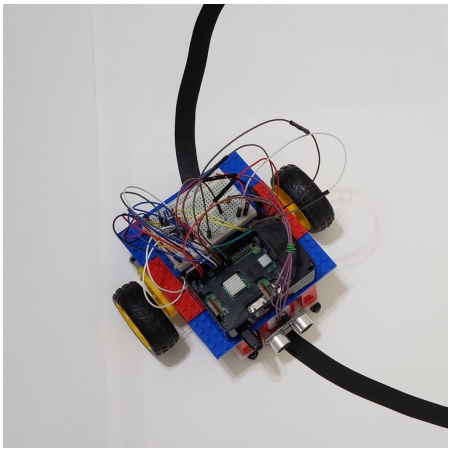

Figure 4.6: Line follower.

More complicated control paradigm is *SPA (Sense, Plane, Act)* implemented in robots such as Shakey [27], 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 4.7) 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.

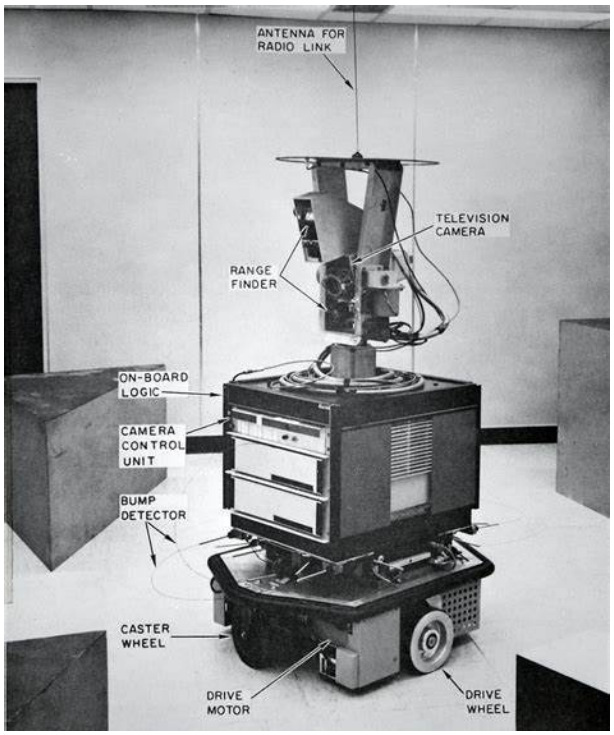

Figure 4.7: Shakey the robot.

Possible data flow from sensors, through planning to actions is shown in Figure 4.8.

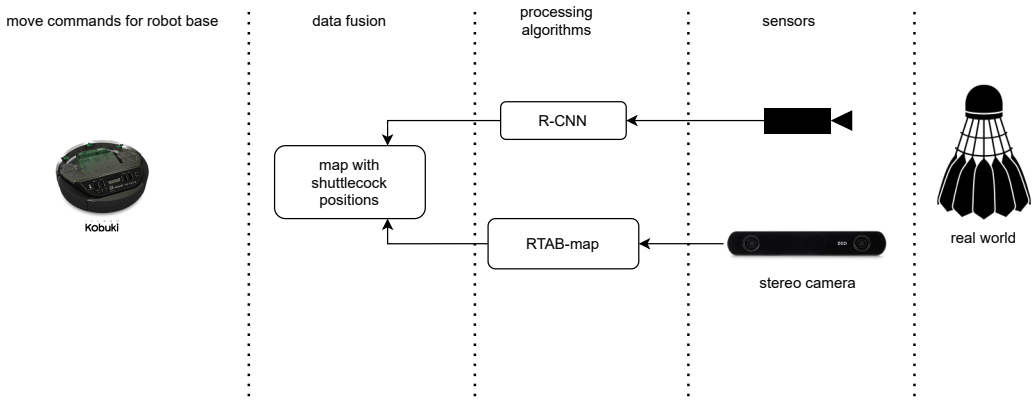

Figure 4.8: Example of data flow of possible robot in a SPA paradigm.

Next progression of robotic control systems was *Subsumption rachitecture* created by Rodney Brooks[28] 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[29].

Next followed architectures that combined reactivity and planning called *layered* or *hybrid* architectures, such as Firbys[30] *three layered* architecture (Figure 4.9) that was divided into *planning*, *executive*, *behavioral* layers.

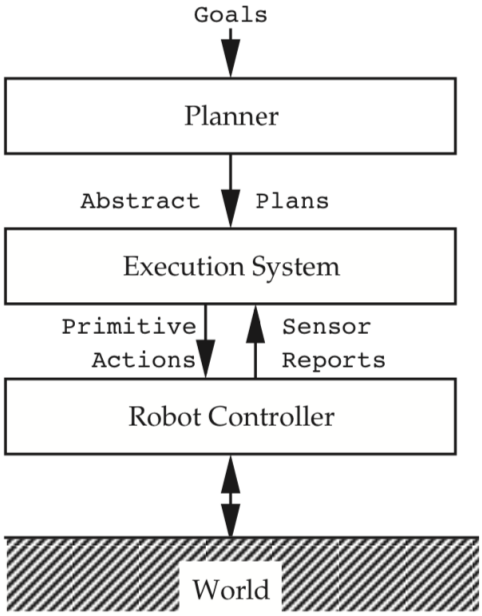

Figure 4.9: Three layered architecture according to Firby, from[30].

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[31]. In the 2010s there have beed numerous advances in computer vision using neural networks, such as R-CNN[32] (Figure 4.10).

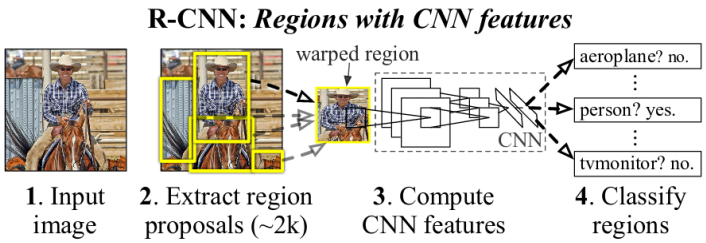

Figure 4.10: Overview of R-CNN architecture.

4.3.3 Mapping

Badminton courts are flat surfaces with marked lines. We could have two approaches to solving shuttle localization problem. **First**,we could have running localization and mapping at all times, and detect shuttles from point clouds created by mapping algorithm. This is very energy inneficient, 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 by 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 4.11). 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 we could use at many different places without any difficult and time consuming installation of cameras.

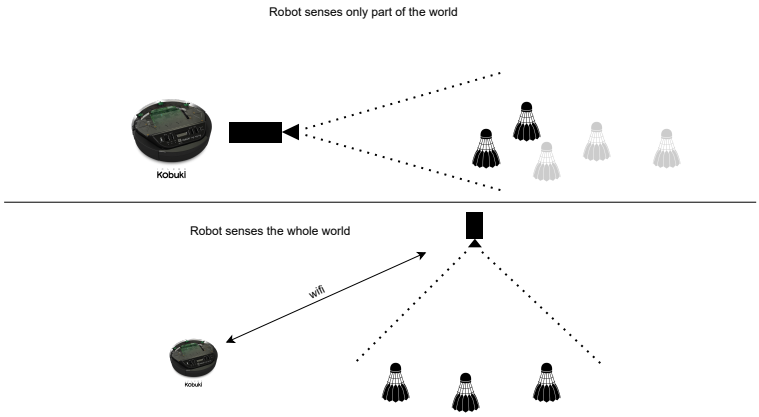

Figure 4.11: Partial vs whole view of the world.

4.3.5 Visualisation

We need to visualise *output* from the robot. Mainly for debugging and development. We need to see *images* from camera, *map* the robot creates, *objects* -shuttles the robot recognizes from environment, *path* or *plan* the robot generates to pick up the shuttles in some shortest metric such at *shortest time* or *shortest path* (we prefer shortest time).

Fortunately, all this is available in RViz- ROS visualization software. Some things we need to create ourselves, all this is described in **5.8.1**

4.3.6 User interface

We would like to visualise output from robot cameras, or where are the shuttles. For simulation we will uze RViz and for real operation of the robot, we need to setup RViz on remote laptop.

4.3.7 Picking

After we located shuttle and moved to it, we need to pick it up.

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.

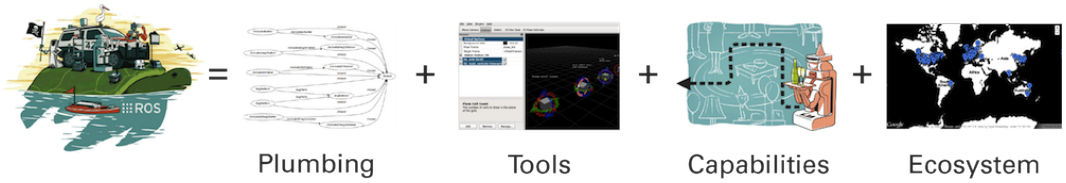

Figure 5.1: ROS equation

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 advantage 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. 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.

5.2 Gazebo

Gazebo [33] 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**[1] 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 [34], which has ROS integration via **gazebo_ros** [2] package. Gazebo is designed to be separate from ROS, and can be controlled programmaticaly using plugins[3]. Plugins are also used to generate sensor data for robots, and can be used to control the world[4].

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 [35],[36]. Possible state machine is shown in Figure Figure 5.2.

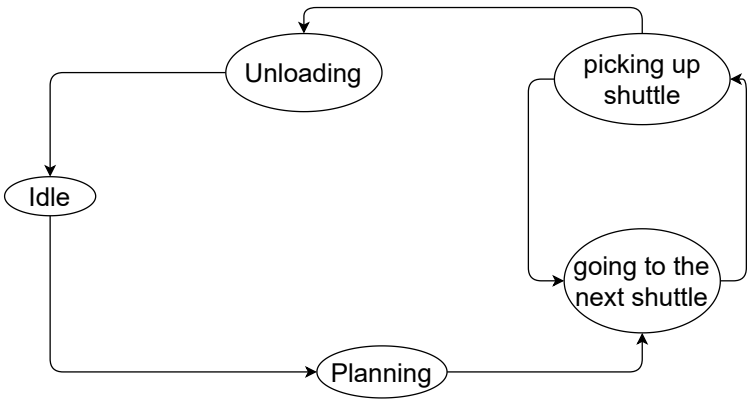

Figure 5.2: Example of possible state machine of the robot.

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

5.4 Mapping and localisation

For the robot localisation and mapping we chose open source ROS package RTAB-Map[5][37]. It is a graph-based SLAM approach[38]. It supports inputs from stereo and RGB-D cameras (Figure 5.3) and outputs pose and occupancy grid which we can use in navigation. First we need to drive robot around environment to get images from which RTAB-Map constructs 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, algorithm 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.

	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 [37].

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

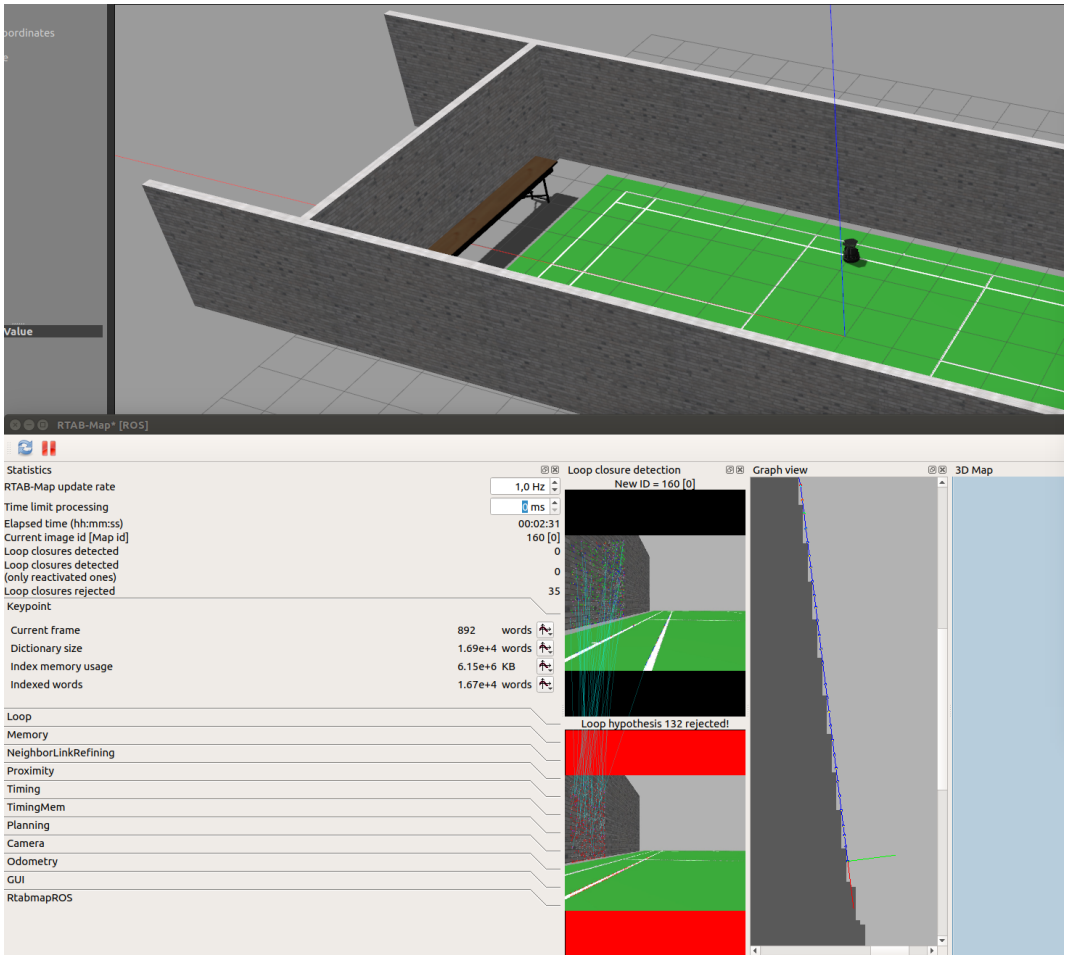

Figure 5.4: Loop closure detection in Rtabmap viewer

5.5 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 5.5). 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 5.5: Example of different court colors and materials.

5.5.1 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].

5.5.2 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* [39], i.e. in some modelling program, if done correctly, can be huge benefit to training algorithm (Figure 5.6). 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.

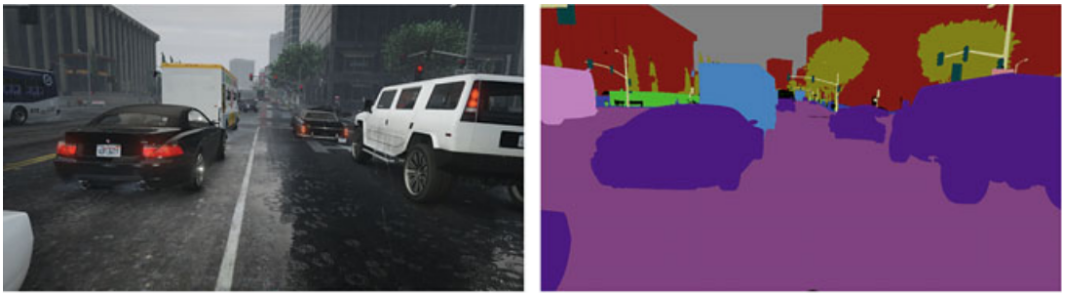

Figure 5.6: Example of synthetic dataset, from [39].

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

5.5.3 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.

5.6 Planning

Picking shuttlecocks is similar in nature to Traveling salesman problem where we have to visit n towns with least distance. There are some differences, for example robot takes time to turn, paths aren't straight lines, robot takes time to accelerate/decelerate. But it is useful abstraction.

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.

5.6.1 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.

5.7 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.

5.8 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 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 0.15m in

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

x-axis and 0.15m 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 5.7) or by **rotary mechanism** (Figure 5.8) as manual solutions mentioned in chapter 3. Arms are slow and would increase complexity of the system. Because of this, we chose simpler solution based on rotating brush which will pick up shuttle on the robot chassis.

Figure 5.7: PincherX 100 Robot Arm by Trossen robotics

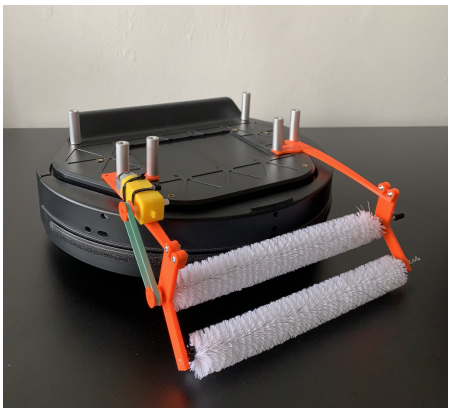

Figure 5.8: 3D printed prototype of picking mechanism

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 goals through clicking on map.

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

6. Implementation

In this chapter we describe implementation of proposed solution, 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.

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 Training data

We acquired data for neural network training by taking images of shuttlecocks in a different scenes. We tried to make images with different backgrounds and different objects so that neural network would generalize well. We trained SSD-mobilenet network using jetson-inference[1]. Dataset is split into training set, evaluation set, and test set. 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 6.1.

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

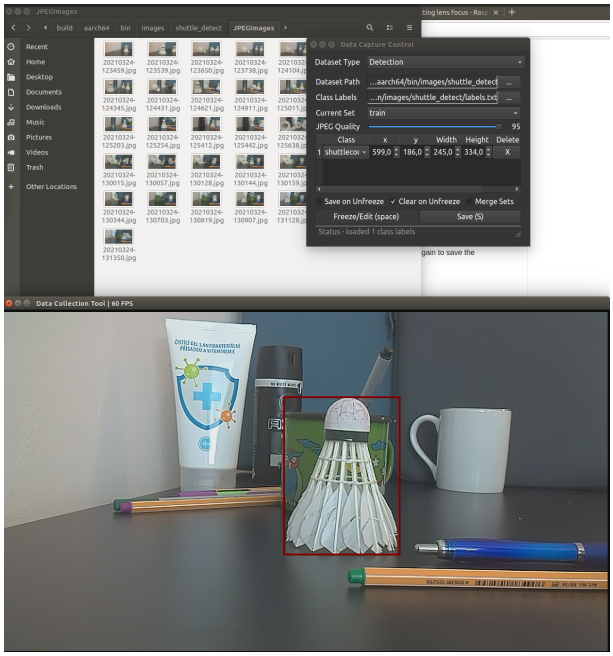

Figure 6.1: Creating dataset manually for object detection.

Shuttlecock detection working on images generated in Gazebo simulation (Figure 6.2).

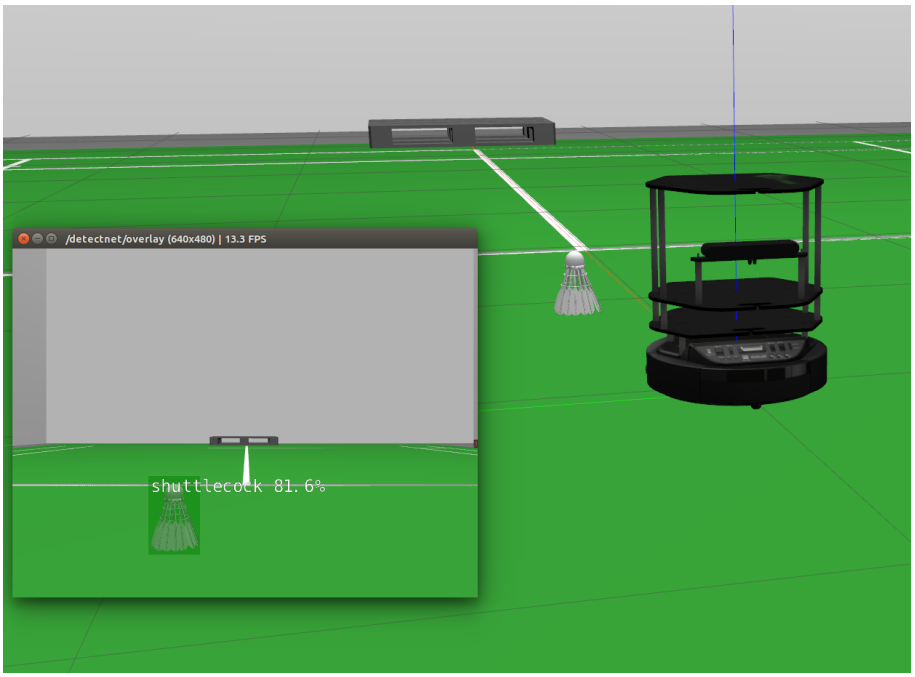

Figure 6.2: Shuttlecock detection in Gazebo

Working shuttlecock detection is shown in figure Figure 6.3.

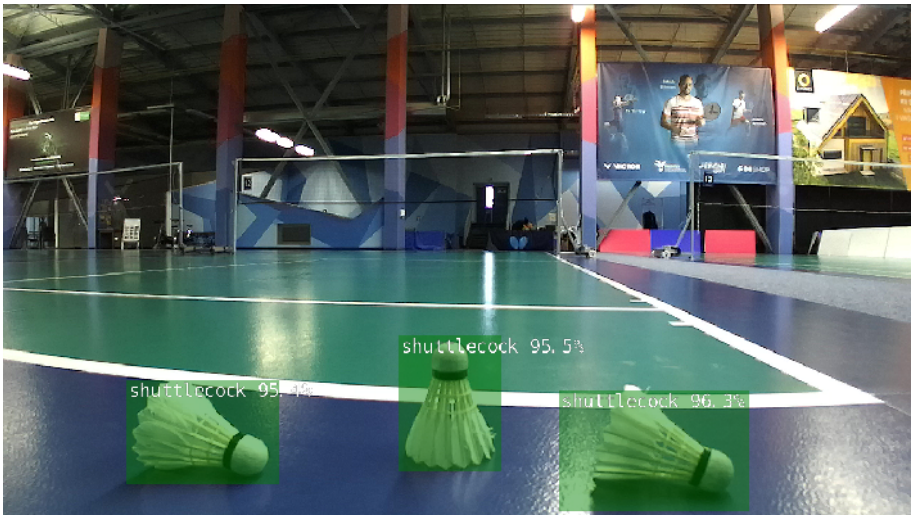

Figure 6.3: Multiple shuttlecocks detected on real badminton court

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 $sensor\_msgs :: PointCloud2$

Normally, every subscriber has its own callback function, but since we want to combine them, we use Synchronizer from *message_filters*[2] package. Since messages have different time arrivals, we combined them into one callback function using message filter Time Synchronizer[3] with *ApproximateTime*[4] 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 6.4), and compute average of the points using centroid[5] method of the pcl[6] library that we are using for manipulating with point clouds.

[2] http://wiki.ros.org/message_filters
[3] https://docs.ros.org/en/api/message_filters/html/c++/classmessage_filters_1_1TimeSynchronizer.html
[4] http://wiki.ros.org/message_filters/ApproximateTime
[5] https://pointclouds.org/documentation/classpcl_1_1_centroid_point.html
[6] https://pointclouds.org/

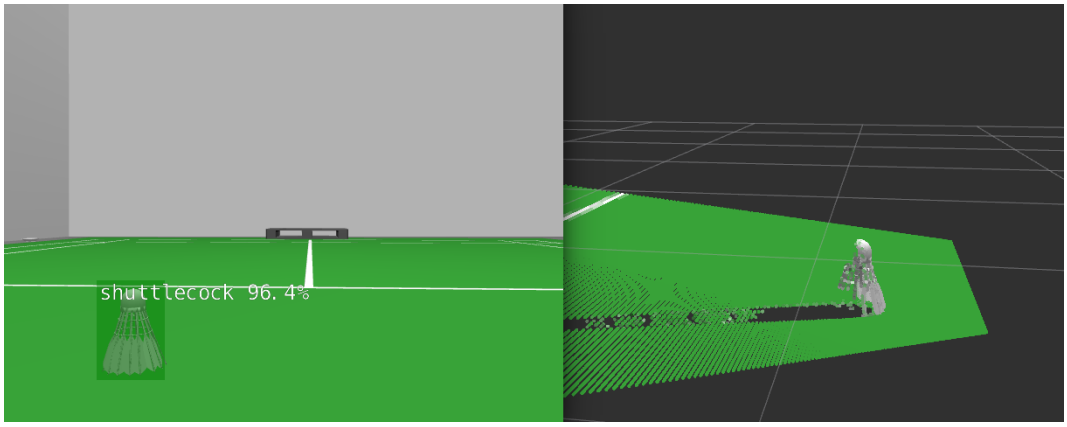

Figure 6.4: Neural network bounding box vs. point cloud. Some points inside bounding box are far behind shuttlecock

This gives us relatively accurate estimate of shuttlecock's position, shown in Figure 6.6, compared with Figure 6.5. 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 6.5: Shuttlecock in front of robot, inside Gazebo.

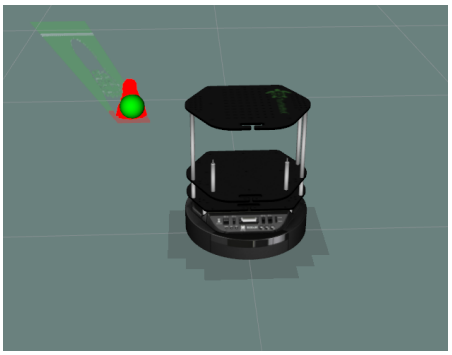

Figure 6.6: Estimated position of shuttlecock, from RViz

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[7] which are RViz visualisation objects.

6.5 Control system

Our control system is a state machine, with help of SMACH[8], a library for task-level execution and coordination in ROS.

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 [**smach**].

For picking one shuttle, we can design following state machine. It consists of states IDLE[9] (Shown in green in Figure 6.7) and COLLECTING and outcomes[10] *failed_picking* and *picked* (Shown in red in Figure 6.7).

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*.

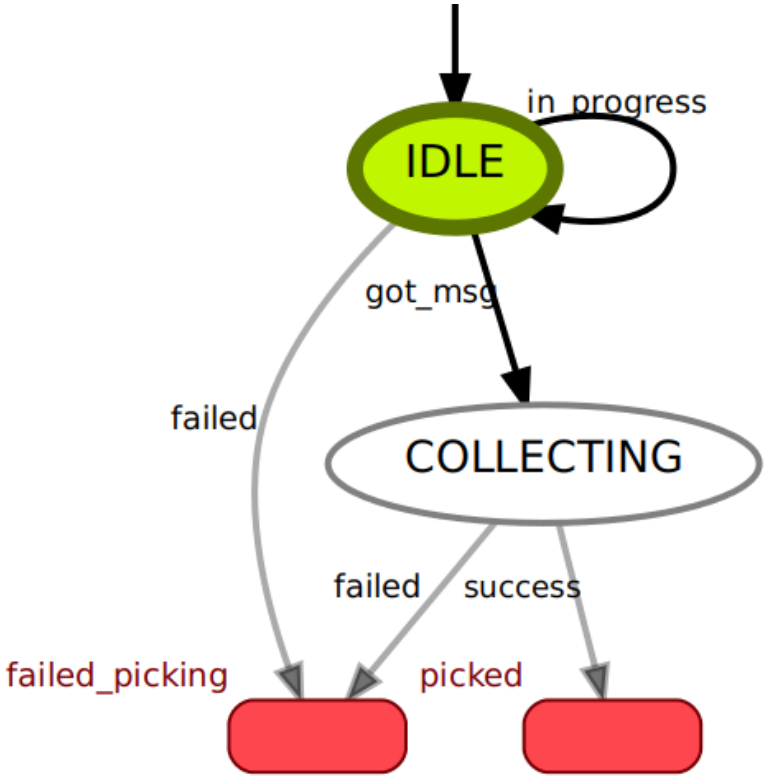

Figure 6.7: System is in first state, IDLE.

[7] http://docs.ros.org/en/api/visualization_msgs/html/msg/Marker.html

[8] http://wiki.ros.org/smach

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

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

In the state COLLECTING (Shown in green in Figure 6.8), (x,y) coordinate of shuttlecock is transformed into the move_base[11] 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.

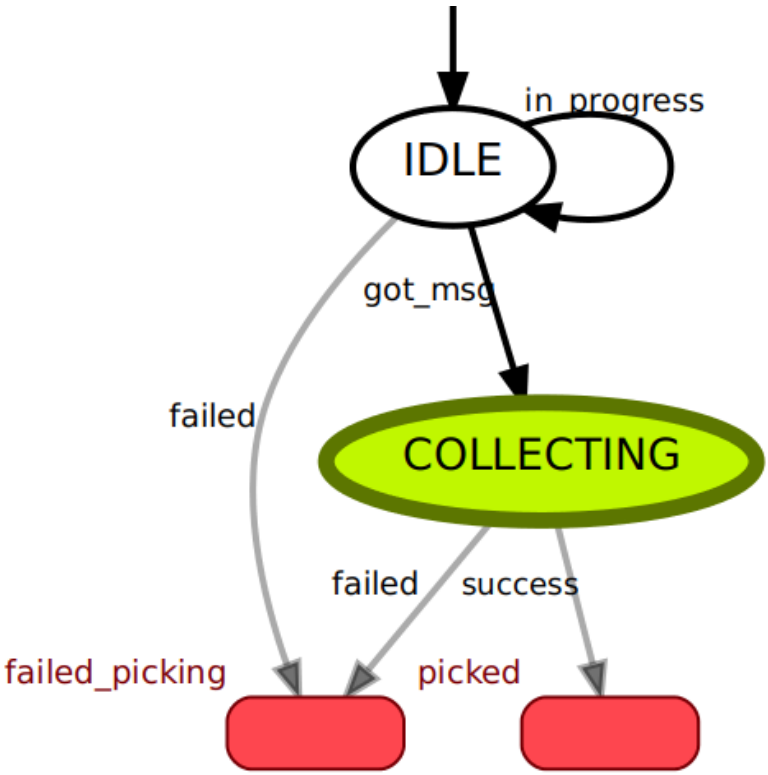

Figure 6.8: System is in second state, COLLECTING.

6.6 Gazebo simulation

For development and testing of our solution, we created multiple worlds for our simulation in the Gazebo simulator (Figure 6.9). 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 to simulate physics, and OGRE engine to draw 3D graphics. We already mentioned parts of gazebo in section 5.2. *World* is a XML file in SDF [40] 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>
```

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

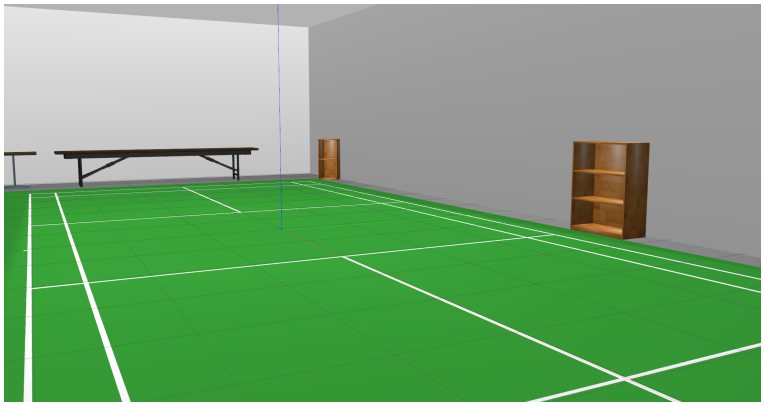

Figure 6.9: Gazebo world.

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 [41] format.

6.6.1 Gazebo Plugins

Gazebo plugins[12] 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*[13] for contact sensor[14]. 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 6.10) to Kobuki's model by modifying original URDF.

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

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

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

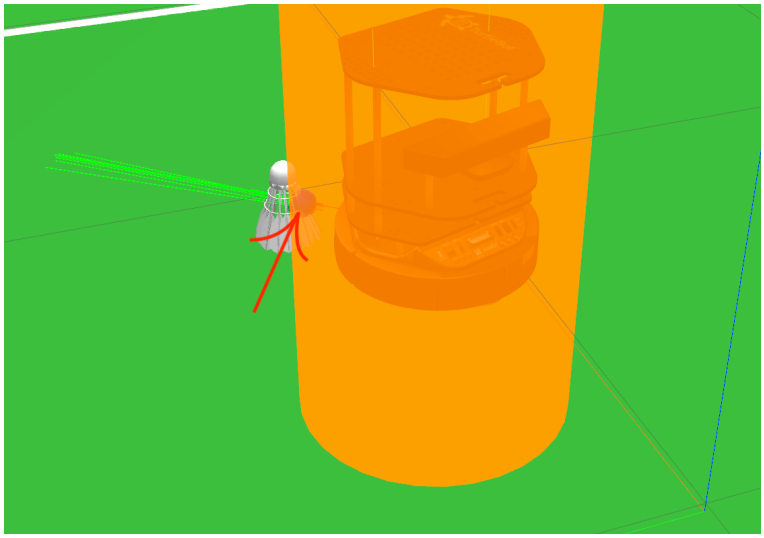

Figure 6.10: Contacts (pointed by arrow) detected between shuttle model and collision element (in Orange).

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*[15] on both models[16].

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

6.6.3 World plugin for controlling simulation

For evaluating performance of robot, automatic spawning of shuttles and following collection of shuttles in episodes is done by **world plugin**[17]. World plugin subscribes to the topic and processes GzString message in callback, using World[18] class to delete the model.

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

[16] Links to be precise

[17] http://osrf-distributions.s3.amazonaws.com/gazebo/api/9.0.0/classgazebo_1_1WorldPlugin.html

[18] https://osrf-distributions.s3.amazonaws.com/gazebo/api/dev/classgazebo_1_1physics_1_1World.html

6.7 Picking system

We developed a prototype of picking mechanism based on rotating brushes. We then printed it on 3D printer, and used small DC motor[19] to rotate brushes, show in Figures 6.11,6.12 and 6.13. Parts are screwed together using M3 bolts and nuts.

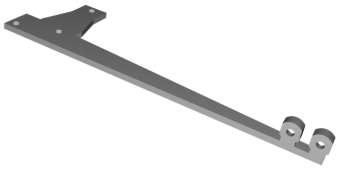

Figure 6.11: Long part extending forward

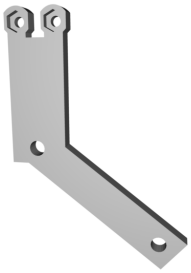

Figure 6.12: Side part, with holes for brush

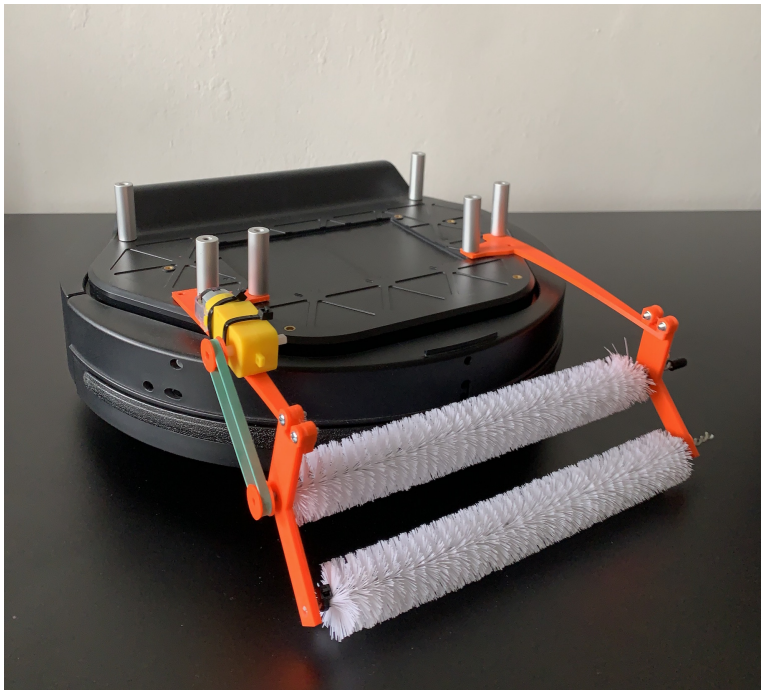

Figure 6.13: 3D printed prototype of picking mechanism.

[19] https://www.laskarduino.cz/tt-motor-s-prevodovkou-plastove-prevody/

7. Results

In this chapter, we evaluated our proposed solution. Robot should be able to recognize shuttlecock in both simulated and real world. We evaluate ability of robot to estimate 3D position of shuttlecock. There are many situations we could evaluate performance in simulation using Gazebo and then using real robot in real environments. In real environments robot could have difficulty recognizing shuttles because of different light conditions or because there are players occluding shuttles. Installation steps and user documentation for robot are in *Appendix A*, User documentation.

7.1 Evaluation of simulated shuttle picking

For evaluation of simulated shuttle picking we used Gazebo simulator. We designed badminton court as Gazebo world, mentioned in section 6.6. It consists of green floor with white lines and few models, such as bench, crates etc.

Our testing procedure consists of:

1. spawning robot in the middle of the court,
2. spawning a badminton shuttlecock,
3. launching control system, neural network, visual processing, localisation, and movement system

Visual system of out robot consisting of neural network recognized shuttlecock. Then the visual processing node fused depth information from depth camera and estimated position of shuttlecock in 3D space with respect to the robot. Then our control system sent command to planning system, which generated movement plan. Robot moved to the shuttlecock and picked it up.

7.1.1 Discussion

During testing and development, we encountered several issues.

We found out that performance of RTAB-Map varies with different environments and ability of algorithm to perform accurate matchings. If, for example, in the simulation, walls have periodic texture, as shown in Figure 7.1, the RTAB-Map will have a difficulty creating loop closures since lot of input images are very similar to each other, even when robot should be in different place according to the odometry from the robot base.

We then made world without repeating textures, using only simple coloured materials for walls and badminton court. This helped in a way, because algorithm no longer picked features in the middle of texture, but on the edges of court and lines, or edges between floor and wall. But we also encountered that robot had troubles with creating loop closures because court is symmetrical, without any features that would tell the robot he is visiting place he was before.

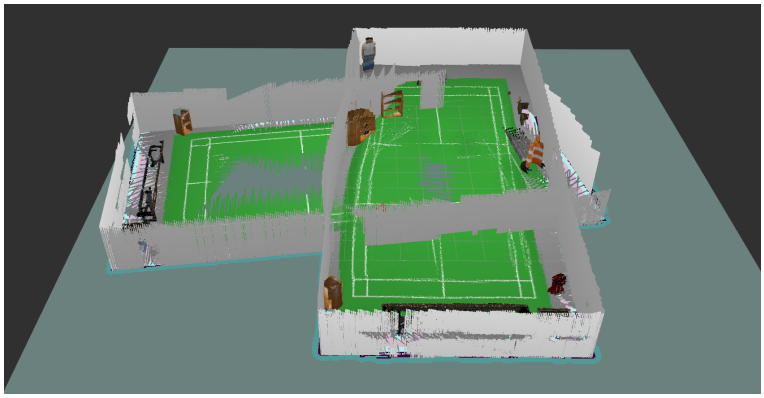

Figure 7.1: Wrong loop closure.

Next, we tried to mimic real world and put objects such as bench, boxes and others to create some not repeating visual cues for the robot. This helped, because RTAB-map then created loop closures when visited and viewed these locations.

Another issue was with Gazebo simulator, due our low testing hardware capabilities, we found out that after we spawned only small number of shuttles (10) the simulation time speed was 10 times slower than real time.

This could be alleviated by having faster computer, or tweaking physics parameters, such as **simpler collision meshes** of shuttlecocks, or less compute resources (limiting *max_contacts* of models or larger *max_step_size*[1]) available for physics engine.

7.2 Evaluation of shuttle picking in real world

We evaluated robot in real environments on different halls with different surfaces under various light conditions. We found out, that reflective floor surfaces such as badminton court poses problems for the ZED stereo camera (Figure 7.2). It puts points that lie on the floor below the floor.

[1] http://gazebosim.org/tutorials?tut=physics_params&cat=physics

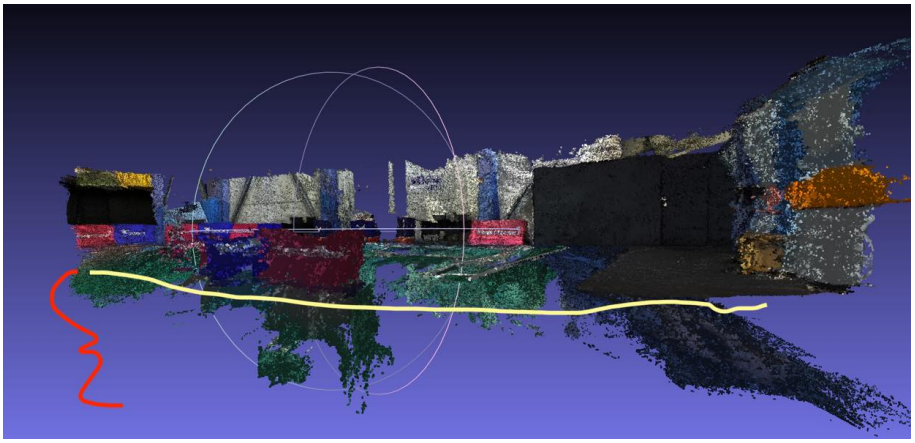

Figure 7.2: Exported pointcloud from the RTAB-Map, showing points that should lie on the floor are much lower, as marked by yellow line.

We took robot to real badminton courts. We created map of the environment by RTAB-Map. We then put several shuttles on the court and robot recognized them.

8. Conclusion

We have developed control system for the robot that pick shuttles scattered on badminton courts, by using ROS - Robot Operating System and designing our own control nodes.

Deep neural network was trained to recognise shuttlecocks from both simulated images and images from real life. To accomplish this, we created image dataset.

To estimate position of shuttlecock in 3D, we created program that integrates depth information from camera and neural network segmentation.

We integrated and tested multiple existing ROS packages such as RTAB-Map for creating maps of the environment and then then used these maps for robot localisation.

We used Turtlebot Kobuki robot as base of the platform and Nvidia Jetson Xavier NX for computation.

To test and develop our solution, we created virtual badminton court and simulated Kobuki robot in Gazebo simulator to test, debug and evaluate our solution and then used this solution with minimal changes on a real robot.

We designed a prototype of simple shuttlecock picking mechanism on a 3D printer.

Our approach was tested in simulation, and our robot recognized badminton shuttlecock using neural network, estimated its position in 3D world using depth information from camera, generated plan and autonomously picked it up.

Future work

It could be useful to test out other types of sensors such as RGB-D cameras that projects IR pattern such as Intel Realsense.

We could also benefit from faster mobile base, but it may be counterproductive because robots should not in any case damage floor surface with excessive weight.

Model training for shuttlecock detections could be trained on larger synthetic dataset to increase accuracy of recognition instead of small dataset made by hand.

Our solution could be adapted in other domains, such as picking of other objects. Because our solution is fairly general, only thing that would have to be modified would be training neural network to recognize objects in the other domain.

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/mens-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.

[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/.

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

[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.

[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/.

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

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

[28] 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.

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

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

[31] 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.

[32] 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.

[33] 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.

[34] (2020). Gazebo website, [Online]. Available: http://gazebosim.org/tutorials.

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

[36] 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.

[37] 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.

[38] 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.

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

[40] (2021). Sdformat (simulation description format), [Online]. Available: http://sdformat.org/.

[41] (Aug. 1, 2021). Collada dae format, [Online]. Available: https://docs.fileformat.com/3d/dae/.

[42] ROS. (2021). Ros installation, [Online]. Available: http://wiki.ros.org/melodic/Installation/Ubuntu.

[43] D. Franklin. (2021). Nvidia jetson inference, [Online]. Available: https://github.com/dusty-nv/jetson-inference.

[44] T. Fischer, W. Vollprecht, S. Traversaro, S. Yen, C. Herrero, and M. Milford, "A robostack tutorial: Using the robot operating system alongside the conda and jupyter data science ecosystems," *IEEE Robotics and Automation Magazine*, 2021. DOI: 10.1109/MRA.2021.3128367.

List of Figures

A. User documentation

This chapter describes hardware of the robot and installation of the software. We will describe our particular build, since the software is changing and we had to build packages that are already unmaintained.

In section Software we describe how to setup all necessary software.

A.1 Hardware

In this section we describe what hardware we used for robot base, what computer we used for running robot control system and other hardware such as camera and picking mechanism.

A.1.1 Robot base

Kobuki is implementation of Turtlebot 2 by Yuijin Robots[1].

Figure A.1: Kobuki platform

A.1.2 Computer

We are using Nvidia Jetson Xavier NX as robot's computer running all the software.

[1] http://kobuki.yujinrobot.com/about2/

Figure A.2: Nvidia Xavier NX

A.1.3 Camera

We are using ZED stereo camera.

Figure A.3: ZED camera

A.1.4 Shuttle picking mechanism

Picking mechanism is attached to robotic base.

A.2 Software

A.2.1 Jetson Xavier NX

To install Linux operating system with preinstalled Nvidia software, use steps described in https://developer.nvidia.com/embedded/learn/get-started-jetson-xavier-nx-devkit

A.2.2 ROS

Install ROS Melodic according to [42]

A.2.3 Jetson - inference

Next we install pretrained neural networks for Nvidia Jetson platform [43] from [2]

```
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.2.4 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 succesfully 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
```

A.2.5 Other dependencies

RTABmap mapping ROS package:

```
sudo apt install ros-melodic-rtabmap-ros
```

ZED stereocamera:

Download *ZED SDK for Jetpack 4.5* from https://download.stereolabs.com/zedsdk/3.5/jp45/jetsons.

```
cd path/to/download/folder
chmod +x ZED_SDK_Tegra_JP45_v3.5.0.run
./ZED_SDK_Tegra_JP45_v3.5.0.run
```

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

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

A.2.6 Source code

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

```
cd ~/catkin_ws/src
git clone https://github.com/martinerk0/shuttlebot.git
cd ..
```

Then use:

```
catkin_make --only-pkg-with-deps <package_name>
```

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

A.3 Usage in simulation

To start example world with control system run:

```
roslaunch shuttlebot_control scenario_01.launch
```

Then we need to tell RTAB-Map where is the robot.[3]

```
rostopic pub -1  /rtabmap/initialpose
 ↪  geometry_msgs/PoseWithCovarianceStamped '{header: {stamp: now,
 ↪  frame_id: "map"}, pose: {pose: {position: {x: 0.0, y: 0.0, z: 0.0},
 ↪  orientation: {x: 0.0, y: 0.0, z: 0.0, w: 1.0}}, covariance: [0.25,
 ↪  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.25, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 ↪  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 ↪  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.06853892326654787]}}'
```

Then we run control system:

```
rosrun shuttlebot_control control_02.py
```

A.4 Usage in reality

A.4.1 Start Kobuki

```
roslaunch kobuki_node minimal.launch --screen
```

A.4.2 Map the environment

```
roslaunch rtabmap_ros demo_turtlebot_mapping.launch
roslaunch rtabmap_ros demo_turtlebot_rviz.launch
```

Then drive with Kobuki around and try to map the environment.

```
roslaunch turtlebot_teleop keyboard_teleop.launch
```

Map will be saved for later use at:

```
~/.ros
```

[3] This is needed for convenience, because RTAB-Map saves map after each run, therefore if you run move robot then quit, robot would think it is at saved position and you would need to localize manually.

A.4.3 Usage

Launch necessary nodes:

```
roslaunch shuttlebot_control shuttlebot_reality_working.launch
```

Launch control system:

```
rosrun shuttlebot_control control_02.py
```

A.4.4 Remote operation

To use robot in real world situation, it needs to be operated remotely and checked upon. To do this, we set up Jetson Xavier to creates access point at startup to which we can connect with notebook. We can connect by ssh, but to open more shells and use GUI of the Ubuntu, we can also run vnc[4] server on Xavier and connect this way. Only disadvantage of this approach is that it does not support OpenGL, so we cannot visualise RViz this way.

To view robot's output using RViz , we have to run RViz on notebook and setup ROS networking.

Since we are using Mac, and ROS is not easily compatible with Mac, we used installation using conda[5] environments called Robostack[6] [44] where we used ROS Noetic. Then we set up few environment variables on client such as:

```
ROS_MASTER_URI=http://10.42.0.1:11311
ROS_HOSTNAME=10.42.0.245
```

which point to IP address of the robot that acts as ROS master node. These variables are also needed to be setup on host (robot). To check if everything works, we can use:

```
rostopic list
```

on client computer to see if the topics from robot are accessible through the network.

Another approach could be for Xavier and notebook join another network, but since this requires knowing password of the network in advance, and robot having large throughput of data messages to RViz resulting in latency of data, we think approach with creating access point on Xavier is better.

[4] https://www.digitalocean.com/community/tutorials/how-to-install-and-configure-vnc-on-ubu
[5] https://docs.conda.io/en/latest/
[6] https://robostack.github.io/GettingStarted.html

Attachments

The attachment contains 2 directories. Firstly, directory **shuttlebot**, which contains:

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

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