TALLINNA TEHNIKAÜLIKOOL
TALLINN UNIVERSITY OF TECHNOLOGY

TalTech Department of Electrical Power Engineering and Mechatronics

OBJECT DETECTION AND TRACKING WITH A MOBILE ROBOT IN ROS

Objekti tuvastamine ja jälgimine liikuva robotiga

MASTER’S THESIS

Student:	Semeniaka Yehor
Student code:	177246MAHM
Supervisor:	Mart Tamre, Professor
Supervisor:	Florent de Lamotte, Professor

Tallinn, 2019

(*On the reverse side of title page*)

AUTHOR'S DECLARATION

Hereby I declare, that I have written this thesis independently.
No academic degree has been applied for based on this material. All works, major viewpoints and data of the other authors used in this thesis have been referenced.

"......." .... 201.....

Author: ....
/signature /

Thesis is in accordance with terms and requirements

"......." .... 201....

Supervisor: ....
/signature/

Accepted for defence

".......".....................201... .

Chairman of theses defence commission: ....
/name and signature/

TTÜ Department of Electrical Power Engineering and Mechatronics

THESIS TASK

Student: Yehor Semeniaka 177246MAHM

Study programme: 02/13 MAHM – Mechatronics

Main speciality: Mechatronics

Supervisor(s): Professor Mart Tamre (TalTech), Professor Florent de Lamotte (UBS)

Consultants: Eric Senn, Lecturer (UBS)

Thesis topic:

(in English) OBJECT DETECTION AND TRACKING WITH A MOBILE ROBOT IN ROS

(in Estonian) Objekti tuvastamine ja jälgimine liikuva robotiga

Thesis main objectives:

1. Establish the communication of different devices in ROS, perform the Pioneer 3-DX motion controlling
2. Detect the object or the operator and calculate the position in front of the robot
3. Calculate the distance to the detected object and implement tracking of that object

Thesis tasks and time schedule:

No	Task description	Deadline
1.	Understand ROS and its principles	23.02
2.	Implement the detection and tracking algorithms	30.03
3.	Retrieve and use the data from depth sensor, test the hole system	27.04

Language: English **Deadline for submission of thesis:** "21"May 2019a

Student: ……………………. ....……............. "…....".................…201….a

/signature/

Supervisor: ………………… ……...........................… "…....".................…201….a

/signature/

Consultant: ……………… ……...............................… "…....".................…201….a

/signature/

Terms of thesis closed defence and/or restricted access conditions to be formulated on the reverse side

TABLE OF CONTENTS

PREFACE

The master thesis task was proposed by Professor Florent de Lamotte from the University of South Brittany. The primary idea of the task is to create the co-robot (cobot) type system which will work within the Robot Operating System (ROS) platform. The cobot system has to be able to detect the operator (this depends on the machine vision approach), track the operator and follow him. The knowledge from robot motion control and machine vision fields have been applied in that work. This system can be applied in different areas of human activity. It might be used to carry things inside the warehouse or to carry the stuff of older people. However, during the study, the task was slightly changed because the additional applications were discovered.

Most of the work was done during the mobility of the Erasmus exchange program at the University of South Brittany in France. I want to express my gratitude to the UBS for providing the accessories like robot and camera which were necessary for successful completion of the task. Also, I want to thank Professor Florent de Lamotte, Eric Senn and Professor Mart Tamre from Tallinn Technical University for inspiring and helping with the task.

List of abbreviation and symbols

ROS – Robotic Operating System

Cobot – or co-bot, is a robotic system or robot which is able to interact with a human being in a shared workspace

OpenCV – Open Source Computer Vision Library

YOLO – the detector based on the neural network approach, YOLO means 'You only look once'

RGB – the image encoding where each pixel of the image is represented as a tuple of three values: red, green, blue

BGR - the image encoding where each pixel of the image is represented as a tuple of three values: blue, green, red

FPS – the parameter which can be described as frames per second. This parameter reflects the speed of processing

DNN or dnn – Deep Neural Network

HSV – The image color model, H – hue, S – saturation, V – value

CNN - Convolution Neural Network

RGBD camera – RGB camera with Depth sensor

MIL - Multiple Instance Learning

KCF - Kernelized Correlation Filter

TLD - Tracking, Learning, and Detection

1. INTRODUCTION

The world is developing at a very high speed, and every day we are astonished by advances in technology. The robotics is one of the leading developing fields, and today, it is a popular trend to replace the simple manual work with robot work. Good examples of this are cooking robots, vacuum cleaning robots, automated grass-cutters and so on. This leads to the replacement of all machines that perform a simple work and require an operator with autonomous robots. The primary objective of this thesis is to create a cobot system which will be able to detect the operator, track and follow him. Object tracking robots in dynamic environment need to address a lot of tracking issues under various challenging situation. Ideally, the robot has to be able to detect the object under partial occlusion, changing in the distance between the robot and an object, and with different speed of the tracked object. The main application of that robots is to carry things inside to a warehouse, house or between departments in a factory or plant. With current detection and tracking algorithms, it is quite challenging to achieve perfect performance as a result. However, in the future, with the development of the robotics field and the invention of new machine vision approaches, the better solutions to this task will help not only to older people but also to avoid some costly mistakes associated with the human factor in production. This project closely related to Machine Vision part. Therefore, different detection and distance calculation methods will be considered. All operations have to be performed in ROS environment. Understanding of the ROS basics and ways how it can be used is another goal of the project.

An autonomous vehicle with its detection and tracking systems is quite a popular subject nowadays, and many car manufacturers are trying to implement it. The topic of this thesis is closely overlapped with that. Autonomous or semi-autonomous robots which are able to move by themselves or follow a human might be very useful in construction or military applications. For now, it can be argued that it is not difficult for a robot to perform such a simple task like moving. The main issue is to develop a detection algorithm which will provide stable and robust detection. In this thesis; how to use ROS for the robot motion control, several detections and tracking algorithms, and the ways of estimating the distance to the detected object are described. Besides a quite old object detection approach based on the color detection method, the state-of-the-art YOLO detector is described in the machine vision chapter. Two methods have been performed to define the distance to the object. The quality of the output data from these methods has been analyzed, and based on this analysis, the better algorithm has been selected. Depending on the output from machine vision part and the data which are produced by the robot itself, the robot moves forward or stay still, turn left or turn right, in other words, track the object. Additionally, equipment like laser radar are used, to get more information about the area around the robot and generate a map. This map could be subsequently used to improve the project by recognizing the path of the robot. The analysis of the equipment has been conducted, and the accessories' specifications are provided as well. The ideas for future development of the work are also described.

During this project, the Pioneer 3-DX robot, RGB – Depth camera, ODROID XU4 board, and RPLIDAR A2 version are used. The descriptions and specifications of the equipment are presented in the relevant chapter. Pros and cons of each part are discussed, and comparative analysis has been conducted. As the primary programming language, Python is used. Almost all libraries support Python programming language, and it is not difficult to find a lot of helpful literature or information on the Internet since it is one of the most popular programming languages. Another reason for using Python was that ROS supports Python or C++. The tools from OpenCV library version 4.0.0 was used for object detection and tracking.

The main body of the thesis has six chapters with their sections and items. The third chapter is dedicated to the equipment which have been used. Next chapter of the paper is related the ROS. In that chapter, the basics of the ROS are described: nodes, topics, and data exchanging inside the system. The crucial part of understanding the ROS is related to the communication between nodes. The subscriber/publisher communication paradigm is described in that division. At the end of the chapter; RosAria, which is the module for motion control, its topics, and the data these topics produce are discussed. The fifth chapter is related to the machine vision. It includes the description and comparison of different detection and tracking approaches. Methods which stand for distance estimation are described in the last section. The penultimate chapter is dedicated to the laser radar and how it could be used for further development of the project.

2. BACKGROUND AND LITERATURE OVERVIEW

Mobile robot systems are used in various fields of human activities such as rescue, military, medicine, and autonomous navigation. One key enabler of such applications was the development of powerful sensors such as color cameras and lasers. These advanced sensors, in conjunction with each other, provide a rich source of information for the robot. It allows to create divorce object recognition and tracking systems and makes the Machine Vision applications applicable for different purposes. Generally, all detection approaches can be divided into two groups. The first one is the use of the pre-trained neural network for object recognition. The second one is the use of the different image pre-processing methods to distinguish the desired object among the rest.

Two very useful approaches from the second group are object detection based on the color detection and image segmentation. A study was conducted by M. Tarokh and P. Ferrari in their work 'Case Study: Robotic Person Following Using Fuzzy Control and Image Segmentation'. (1) They applied an iterative thresholding method to automatically select threshold values. By using these values, they removed all objects in the image which were different from person's shirt color. Since it is possible to have objects that have colors similar to the desired object in the image, they conducted an image segmentation by using standard region growing method. Then the regions which has the closest shape to the person's shape was chosen. The detection method looks robust enough according to their results. However, they calculate the distance to the person from a parameter, which is called the mass of the image. The description of the parameter is that the program summarizes all pixels in the binary image after all processing operations. White pixel means 1 and black pixels means 0 in the binary image. The sum value reflects the distance to the object. The bigger value of the parameter means the closer distance to the detected object. Estimating the distance to the person with that method may be not so accurate and using depth sensor can be a better solution. Additionally, authors used a fuzzy logic controller for the robot motion control, which as well as a PID controller, is a good solution for that task.

Another interesting approach is to use a depth image or stereo image to detect an operator. This approach was described in 'Development of a Personal Following Robot and Its Experimental Evaluation' paper. (2) It stands for person detection by using depth templates. The authors defined three templates of the person in depth image corresponding to three body directions. The templates are presented in the figure 2.1. The background in the images is white to emphasize a silhouette. In the original image from the depth sensor, the background will be darker depends on the distance to the background objects. However, it will be enough to distinguish the person.

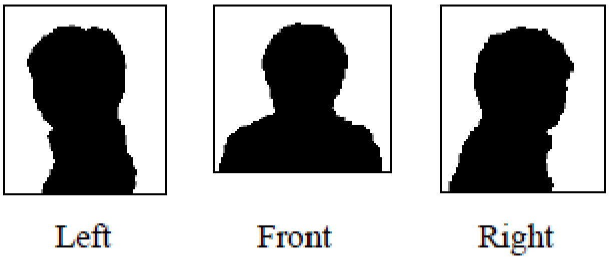


Figure 2.1 Depth templates of the human being from three viewpoints (2)

These templates are used simultaneously to detect the person. This approach is very similar to a simple pattern matching and therefore, can produce a lot of false detections for objects with a similar silhouette to the person. In order to avoid false detections, the developers trained SVM-based classifier. In their work, each detected region is resized to 20 x 20 pixels, and a set of pixel values is directly used as an input to the classifier. All templates have been made with a person who was about 2 meters away from the camera and that might be a problem. The person's silhouette may differ from the template when the distance is longer or shorter than 2 meters. The pattern matching cannot detect the person in that case. However, based on the paper results, the success tracking rate was about 95%, which is very high. Like in the previous example, any controller can be used to implement a motion control.

Another method for object detection is based on the use of HSV color space. The HSV color space is more suitable for object detection compared to RGB because the color is described with only one value. Compared with the RGB model, that color representation fits more to how human describe the color. This method can be called HS histogram based color extraction method. (3) The program distinguishes the object by its color. For example, the operator can be detected by the yellow vest. The main issue of that algorithm is that the color is changing over time because of the different light conditions. Moreover, usually, the color is not exactly the same on the left and right side of the vest. The color can be different depending on the camera angle as well. So that method requires defining the threshold boundaries of the desired color. Then the binary conversion is required in order to detect a contour. The binary conversion has to be done in a way that the pixels inside boundaries will be white and the pixels outside will be black. Afterwards, additional filter can be applied. For example, the specific object can be found by using its personal detail, such a shape or size. Another way is to distinguish the object by using nested colors. On the yellow vest can be some blue symbols which can be also detected. Using these features help to detect the desired object among the rest objects in front of the camera.

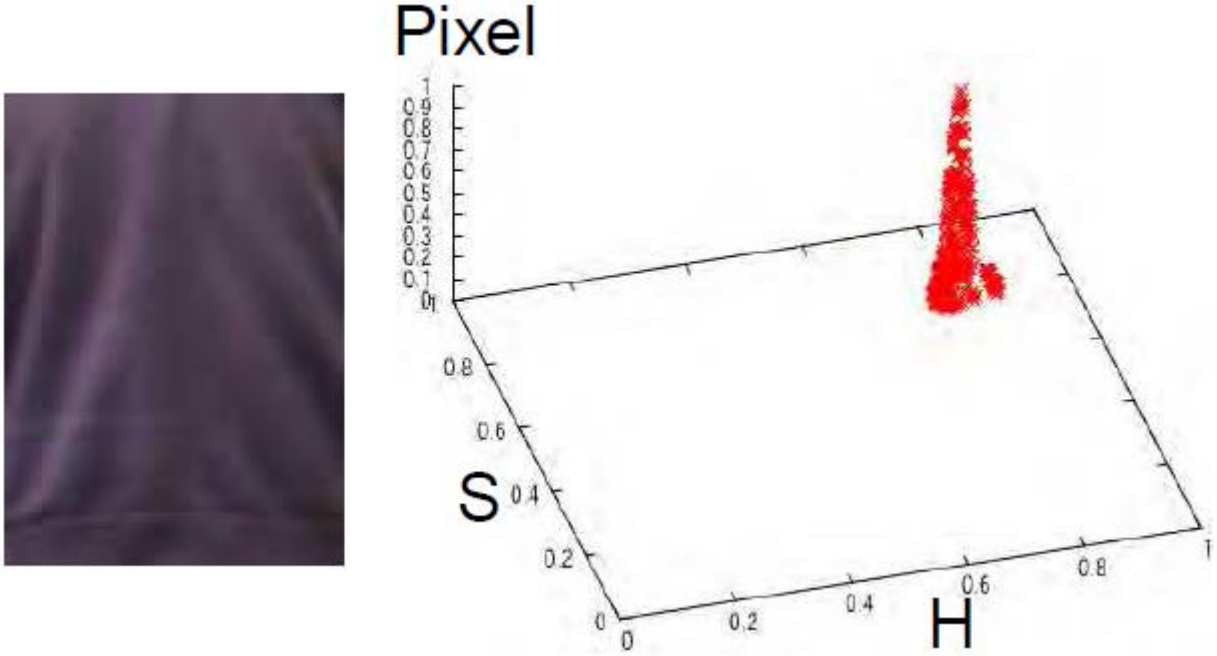


Figure 2.2 The histogram model on the right side of the image represents the colors on the left side (3) H – hue, S – saturation (3)

This method provides robust and stable color detection and can be combined with other methods to achieve a better performance. Even a combination of two separate color detectors can be applied. High sensitiveness to the changing in light conditions is the main drawback of that method.

The second group of object detection algorithms related to the neural networks. The newest and the most powerful detector is a YOLO detector. (4) This state-of-the-art approach provides object detection with a very high quality and fast processing speed. The detector was created by Darknet and there are several YOLO detectors available on the Darknet official webpage. All of them are trained to detect the most common objects. However, it is possible to train the detector for specific objects if it is needed. The way how YOLO detector can be used will be described in the relevant chapter. There are a lot of other detection approaches based on the neural networks except YOLO detector. It is possible to use a CNN, which stands for Convolution Neural Network for person detection. This approach was described in the 'Integrating Stereo Vision with a CNN Tracker for a Person-Following Robot' paper written by Bao Xin Chen, Raghavender Sahdev, and John K. Tsotsos. (5) They applied a CNN model in conjunction with RGBD camera to achieve good and stable detection of the operator. They developed three different CNN models. In order to use CNN to track a person, they firstly initialize the model. After network initialization from scratch with random weights coefficients, they trained this network online. The person must stand in a pre-defined boundary box which is placed in the center of the frame. Once the network is activated, the patch in the boundary box is considered as a positive image, and the patches outside the boundary box are considered as negative templates. The CNN starts the training process by using that data set.

During the work the net updates itself by using a set of positive patches from previous frames. The most recent 50 positive patches are retained from the previous frames to form the positive patch pool. Developers used PID to control the robot.

The ROS official documentation (6) and OpenCV official documentation (7) were the main sources while performing the work. A lot of information related to the equipment and devices have been taken from the manufacturers and distributors web pages.

3. ACCESSORIES

This chapter is dedicated to the accessories which have been used during the work. The division consists of the description of each accessory and explanations about why that device has been used. It also includes the comparative analysis for some devices with explanations of their advantages and disadvantages. In the thesis work Pioneer 3-DX robot, Odroid XU-4, Orbbec Astra camera, and RPLIDAR A2 version have been used. All devices were provided by the University of South Brittany. The Pioneer 3-DX robot, which was used as a testing robot, has a basic configuration with two front and one rear wheel. The robot also has eight sonars which are embedded into the front bumper, three accumulators and a lot of different sensors which provides information like motor state and power state. The robot is also equipped with a sound piezo buzzer. The Astra camera was used in Machine Vision for object detection. It has a depth sensor which was required in the project. Odroid board with Ubuntu 16.04 operational system serves as a PC to do all program calculation and to control the robot. The mapping is performed by using the RPLIDAR, which is mounted on the top of the robot.

3.1 Odroid XU-4

The Odroid XU-4 is a powerful and low-cost single board computer which can be used in various applications such as Arduino or Raspberry Pi. Since the board has a small size and can be placed almost everywhere, it is very popular for web browsing, computing and software development. The board can be also used for home automation. The Odroid XU-4 is equipped with octa-core Exynos 5422 big.LITTLE processor, advanced Mali GPU, and Gigabit Ethernet which makes the device cheap and fast enough for Mechatronics purposes. The board is an ARM device, which is the most widely used architecture for mobile devices and embedded 32-bit computers. (8) The ARM processor architecture has its pros and cons. The advantages are the size, speed and excellent performance as it was mentioned before. However, the main disadvantage which was faced during the work is that some advances libraries such as opencv-contrib are not adapted to that type of architecture and require a PC.

The operating systems available for Odroid board are Ubuntu, Debian, Android, Fedora, ARCHLinux, and OpenELEC. All these operating systems have thousands of open-source libraries and applications which makes it easy to use, modify or update if it is necessary to fit your specific needs.

The greatest benefit of the board is that the power consumption is between 10 W and 20 W. It is much lower compared to the standard PC thereby in this case the Pioneer robots' accumulators easily supplied the board. The device contains the same connection ports as typical computer, with 1 USB 2.0 port, 2 USB 3.0 ports, Ethernet port which supports Gigabit transfer speed and the HDMI port to which the user can plug the monitor. In additional to these standard inputs, the board includes slot for microSD and 40-pin GPIO port.

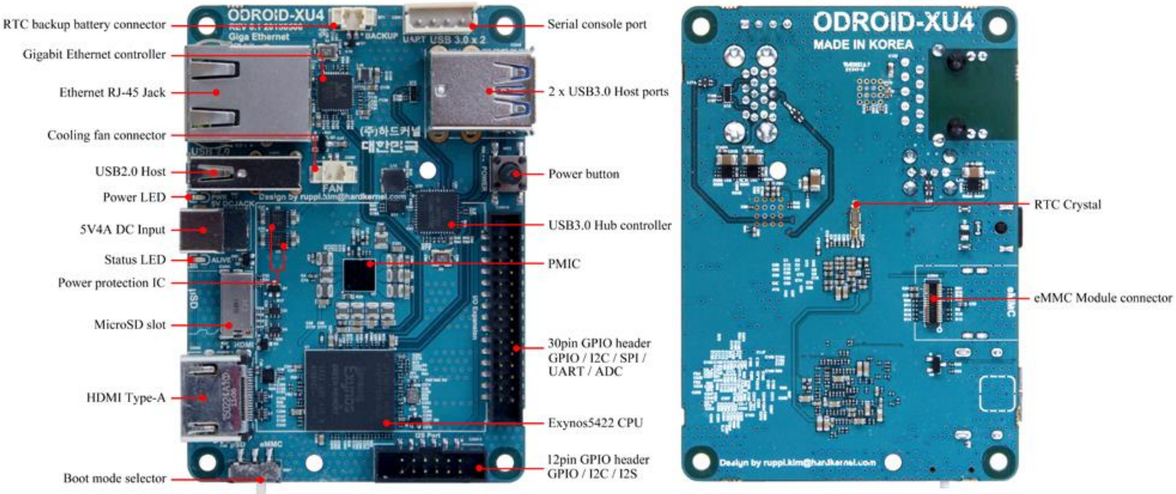


Figure 3.1 Annotated Odroid XU-4 board image (9)

In the comparative analysis with more popular Raspberry PI 3 board, two significant advantages were found. The high-performance 2GB DD3 RAM is an advantage that allows compiling the code much faster on Odroid XU-4 than on Raspberry PI 3. Another advantage is the availability of the USB 3.0 port, which increase the data transfer speed greatly.

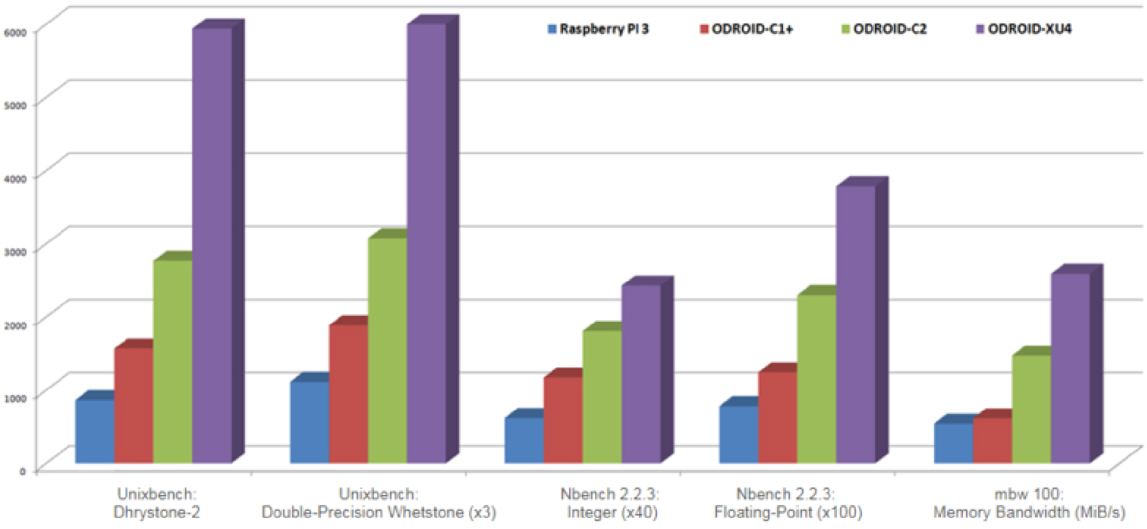


Figure 3.2 Comparative analysis of single board computers. Several benchmarks were conducted to measure the computing power of the boards. The values of the test results were scaled uniformly for comparison purpose (8)

3.2 Pioneer 3-DX robot

The robot which was selected for the master thesis was proposed by Professor Florent de Lamotte. Pioneer 3-DX robot has a simple configuration. It was explicitly designed for the research proposes. The aluminum body makes Pioneer robot light and rugged, capable of dealing with different obstacles on his way. The robot benefits from a complete ROS, which allows the engineer to use different packages dedicated to the robot motion control and data acquisition from embedded sensors. ROS also allows to use additional accessories and combine them with a robot, writing programs which combine various sensors and perform the data analyzing. Hence the robot is popular for doing the research projects with autonomous navigation, manipulating and mapping for remote-controlled applications. (10) As it was mentioned Pioneer 3-DX robot has a simple configuration. It includes microcontroller with ARCOS firmware, eight sonars, two front and one rear wheel, wheel encoders, piezo buzzer, three batteries. They all are built in the body of the robot. Inside or on the top of the robot there is free space available. The developer can mount additional devices like camera, PC or single board PC, additional sonars or other sensors there. The autonomy time is 8-10 hours with three fully charged batteries.

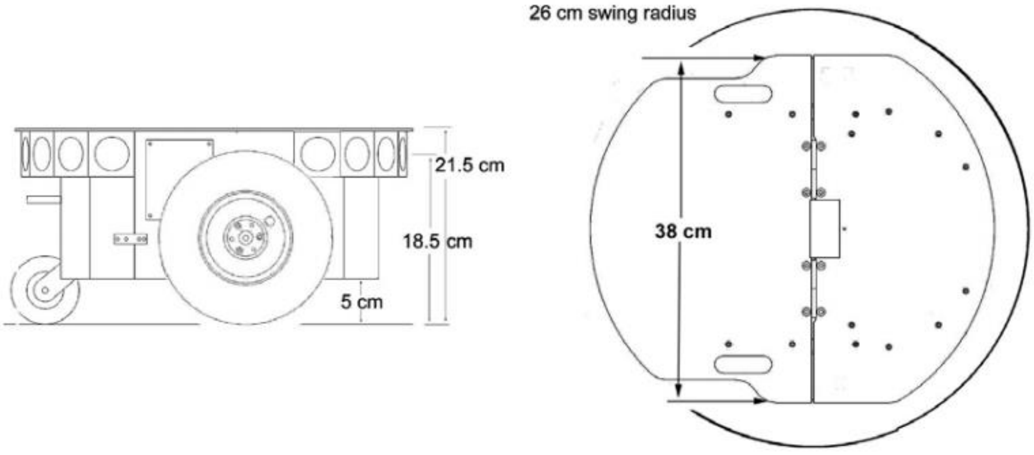
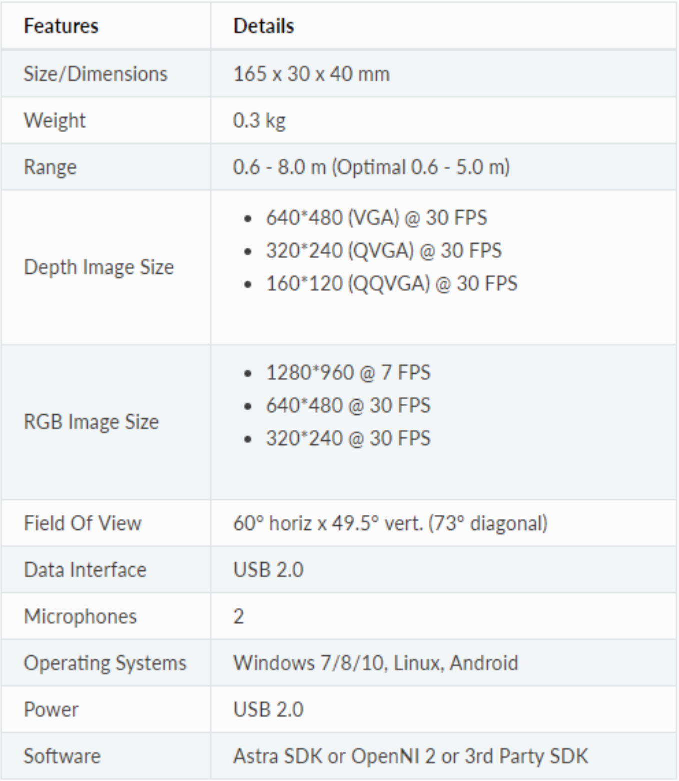


Figure 3.3 Pioneer 3-DX's physical dimension and swing radius (10)

The core applications for Pioneer robot generation is ARIA framework. ARIA is necessary to control the robot and receive the information from all sensors. This framework is Linux and Windows compatible. In the thesis, the adapted to ROS version of ARIA was used. Pioneer 3-DX has maximum linear speed 1,2 m/sec and maximum rotational speed 300^0/sec. (10) This robot appears to be a good one for implementing the tasks of the project.

3.3 Orbbec Astra camera

The significant part of the work is related to the Machine Vision, therefore camera selection was the essential part of the work. Initially, the idea was to do all the work with a simple RGB camera. As it was mentioned before Pioneer 3-DX has eight embedded front sonars, so knowing the object position in front of the robot, it is possible to estimate the distance by using the data from sonars. However, during the work it was found that the camera with a depth sensor suits the project completely and it makes the work a little bit more challenging and interesting. Orbbect Astra 3D seria is an excellent device for a wide range of scenarios, including gesture control, robotics, 3D scanning, and point cloud development. The good thing is that the camera manufacturer has already created the package for ROS system and maintain it on the GitHub. It is effortless to launch the camera and get the RGB or depth information in different formats by using astra_ros package. For instance, the depth information can be obtained from a depth image or from a point cloud. The same is also valid for a color image. The developer can get a BGR image or grayscale image from specific topics of that package. Astra depth sensor has the range from 0,6 to 8 meters, but the optimal maximum range for high performance is up to 5 meters. This range is quite enough for the projects successful implementation. The camera resolution is 640x480 pixel with 30 fps. (11) Full camera specification can be found on the figure below.



Features	Details
Size/Dimensions	165 x 30 x 40 mm
Weight	0.3 kg
Range	0.6 - 8.0 m (Optimal 0.6 - 5.0 m)
Depth Image Size	• 640*480 (VGA) @ 30 FPS • 320*240 (QVGA) @ 30 FPS • 160*120 (QQVGA) @ 30 FPS
RGB Image Size	• 1280*960 @ 7 FPS • 640*480 @ 30 FPS • 320*240 @ 30 FPS
Field Of View	60° horiz x 49.5° vert. (73° diagonal)
Data Interface	USB 2.0
Microphones	2
Operating Systems	Windows 7/8/10, Linux, Android
Power	USB 2.0
Software	Astra SDK or OpenNI 2 or 3rd Party SDK

Figure 3.4 Orbbec Astra camera specification (12)

Additionally, it can be stated that since Astra camera is widely used in robotics projects, a huge community of users has been formed on the Internet. It is possible to find the answers for almost all camera related issues there.

3.4 RPLIDAR A2 Laser Range Scanner

The other accessory which was used in the project is an RPLIDAR. RPLIDAR A2 is a laser range scanner manufactured by Slamtec company. It is a device created for quick and accurate mapping with a sample rate up to 8000 times, which is the highest in the current economical products in LIDAR market. It runs clockwise to perform 360^0 laser range scanning. With ROS already developed by the manufacturer package, the lidar produces the map of its surrounding environment after scanning. By using a self-developed brushless motor, it runs smoothly without any noise. The maximum range of the lidar is 18 meters which is good enough to use it for robotics related projects. (13) It can be implemented in many mechatronics projects, in conjunction with machine vision applications. The lidar has a serial port connection, but the USB adapter goes with it in the kit. This adapter creates a bridge from serial to USB connection which allows to connect the lidar to ODROID board easily. In the project, RPLIDAR was mounted on the top of the plastic box which serves the protection body for ODROID XU-4. The figure of the robot is presented below.

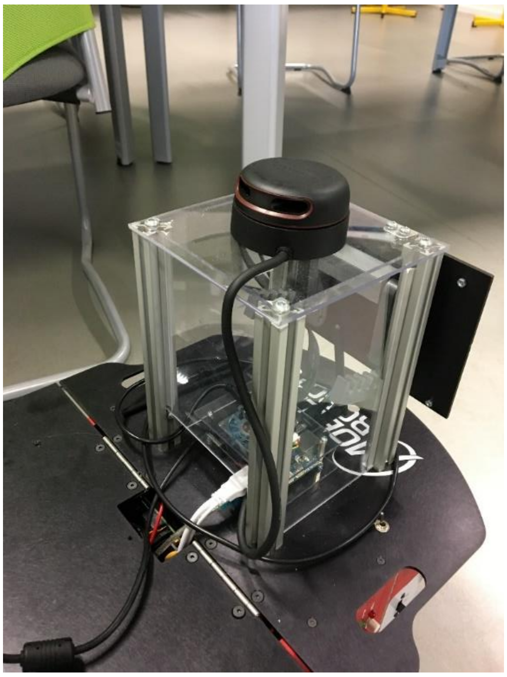


Figure 3.5 RPLIDAR mounted on the Pioneer 3-DX robot

4. ROBOT OPERATING SYSTEM

This chapter is dedicated to ROS. (6) ROS was created by Willow Garage company for writing and creating robot software. It has a big collection of tools and libraries to simplify the task of creating a complex and robust robot behavior. Nowadays ROS is a collaborative robotics software development, and this is a massive benefit of ROS. There are lot of different engineering laboratories all around the world, some of them are focus on mapping and robot autonomous motion others are focused on machine vision and robot tracking. All of them conduct researches and increase global ROS library every day with new, more sophisticated packages. Open source packages can be easily installed on Linux with a command 'apt install'. After installing the package, it can be immediately used with 'roslaunch pkg_name file_name' command. The only thing which is required to launch before is a roscore. Roscore is a collection of nodes and programs that are pre-requisites of a ROS system. Nodes in ROS are basically executables that represent programs. Each node can require some input data and can produce some output data, roscore delivers the data between the nodes and maintains the whole system. In simple words it could be explained like the nodes are the stations, the messages are the trains, and roscore is railway lines. Without Roscore it is not possible to launch the node, receive or send the message. All messages in the ROS have their formats. There is a standard message library 'std_msgs' where the user can find all basic message formats like float, integer, string. Another embedded in ROS important message library is 'sensor_msgs'. It contains the messages which is used to send more sophisticated information from sensors such as images. The user can also create his custom message if it is needed. For instance, it is possible to create a message with three integer values. The most popular communication model in ROS is a publisher/subscriber communication paradigm which is using the messages. Beside the message data transfer, the services and actions are available in ROS.

The publisher/subscriber is a very flexible communication method. This method has been used in the project and will be described later. However, sometimes the task requires the request/reply communication model. This model is implemented via ROS service which is defined by a pair of messages: one is for request, and the other one is for the reply. Two nodes are required for that type of communication, one node is a client which calls the service, and the other one is a server which provides some functions or conducts some operations. After completing a task, the server sends a reply to the client. The principle of the service communication model is that while the client calls the service, it will stop and wait for the reply. It means that the client node will not proceed the code until the server node finished the request. Service consists of a request and reply parts and as a regular message, can be created by the user. The user can check all available services with a command ‘rossrv list’. As well as with messages the user can check the structure of the service with ‘rossrv show srv_name’.

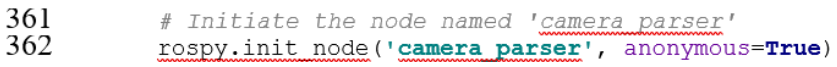
The last communication scheme is the action. In some cases, the service takes a long time to execute the operations, and the user wants to perform some other operations in that time, or he wants to track the status of the execution. Action provides the ability to create a server that perform long-time goals with continuous feedback. The message, in that case, consist of three part: goal, feedback, and result. As it was mentioned the advantage of the action is that the client node can execute some certain operations while receiving specific feedback from the server node. When the server node sends the result, the client node can go on and start to execute another block of code. (14)

With very convenient and various communication paradigms, ROS is a powerful development toolset. These tools support debugging, plotting and visualization of the robot states or outputs from sensors and cameras. The most popular embedded ROS tools are rviz and rqt. The first one serves for the visualization of almost all data types in ROS. Rviz can visualize laser scans, three-dimension point clouds, and camera images. This tool also has its own GUI where the developer can choose the sensor or camera from the list. Visualization of all available data from the robot sensors looks impressive and allows the user to go quickly through this data for debugging. Another powerful ROS embedded tool is rqt. Rqt is a Qt-based framework for developing a graphical interface for the robot. It consists of several different plugins, one of them is rqt_graph which provides real-time information about a ROS system. By using rqt_graph, the user can check active nodes, created topics and the communication between nodes. Communication between nodes means the subscribers and publishers which are presented in the graph. This tool is the most powerful one for debugging because the user can easily understand which nodes are currently active and which of them is sending or receiving the information. With rqt_plot plugin the developer able to monitor encoders, voltage or anything that can be represented as a number that varies over time. In the master thesis rqt_graph is used to describe the ROS system.

Another benefit of ROS is the launch file. These files have their own structure. They are used to launch several nodes at the same time. Also, the launch files are used to pass specific parameters more conveniently or to specify the output option for the node. The principle is similar to passing the arguments through the terminal in Linux. In order to launch the node, the developer has to specify the package to which that file belongs, the name on the file, the name of the node created in that file and the output option. Usually, the output option is always a ‘screen’ which means that the output like rospy.loginfo() function will be displayed in the terminal from which the launch file is launching.

4.1 Subscriber/Publisher communication paradigm

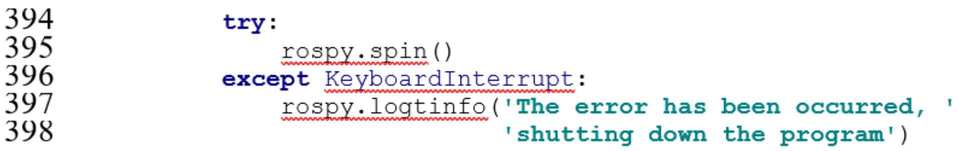
In this section, the communication scheme which was used in the thesis is described. The node should be initiated before writing an executable code and establishing the communication. The developer can create a node by using the next function:



```
# Initiate the node named 'camera_parser'
rospy.init_node('camera_parser', anonymous=True)
```

Figure 4.1 Initiating a node with ‘camera_parser’ name

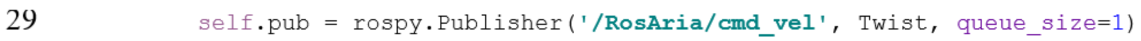
In the code above ‘camera_parser’ is the name of the node. The second argument ensures that the node will have a unique name by adding random numbers to the end of the name. This argument should be used because each node in ROS has to have a unique name. If the developer is sure that this name is not used, then he can omit second argument and specify only the name of the node. After the node has been created the developer has to keep the node on by using while loop in python or the next command:



```
try:
    rospy.spin()
except KeyboardInterrupt:
    rospy.logtinfo('The error has been occurred, '
                   'shutting down the program')
```

Figure 4.2 rospy.spin() function is used to keep the node on. It can be replaced with simple while loop

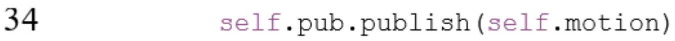
Subscriber/Publisher is the simplest and the most popular communication paradigm. This method was used in the thesis work for the communication between Robot, Camera, and RPLIDAR. The scheme of data transfer is quite simple and lies on the fact that each node which produces the output data creates an object called the publisher. The publisher can be created by using the next code:



```
self.pub = rospy.Publisher('/RosAria/cmd_vel', Twist, queue_size=1)
```

Figure 4.3 This part of code is used to create a publisher to /RosAria/cmd_vel topic which will publish Twist message

Here '/RosAria/cmd_vel' is a name of the topic to which the pub will publish the data. The second argument is a message type which will be published and 'queue_size = 1' is an argument which limits the number of queued messages. The queue is needed in the cases when subscriber is not receiving the messages fast enough. Since the publisher was created, it creates a topic automatically where the information will be published, and the other nodes can use a subscriber to get these data whenever the information is updated. To publish a message, following code should be used:



```
self.pub.publish(self.motion)
```

Figure 4.4 This code is used to publish the message

The method 'publish' requires only one argument which is the message. The message has to match the format which was specified when the publisher was created. In case the developer wants to publish the information with a specific rate, a sleep time has to be added to the while loop.

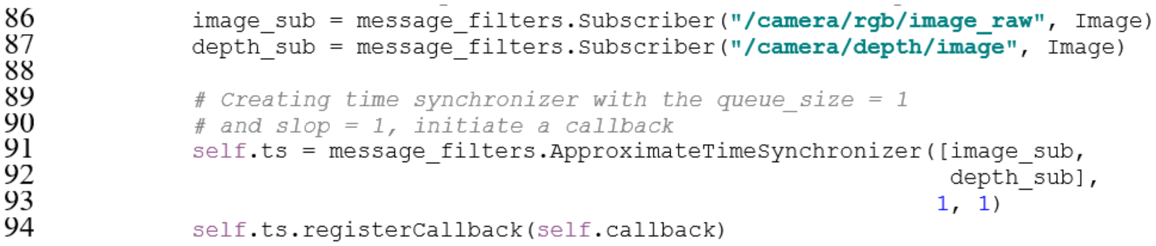
Since the publisher was created and the node starts to publish the information to the topic, in another node the subscriber has to be created to read that information. The subscriber can be created with the next line of code:

```
rospy.Subscriber("chatter", String, callback)
```

In the code above the Subscriber method of rospy library takes three arguments: the name of the subscribed topic, the message type and the name of the function which will be launched in case the topic gets updated. The callback function takes only one argument, and it is the data from the topic. The developer can work with that data inside the callback function or store it into a global variable or class variable and work with it in another block of code. The interesting point is that the subscriber does not require a loop. A separate branch is created for each subscriber, and whenever the topic is updated the subscriber automatically runs the callback function. At the same time, the node can publish the data to several topics. It also can be subscribed to several topics. In case the task requires the data from several topics which have to be synchronized with the same timestamp, the message_filters package has to be used. (15) This package will be described in the next section.

4.2 Message_filters package

The package is devoted to take in messages from subscribed topics and output them at a later time. An example is a time synchronizer which was used in this project. It takes messages of different types from multiple sources and outputs them only if the filter has received messages with the same timestamp. In this project, the program gets the messages from the depth sensor and RGB camera. These messages must be synchronized to calculate the depth value of the pixel where the object is. This is required because the node which producing depth image has smaller publishing frequency compare to the node which producing RGB image. To use the message synchronizer the developer needs to create subscribers without a third argument which usually defines a callback function. This can be done by using the same message_filters package (line 86,87). Then these subscribers have to be specified as an argument to 'ApproximateTimeSynchronizer' function (line 91):



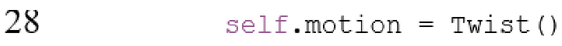
```
86      image_sub = message_filters.Subscriber("/camera/rgb/image_raw", Image)
87      depth_sub = message_filters.Subscriber("/camera/depth/image", Image)
88
89      # Creating time synchronizer with the queue_size = 1
90      # and slop = 1, initiate a callback
91      self.ts = message_filters.ApproximateTimeSynchronizer([image_sub,
92                                                             depth_sub],
93                                                            1, 1)
94      self.ts.registerCallback(self.callback)
```

Figure 4.6 Describes how to use message_filter package to synchronize several subscribers

The first argument of 'ApproximateTimeSynchronizer' function is a list where the developer specifies all subscribers which he wants to synchronize. In the code above 'image_sub' and 'depth_sub' are subscribers to RGB and depth topics respectively. The Second argument specifies how many sets of messages it should store from each input while waiting for a message to arrive. The last argument is a slop parameter which defines the delay in seconds with which messages can be synchronized. After the message filter has been created the general callback should be launched (line 94). In the code above the callback function is created which takes synchronized data from the depth sensor and RGB image. The data is subsequently analyzed and processed.

4.3 RosAria

RosAria is an adapted version of Aria for ROS. (16) This node communicates with Pioneer 3-DX robot. It means that by using RosAria the developer able to send motion commands to the robot and receive the information about battery voltage or state of charge. RosAria is publishing the information to eight topics and subscribed to only one topic which is /RosAria/cmd_vel. Cmd_vel is a topic intended for robot motion control. It receives a Twist message type which consists of linear and angular parts. In order to use ROS messages in Python or C++ files, they have to be imported separately in the same way it is done for modules. If the message, such as the Twist message, belongs to the geometry_msgs package, it should be imported from geometry_msgs.msg. After the message has been imported, an object of that type of message can be created.

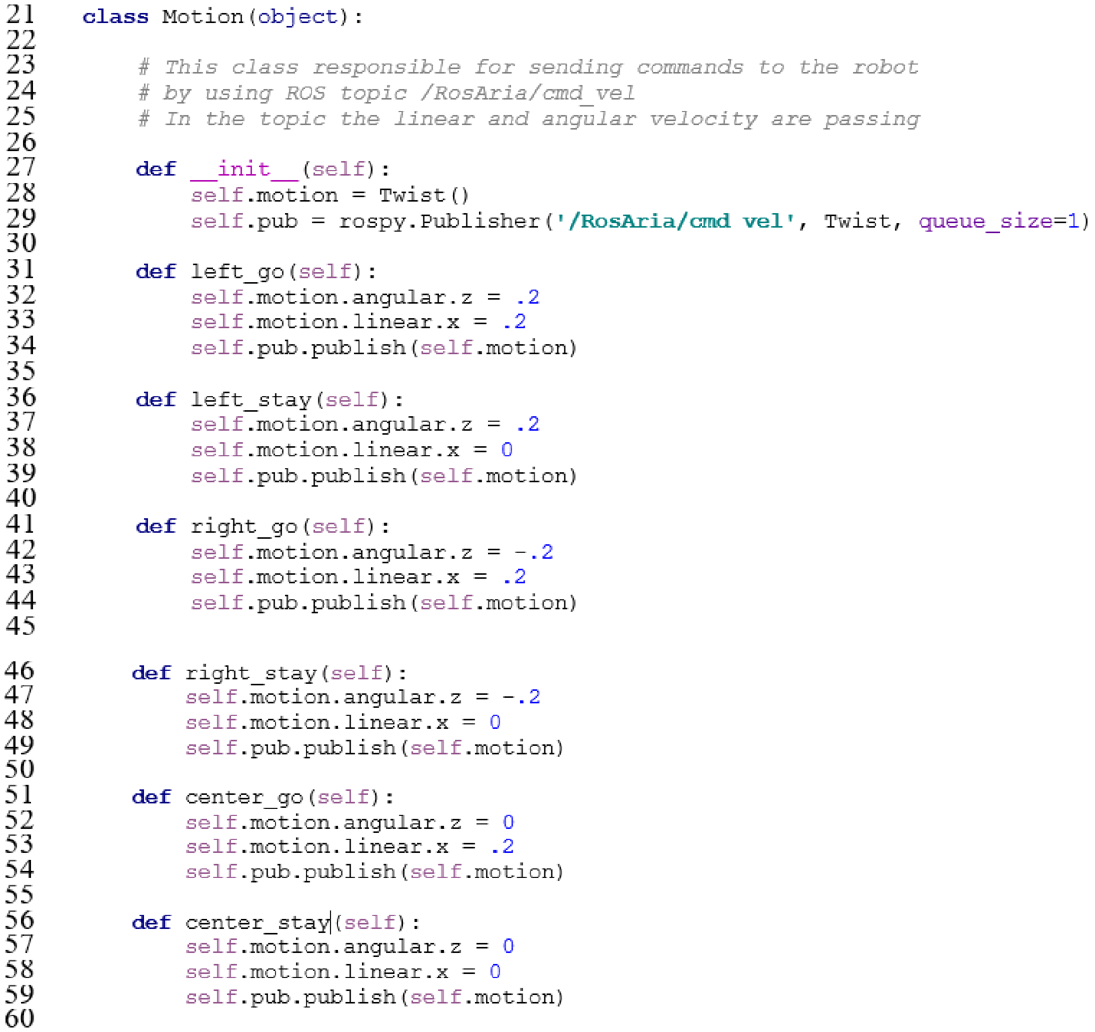


```
            self.motion = Twist()
```

Figure 4.7 Creating a Twist message

Positive number in the linear component leads to forward motion and a negative number leads to reverse motion. A positive number in the angular component leads to counter-clockwise rotation and otherwise. The operator can control the robot even from the terminal. For this 'rostopic pub topic_name message_type' can be used. At the same time RosAria publishes information to eight topics. These topics will be presented in the 3.4 section by using rqt_graph tool. These topics are: pose (position of the robot), bumper_state, sonar (publishes sonar readings, Pioneer 3-DX has eight sonars), sonat_pointcloud2 (information from the same sonars but in Point Cloud data structure), battery_state_of_charge, battery_voltage, battery_recharge_state, motor_state. In case the robot has more sophisticated configuration additional topics can be created. The sonars are turned on only if any node subscribed to the sonar's topic. When the last subscriber unsubscribes from both sonar's topics they will be turned off. This is done to save battery. RosAria provides also two services which are enable_motors and disable_motors. These services can be useful when the developer is writing a code for an autonomous robot. In that case, one can write a block for emergency stop and subsequent re-enabling of the motors. By using this service, the robot will not enable the motors until some condition is met. Thses conditions, for example, can be related to the detection part or can be even interconnected with the data from RPLIDAR.

One of the aims of the thesis is robot motion control. It means that the robot should move depending on the outputs from Machine Vision part. For this, after getting the results of the detection and selecting the proper command based on the detection results, the command has to be converted to the ROS geometry_msgs/Twist message type and published to the /RosAria/cmd_vel topic. The geometry_msgs/Twist message type expresses velocity in free space separated into its linear and angular parts. Each of these two parts are represented as a vector with three elements. In this project, we are interested in a z component of the linear part and z component of the angular part. The linear and angular velocity were set to 0.2. This value can be changed in the code. An additional option is to make a function that will increase the linear velocity proportionally to the depth distance. The same can be done for angular speed. However, it should be mentioned that the robot has inertia. Therefore, increasing the velocity will lead to more significant inertia, which in turn will decrease the robustness of the detection. The next Python class (line 21) was created to control the robot by sending the specific command to the /RosAria/cmd_vel topic in ROS.



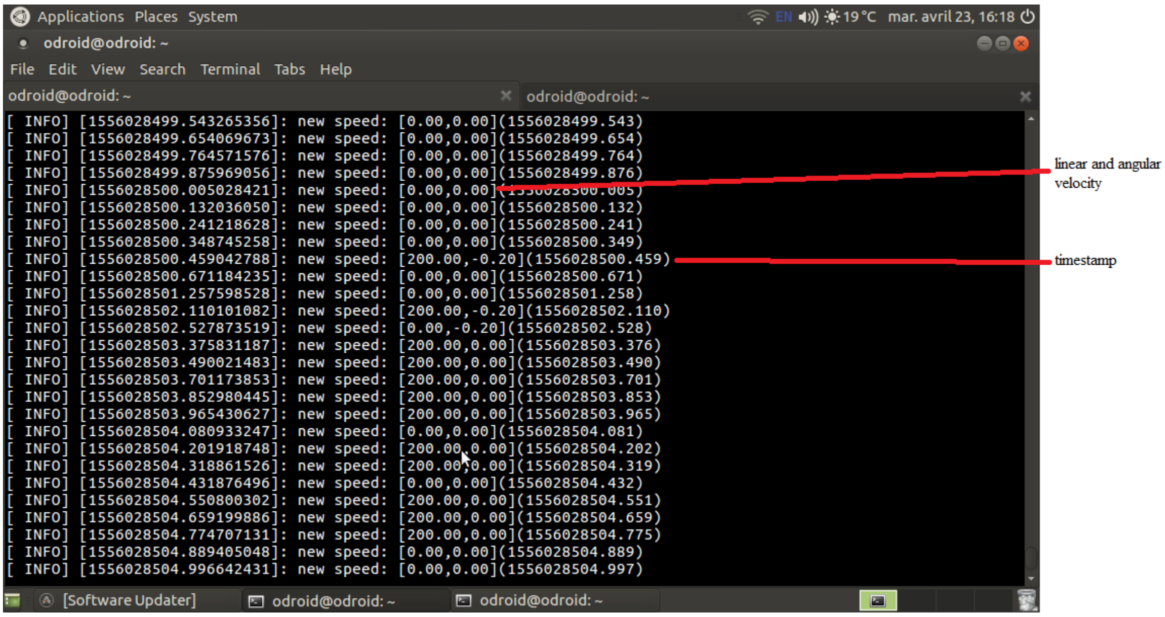
```
21  class Motion(object):
22
23      # This class responsible for sending commands to the robot
24      # by using ROS topic /RosAria/cmd_vel
25      # In the topic the linear and angular velocity are passing
26
27      def __init__(self):
28          self.motion = Twist()
29          self.pub = rospy.Publisher('/RosAria/cmd vel', Twist, queue_size=1)
30
31      def left_go(self):
32          self.motion.angular.z = .2
33          self.motion.linear.x = .2
34          self.pub.publish(self.motion)
35
36      def left_stay(self):
37          self.motion.angular.z = .2
38          self.motion.linear.x = 0
39          self.pub.publish(self.motion)
40
41      def right_go(self):
42          self.motion.angular.z = -.2
43          self.motion.linear.x = .2
44          self.pub.publish(self.motion)
45
46      def right_stay(self):
47          self.motion.angular.z = -.2
48          self.motion.linear.x = 0
49          self.pub.publish(self.motion)
50
51      def center_go(self):
52          self.motion.angular.z = 0
53          self.motion.linear.x = .2
54          self.pub.publish(self.motion)
55
56      def center_stay(self):
57          self.motion.angular.z = 0
58          self.motion.linear.x = 0
59          self.pub.publish(self.motion)
60
```

Figure 4.8 Motion control class which is created to control the robot

The twist message type and the publisher to /RosAria/cmd_vel topic will be created automatically in the class constructor (line 28,29) when the class gets initialized in the Machine Vision part. Its methods will be called depending on the results of object tracking and depth calculation. Since the topic has been updated the RosAria node continuously reads it. In case of various problems like wrong detection of tracking, the program sends the message with 0 linear and 0 angular velocity.

The name of this method is center_stay (line 56). The same method must be used at the end of program to stop the robot properly. Otherwise, the robot will keep executing the previous command.

The image below shows the output message from camera_parser node to the /cmd_vel topic. The first value in the list of two float values is a linear component of the message and the second one is an angular component. The message also includes a timestamp. This message outputs to the screen by using rospy.loginfo() function.


Applications Places System
19 °C mar. avril 23, 16:18
odroid@odroid: ~
File Edit View Search Terminal Tabs Help
odroid@odroid: ~
odroid@odroid: ~

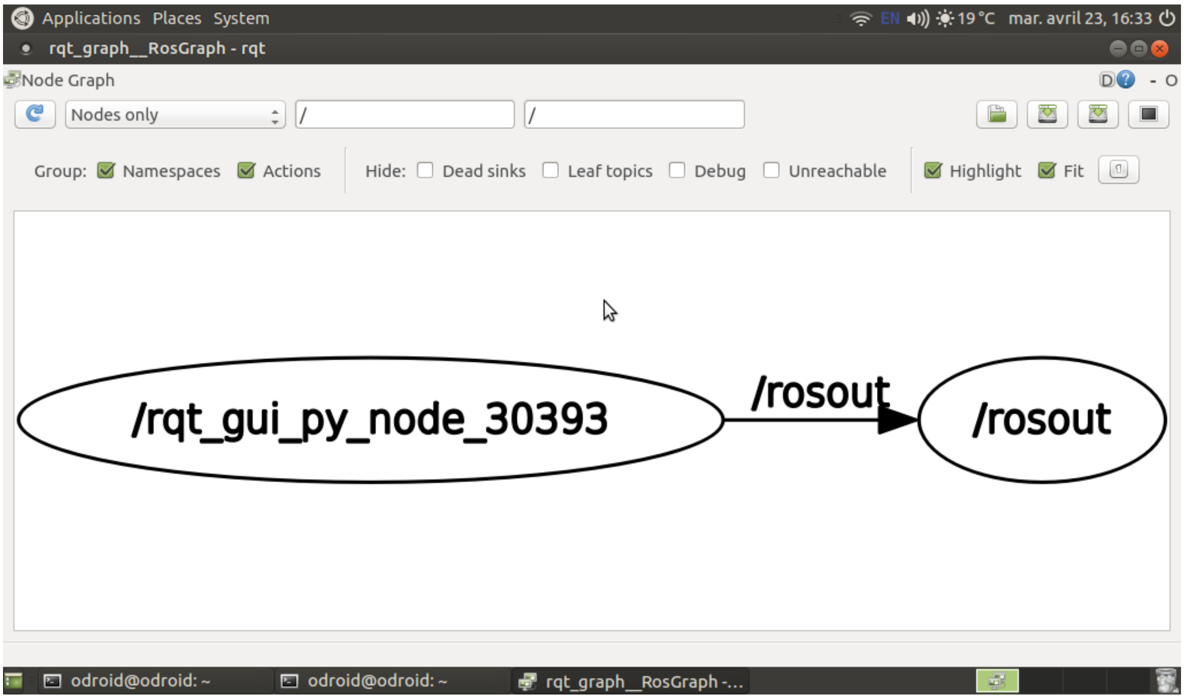
```
[ INFO] [1556028499.543265356]: new speed: [0.00,0.00](1556028499.543)
[ INFO] [1556028499.654069673]: new speed: [0.00,0.00](1556028499.654)
[ INFO] [1556028499.764571576]: new speed: [0.00,0.00](1556028499.764)
[ INFO] [1556028499.875969056]: new speed: [0.00,0.00](1556028499.876)
[ INFO] [1556028500.005028421]: new speed: [0.00,0.00](1556028500.005)
[ INFO] [1556028500.132036050]: new speed: [0.00,0.00](1556028500.132)
[ INFO] [1556028500.241218628]: new speed: [0.00,0.00](1556028500.241)
[ INFO] [1556028500.348745258]: new speed: [0.00,0.00](1556028500.349)
[ INFO] [1556028500.459042788]: new speed: [200.00,-0.20](1556028500.459)
[ INFO] [1556028500.671184235]: new speed: [0.00,0.00](1556028500.671)
[ INFO] [1556028501.257598528]: new speed: [0.00,0.00](1556028501.258)
[ INFO] [1556028502.110101082]: new speed: [200.00,-0.20](1556028502.110)
[ INFO] [1556028502.527873519]: new speed: [0.00,-0.20](1556028502.528)
[ INFO] [1556028503.375831187]: new speed: [200.00,0.00](1556028503.376)
[ INFO] [1556028503.490021483]: new speed: [200.00,0.00](1556028503.490)
[ INFO] [1556028503.701173853]: new speed: [200.00,0.00](1556028503.701)
[ INFO] [1556028503.852980445]: new speed: [200.00,0.00](1556028503.853)
[ INFO] [1556028503.965430627]: new speed: [200.00,0.00](1556028503.965)
[ INFO] [1556028504.080933247]: new speed: [0.00,0.00](1556028504.081)
[ INFO] [1556028504.201918748]: new speed: [200.00,0.00](1556028504.202)
[ INFO] [1556028504.318861526]: new speed: [200.00,0.00](1556028504.319)
[ INFO] [1556028504.431876496]: new speed: [0.00,0.00](1556028504.432)
[ INFO] [1556028504.550800302]: new speed: [200.00,0.00](1556028504.551)
[ INFO] [1556028504.659199886]: new speed: [200.00,0.00](1556028504.659)
[ INFO] [1556028504.774707131]: new speed: [200.00,0.00](1556028504.775)
[ INFO] [1556028504.889405048]: new speed: [0.00,0.00](1556028504.889)
[ INFO] [1556028504.996642431]: new speed: [0.00,0.00](1556028504.997)
```

Figure 4.9 The image presents how the executable file is publishing the message to the /cmd_vel topic during the work.

It is noticeable that the program publishes messages with high frequency. It conducts this operation every iteration. The publishing speed is approximately four messages per second. The speed depends on the numbers of operations which are conducted by the program.

4.4 Overview of the ROS system

Rqt tool which was mentioned at the beginning of the chapter is a perfect instrument to observe the system. By using this tool, the developer can get information about all available nodes and the communication between them. In this section, the ROS system is presented and described. This was done in order to visualize how the nodes are connected inside ROS. The visualization helps to understand the project better. It was conducted by launching the nodes one by one and checking the graph after each step. Firstly, the only ‘rosout’ node has been launched. It should also be mentioned that with ‘rosrun’ command each node has to be run in a separate terminal. In order to launch several nodes at the same time, the launch file has to be created, which was mentioned before.

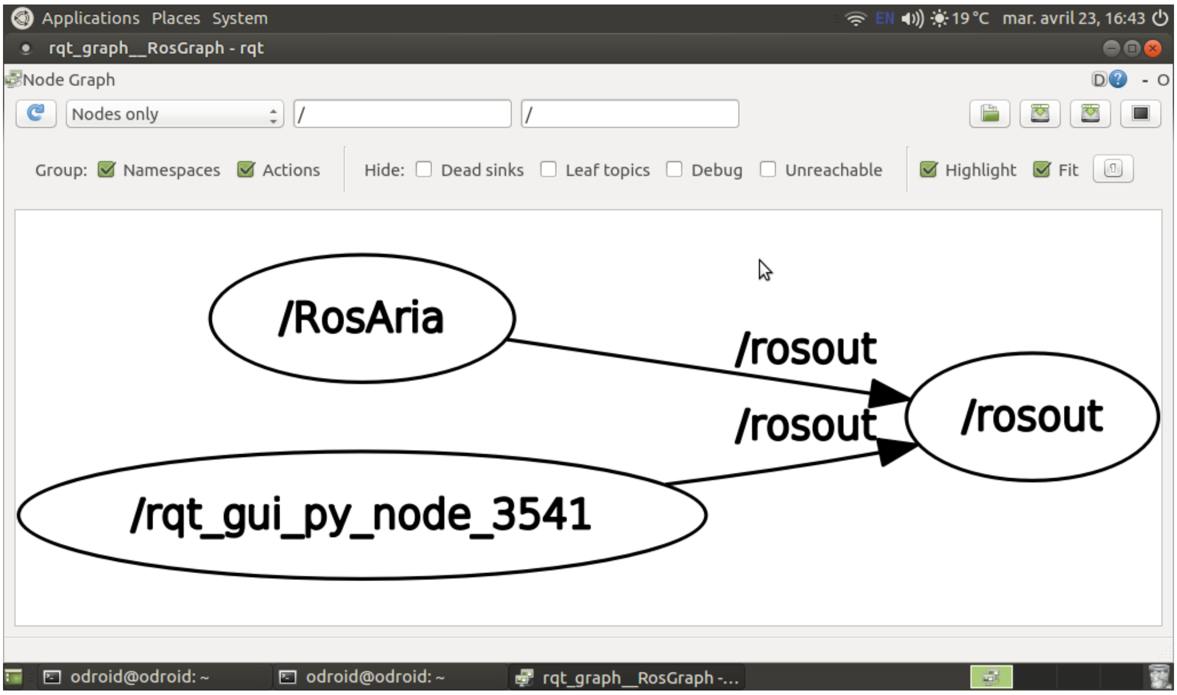


Nodes only / /

Group: ☑ Namespaces ☑ Actions Hide: □ Dead sinks □ Leaf topics □ Debug □ Unreachable ☑ Highlight ☑ Fit

Figure 4.10 ROS system with roscore node displayed in rqt_graph

The figure above illustrates that only /rosout and /rqt_gui_py_node_30393 nodes have been launched. Rqt_graph node has anonymous = True argument, that is why the name of the node always contains the random numbers at the end of the name. The next step is launching RosAria. As it was mentioned in the previous section, RosAria is the node which receives the motion commands and executes them. The figure below presents the ROS system with only roscore and RosAria nodes. The topics are hidden in the image, but it can be noticed that /RosAria and /rqt nodes communicate with /rosout. The reason why the name of roscore is /rosout in the graph is that because the command in the terminal and the name of the node are entirely different things and they are not related to each other. The developer creates the node inside the executable file and can name this node as he wants.

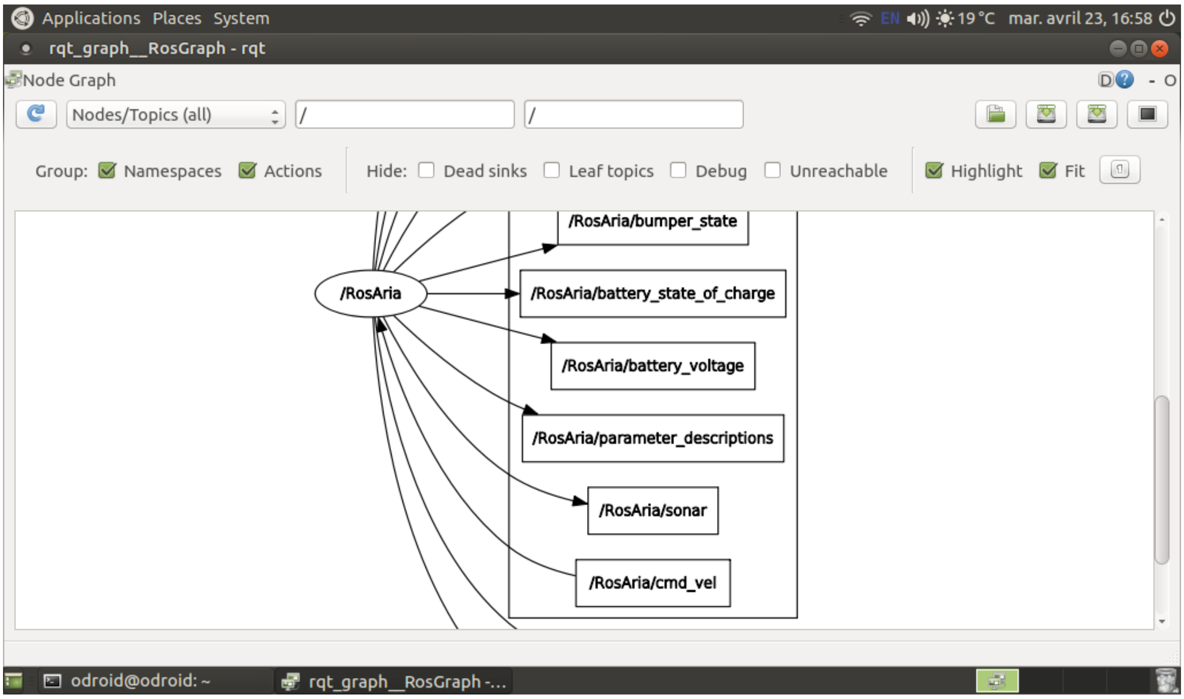


Nodes only

Group: ☑ Namespaces ☑ Actions Hide: ☐ Dead sinks ☐ Leaf topics ☐ Debug ☐ Unreachable ☑ Highlight ☑ Fit

Figure 4.11 ROS system with roscore and RosAria nodes

All screenshots above displays only nodes, but when RosAria is launched, it is better to present another figure which also shows all topics mentioned in the 4.3 section.



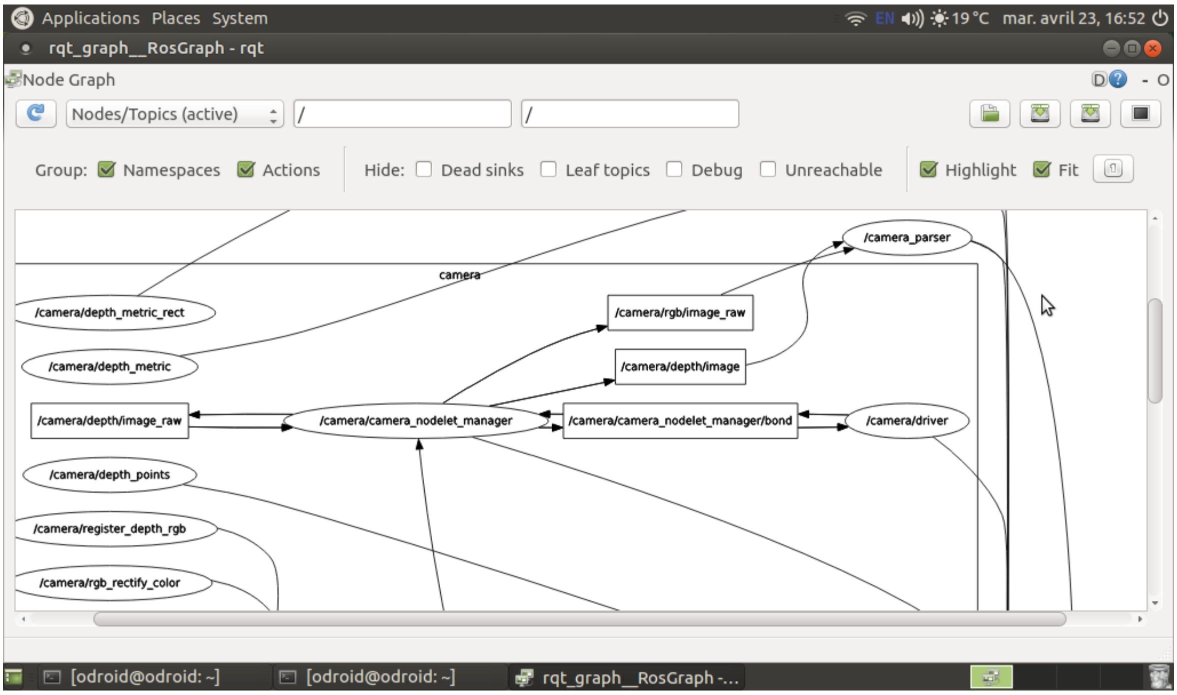
Nodes/Topics (all)

Group: ☑ Namespaces ☑ Actions Hide: ☐ Dead sinks ☐ Leaf topics ☐ Debug ☐ Unreachable ☑ Highlight ☑ Fit

Applications Places System
EN 19 °C mar. avril 23, 16:58
rqt_graph__RosGraph - rqt
Node Graph
/
/
/RosAria
/RosAria/bumper_state
/RosAria/battery_state_of_charge
/RosAria/battery_voltage
/RosAria/parameter_descriptions
/RosAria/sonar
/RosAria/cmd_vel
odroid@odroid: ~
rqt_graph__RosGraph -...

Figure 4.12 RosAria node and its publishing topics

The figure 4.12 presents the /RosAria node and the arrows which lead to the publishing topics: /RosAria/bumper_state, /RosAria/battery_voltage and so on. The next step is to launch the Astra camera node. This node will get the data from the camera and publish it to the relevant topics. The node publishes color images of different size, grayscale image, data from depth sensor in two formats and so on.



Nodes/Topics (active)

Group: ☑ Namespaces ☑ Actions Hide: ☐ Dead sinks ☐ Leaf topics ☐ Debug ☐ Unreachable ☑ Highlight ☑ Fit

Applications Places System
19 °C mar. avril 23, 16:52
rqt_graph__RosGraph - rqt
Node Graph
/
/
camera
/camera_parser
/camera/depth_metric_rect
/camera/rgb/image_raw
/camera/depth_metric
/camera/depth/image
/camera/depth/image_raw
/camera/camera_nodelet_manager
/camera/camera_nodelet_manager/bond
/camera/driver
/camera/depth_points
/camera/register_depth_rgb
/camera/rgb_rectify_color
[odroid@odroid: ~]
[odroid@odroid: ~]
rqt_graph__RosGraph -...

Figure 4.13 Astra came nodes

On the left side of the image 4.13, Astra camera nodes are seen. The Astra camera developers created quite complicated system controlled by camera_nodeled_manager which can be seen in the middle of the image. On the right side, the /camera_parser node is located. This node includes the executable code of the thesis project. It is visible that /camera_parser node subscribed to two topics provided by Astra camera. They are /camera/rgb/image_raw from which the color image is taking and /camera/depth/image from which the depth data is taking. The data is processed and analyzed inside the node and then based on the results the node publishes a message to /RosAria/cmd_vel topic. The line from the rights side of /camera_parser ellipse leads to the /RosAria/cmd_vel topic. The data processing and analyzing are described in the Machine Vision chapter.

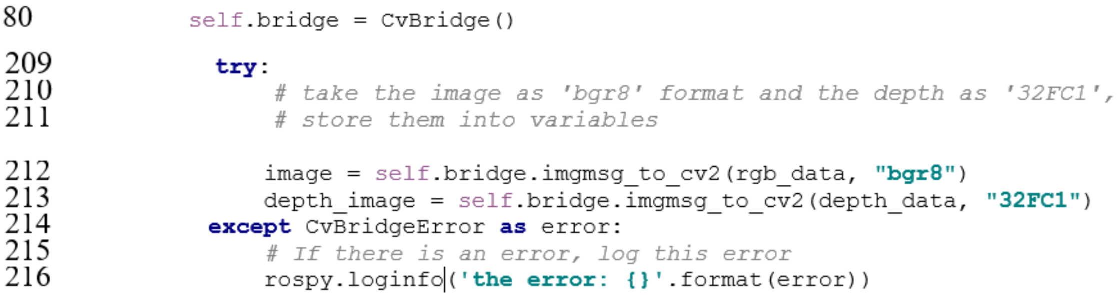
5. MACHINE VISION

The important part of this thesis is related to Machine Vision. The idea was to create a cobot type system where the robot tracks and follows the operator. To achieve this, it was initially decided to have an object which is easy to detect for the robot by using Machine Vision. Color detection algorithm was selected for that and as an object the tennis ball was used. The ball is easy to detect with color detection because it has bright yellow color which is easy to distinguish. However, color detection has several disadvantages. The first one is that it depends on the ambient light significantly. The second one is that the algorithm might detect other objects around which have a similar color. The second drawback can be eliminated by utilizing additional filters or functions in the algorithm. Unfortunately, it is much more difficult to find a suitable solution for the issue related to light conditions. It can be solved by applying the same light conditions at the plant or factory, but it cannot be done outside of the building. The color detection algorithm is described in the relevant section. Another option to detect the operator has been found after testing the color detection and finding that the performance might be not good enough. The idea of using neural networks, YOLO detection algorithm, in particular, was implemented. Primal YOLO neural network is trained to detect a limited number of things. It is able to detect around eighty objects like cats, dogs, cars. The network is also able to detect human beings which is required in that project. However, all the same, the additional detections or tracking algorithms are necessary, because the network will detect all operators who are in front of a camera and the robot will not know which operator it should track. All information related to YOLO object detection is presented in the 4.3 section. After detection algorithms, different tracking algorithms, which were used for object tracking once it was detected, are described. The last section of this chapter is devoted to the distance estimation. Two different methods of distance estimation are described. This part is essential as well as object detection because the robot is moving depending on the distance to the detected object.

All machine vision operations and conversions in the project have been done by using the OpenCV library. OpenCV is the most well-known and the most powerful library for solving the tasks related to machine vision. (7) The library has vast numbers of tools for image conversion and image preprocessing. OpenCV has a module structure, which means that the package includes several shared or static libraries. In the latest version of OpenCV several tracking algorithms, which will be described later, were added. These tracking algorithms have been used after color detection for tracking the object. The way to use YOLO detector is also implemented in OpenCV and can be used with embedded OpenCV functions.

5.1 Object detection

The first thing which has to be mentioned is the data conversion from ROS to OpenCV. As it was described in the ROS chapter, all messages in ROS have specific format. ROS passes images in its own sensor_msgs/Image format. In order to use the image with OpenCV, it has to be converted to BGR or another OpenCV format. For that a specific package is required. Cv_Bridge is a ROS library that provides an interface between ROS and OpenCV. The package should be separately installed and subsequently imported to the executable file. This package can be used in two directions, it can convert OpenCV image to the ROS message. The example of how that package was used during the work is presented in the code below:



```
80          self.bridge = CvBridge()

209           try:
210               # take the image as 'bgr8' format and the depth as '32FC1',
211               # store them into variables

212               image = self.bridge.imgmsg_to_cv2(rgb_data, "bgr8")
213               depth_image = self.bridge.imgmsg_to_cv2(depth_data, "32FC1")
214           except CvBridgeError as error:
215               # If there is an error, log this error
216               rospy.loginfo('the error: {}'.format(error))
```

Figure 5.1 Cv_bridge package is used in order to convert the image from ROS format to OpenCV format

In the line (80) the CvBridge object has been created. CvBridge object has several methods, and one of them is imgmsg_to_cv2() (line 212,213). This method requires two arguments: the first one is the input data from ROS which in the above image is rgb_data of sensor_msgs/Image message type, the second argument is the conversion encoding. As a result, the converted image of BGR8 type was stored into ‘image’ variable.

5.1.1 Object detection based on the color detection

For utilizing a color detection algorithm, a lot of preprocessing operation with the image has to be done. From the Astra camera, the developer receives raw BGR image with a resolution of 640 by 480. The resolution can also be reduced to increase the performance if it is needed. In the project that was not required, because the calculation speed was fast enough. Thus, the first step is blurring to get rid of sharp edges and make the image smooth. This can be done by applying various low pass filters. The Gaussian Blurring has been used with function cv2.GaussianBlur(). (7) While using that filter, the width and height of the kernel should be specified. It is obligatory for width and height to be positive and odd. A larger convolution kernel size will generally result in a higher degree of filtering. In other words, a bigger kernel effects more on the detail quality and increases blurring. The standard deviations sigma X and sigma Y can be specified as well. In case only sigma X was specified sigma, Y will be the same as sigma X. If none of them is specified, they will be calculated from kernel size. Sigma is a variance and increasing its value leads to an increasing degree of image blur as well as increasing the kernel size.

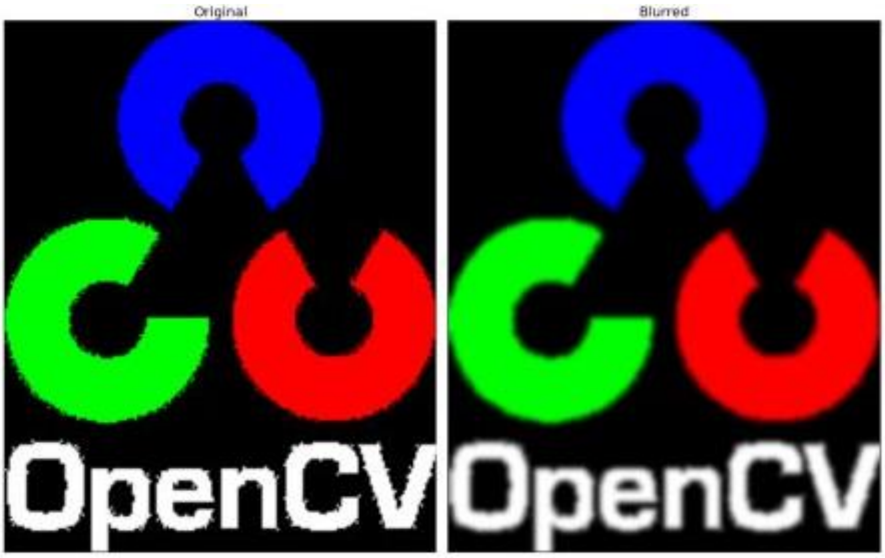
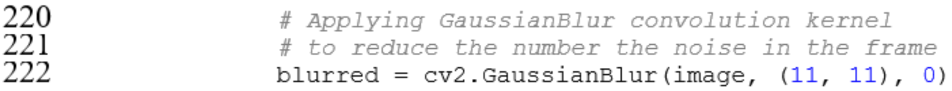


Figure 5.2 An example of Gaussian Blurring with OpenCV function (7)

The next part of the code presents the cv2.GaussianBlur() function and shows the values of arguments which were used.



```
# Applying GaussianBlur convolution kernel
# to reduce the number the noise in the frame
blurred = cv2.GaussianBlur(image, (11, 11), 0)
```

Figure 5.3 Using of GaussianBlur() function in the project

The first argument is an input image which was received from a camera node and converted to BGR format, the second one is tuple which describes a kernel size, and the last one is sigma X. The kernel size and sigma were determined based on experience and can be changed if it is needed to achieve better detection results.

The next operation in the image preprocessing is a converting to HSV color space. There are more than 150 color-space conversion methods available in OpenCV and HSV is one of the most popular one as well as BGR and Grayscale. For color conversion, the function cv2.cvtColor() is used. This conversion is required because it is easier to emphasize a specific color which the developer wants to detect in HSV color space. HSV color space can be described as hue, saturation, value. Hue is the color portion of the color model, saturation is the amount of gray in color and value describes the brightness or intensity of the color. The important point is that in OpenCV for HSV, hue range is [0, 179], saturation range is [0, 255], and value range is [0, 255]. This must be taken into account if these values are taken from another software.

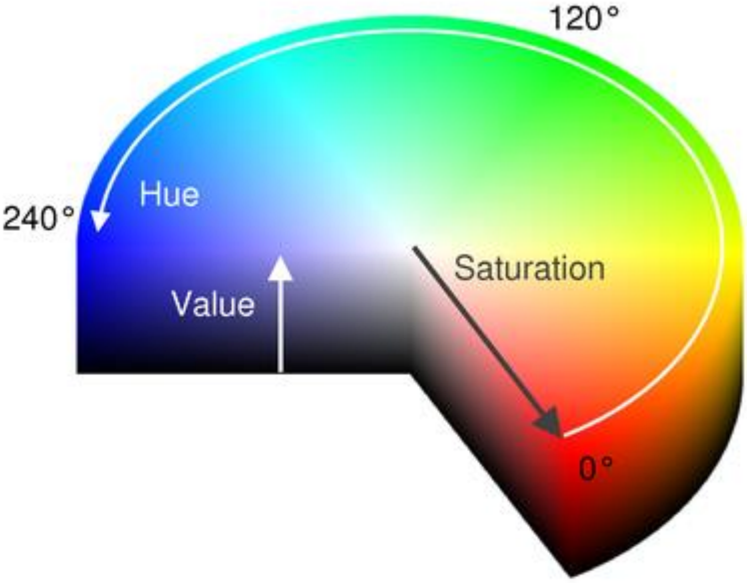
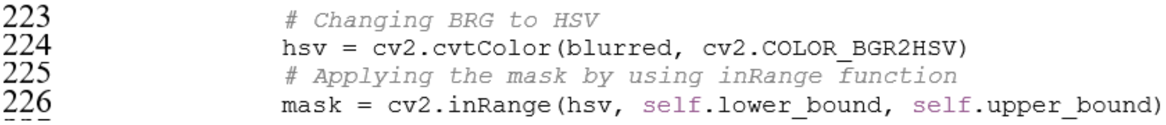


Figure 5.4 HSV model (17)

The HSV model above can visualize hue, saturation and value. In the project the HSV values for yellow color were used, which are [29, 75, 6, 65, 255, 255]. However, these values can vary greatly depending on the lighting conditions, so before launching the code, they should be checked and adjusted. Lighting conditions dependency is the biggest drawback of the color detection algorithm. The fact that even in the different parts of the same room light conditions could be different, reduces the robustness of the color detection significantly. The list of HSV values consists of six numbers because the first three describe the lower bound and the second three describe the upper bound. This will be described more precisely later. In order to get the HSV boundary values, an additional program was used. The program was created to define the HSV values for a specific color and subsequently can be added to the main code. The code creates trackbars for H_min, S_min, V_min, H_max, S_max, V_max. Each iteration the program extracts the values from trackbars and uses them with cv2.inRange() function to highlight the described color. The function cv2.inRange is also used in the main code to apply a threshold. This function separates the pixels which have HSV values in the range between upper and lower bound by changing their values to 255, and pixels which are outside that range to 0. This way the image becomes a binary in order to find the contours of the object. While launching the main code, it demands to choose the option for HSV values. The operator can use the default ones, which are mentioned above or type new oens. Next part of the code is related to color space conversion and creating the mask for a specific color.

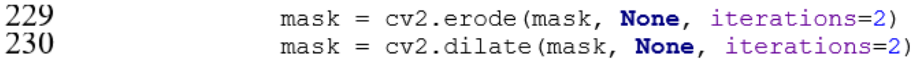


```
# Changing BRG to HSV
hsv = cv2.cvtColor(blurred, cv2.COLOR_BGR2HSV)
# Applying the mask by using inRange function
mask = cv2.inRange(hsv, self.lower_bound, self.upper_bound)
```

Figure 5.5 Changing the image encoding and applying inRange() function

The color conversion function takes an input image as a first argument and a conversion flag as a second argument. In case of an inRange() function, there are three arguments required: input image, lower bound and upper bound. This is why the HSV list consists of six values.

The third part of image preprocessing is a high frequency noise elimination. After the binary image with the required color was acquired, it still has a lot of small parts which were extracted from object because they have the HSV values inside a boundary range. This noise exists almost always since the different pixels of the object have slightly different HSV values. Moreover, depending on the lighting conditions the HSV range can get bigger which will increase the noise level. The high frequency noise can be eliminated by using morphological operations. Morphological transformations are simple operations based on the image shape which are typically performed on binary images. There are two operations considered: erosion and dilation. The basic idea of erosion is like spoil erosion, it erodes the boundaries of the foreground objects. In the image, the foreground consists of white pixels, and the background consists of black pixels. The erode function takes the kernel and slides it through the image, pixel in the original image will be considered as 1 (which it white) only if each pixel under the kernel is 1. Otherwise, it will be made 0 (black). The size of the kernel can be specified in parameters, or the function will consider this parameter as a default which is 3x3. The bigger kernel leads to increased erosion level. Another parameter, which is 1 as a default, is the iteration number. Increasing this parameter increases the erosion level as well.



```
mask = cv2.erode(mask, None, iterations=2)
mask = cv2.dilate(mask, None, iterations=2)
```

Figure 5.6 Erode and dilate functions

The part of the main code presented above shows how the cv2.erode() function is used. It takes the image and erodes it with a default kernel size twice. Afterward, the program assigns the new image with the same variable. As a result, the size of the detected object will be decreased, and in order to return the primary object size, an opposite operation, which is dilation, has to be performed. The cv2.dilate() function slides the kernel through the original image, a pixel in the original image will be considered as 1 if at least one pixel under the kernel is 1. In the program, the dilate operations goes after the erode with the same number of iterations.

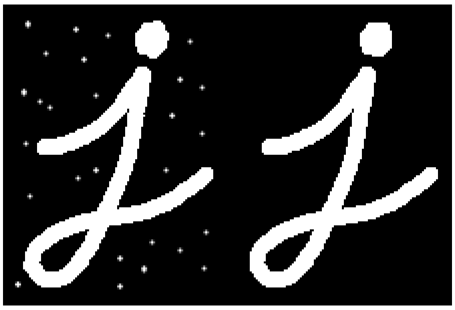


Figure 5.7 The result of performing cv2.erode() and cv2.dilate() functions. On the left side is an image before performing functions. (7)

A clear binary image has to be achieved at the end of image processing. The detected object, which is a tennis ball, has to be white and the other things have to be black. This binary image will be an input to another function. This function is looking for the contours and will be described in the next paragraph. A small white spot is visible on the left of the detected ball in the figure below. It is a ladder which has a quite similar color with a ball. This contour will be avoided by looking for the biggest contour in the list of contours.

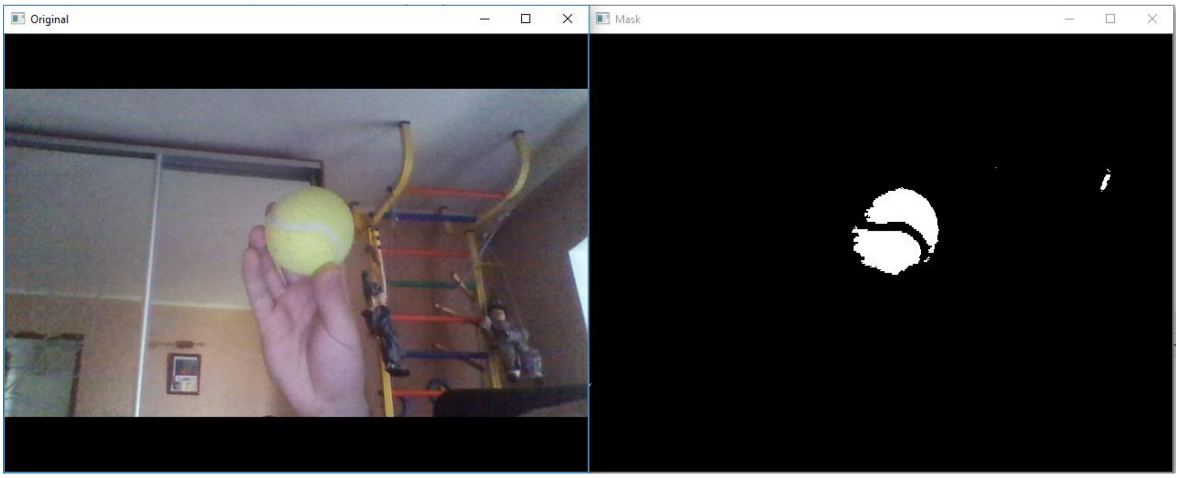
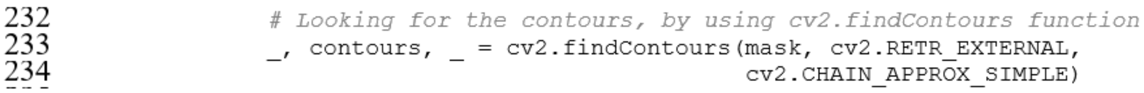


Figure 5.8 The output binary image after image processing

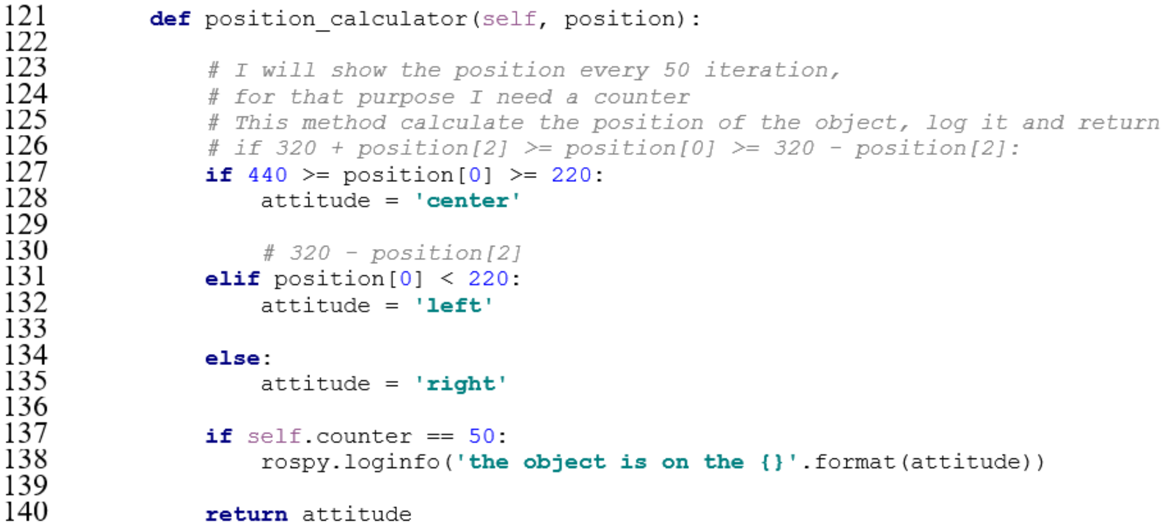
Since all pre-processing operations have been done, the program can go ahead and start looking for contours in the image. This is conducted with the OpenCV function cv2.findContours(). As it was mentioned before, OpenCV is a powerful library for Machine Vision and this tool is often used for object detection and recognition. For better accuracy, the function has to be used with binary images. It simply detects a border between different pixel values. Each contour is a NumPy array of (x, y) coordinates of boarding points of the object. The function stores all founded contours in a Python list. The part of code related to contours detection presented below.



```
        # Looking for the contours, by using cv2.findContours function
        _, contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL,
                                          cv2.CHAIN_APPROX_SIMPLE)
```

Figure 5.9 Looking for contours after image preprocessing

There are three arguments in cv2.findContours() function: source image, contour retrieval mode and contour approximation method. The second argument is related to contour hierarchy, and it can be useful when the developer, for instance, wants to omit nested contours. In this project, it is not essential because it is considered that there will be only one detected object without any additional frames. Contour approximation method is the third argument, and it is added to make the code works faster. It removes all redundant points and compresses the contour in order to save memory. In the case of ball detection, the function might not reduce an array much, but in other cases, this will be useful. Afterwards, the program takes the biggest contour which has to be a tennis ball and calculates minimum enclosing circle with OpenCV function cv2.minEcnlosingCircle(). This function takes only an array of points and returns the (x, y) coordinate of the circle and the radius. This one gives precise information about the position of the ball in front of the robot. Depending on the position, the robot turns left, right or stay still. The next block of code shows the method which takes the ball position and returns the command to the robot.



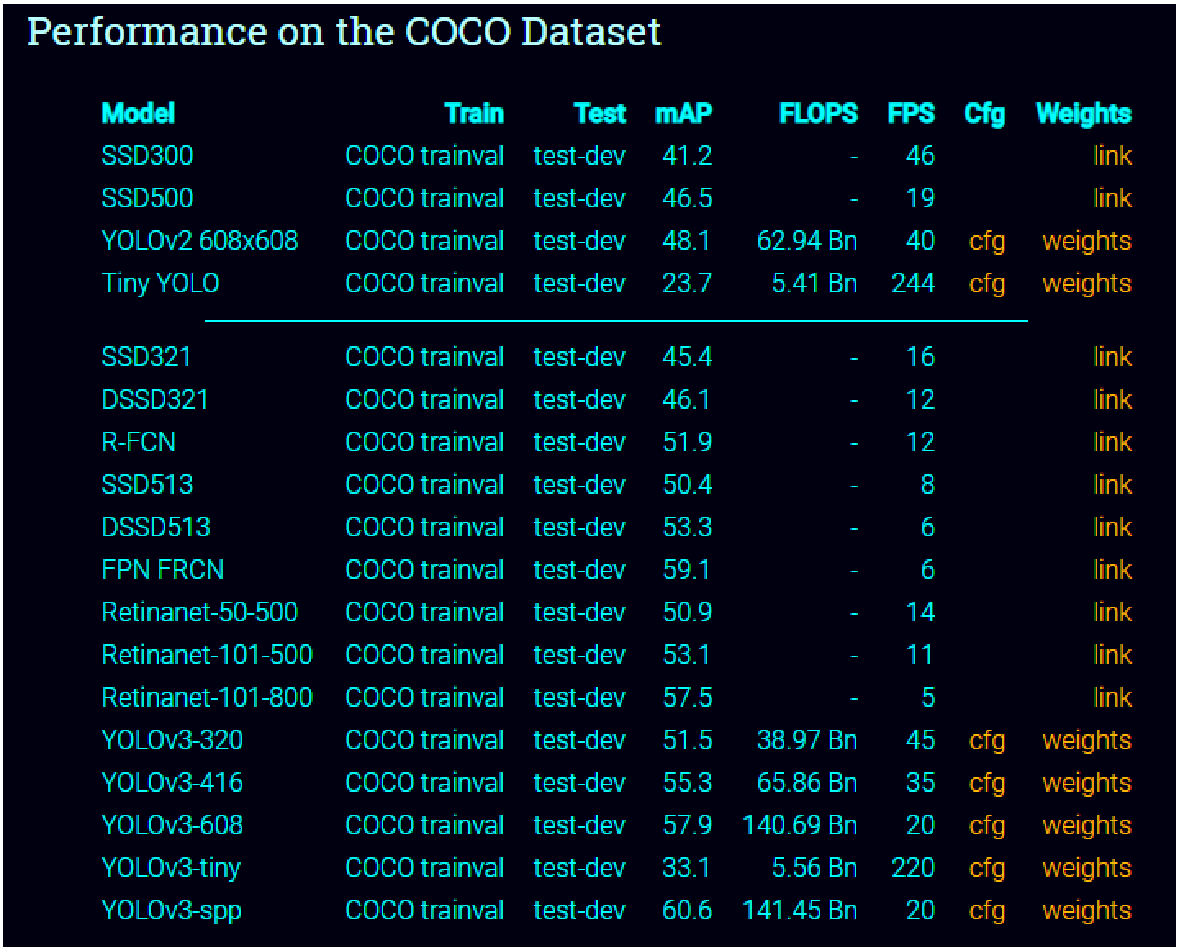
```
    def position_calculator(self, position):

        # I will show the position every 50 iteration,
        # for that purpose I need a counter
        # This method calculate the position of the object, log it and return
        # if 320 + position[2] >= position[0] >= 320 - position[2]:
        if 440 >= position[0] >= 220:
            attitude = 'center'

            # 320 - position[2]
        elif position[0] < 220:
            attitude = 'left'

        else:
            attitude = 'right'

        if self.counter == 50:
            rospy.loginfo('the object is on the {}'.format(attitude))

        return attitude
```

Figure 5.10 Position calculation method which based on the object position outputs in which direction the robot should rotate

5.1.2 YOLO object detection

Another way of object detection is the use of neural networks. The neural network is a fairly novel technology which has started developing roughly fifty years ago. Moreover, the development speed of this field of technologies is increasing. Every day the developers train new classifiers with better performance which are based on the neural network approach. The working principle of the neural network is not related to this project; hence it will not be described. This section is about YOLOv3 object detector which was developed in Darknet and is the latest variant of this detection algorithm. Darknet is an open source neural network written in C and CUDA. (4) The user can find the source and materials on GitHub or can read about it on its webpage. You only look once (YOLO) is a state-of-the-art, real-time object detection system. The main benefit of the algorithm is fast and at the same time accurate performance. In the latest version, the user can even tradeoff between speed and performance by changing the size of the model. In that case, the retraining of the net is not required. The YOLOv3 detector uses completely different approach compared to other classifiers. In traditional computer vision approaches, the sliding window was used to look for objects in the input image in different locations and scales. YOLO detector forwards the image through the network only once. The network divides it into regions and predicts boundary boxes and the probability of each region. After that, the user can threshold the result by some value to filter out the boundary boxes with low probability. (18) This approach increases the parsing speed extremely fast, more than one thousand times faster compare to R-CNN neural network. Initially, the developers did not compile Darknet with OpenCV, and it was not possible to display the image with results of the detection directly. Instead, it saves the result in a png format file. Additionally, the Darknet outputs the result in the terminal after processing. The guide on how to install Darknet and use YOLO detection algorithm there can be checked on their official webpage. (4) In this project, the way how this algorithm is used inside the Darknet framework is not discussed due to the fact that we are engaging in applying that detection algorithm in conjunction with other functions in ROS. This section is related to the information about how YOLO detection can be used with OpenCV. Due to the lack of time, this approach wasn’t tested with the ODROID XU4 board and Pioneer 3-DX robot. However, it can be said for certainty that the speed of processing would be slower with the standard CPU and it would be better to replace the single board computer in order to achieve better results.

The YOLO detection algorithm model which will be used is trained on COCO (common objects in context) from Microsoft. The model can detect eighty common objects. (19) In order to run the network, the dnn, which is Deep Neural Network module, has been used. This module initially was a part of opencv-contrib repository. However, it was moved to a master branch last year. It allowed developers to use this module with ARM processor architecture unlike tracking algorithms integrated into OpenCV. These algorithms will be described in the relevant chapter. The dnn module is not used to train or re-train the network. It is used only to launch the net. Before running the YOLO detection, several files should be downloaded from GitHub. These are the network configuration file, pre-trained weight coefficients and the list containing class names which were mentioned at the beginning of this paragraph. All these files could be passed as arguments through the terminal. Another option is to store them into variables by specifying the path to each file inside the program separately. It has to be mentioned that there are several YOLO network models available in the Darknet, and the user can choose any of them. The figure 4.8 present the table of different neural networks and the results of tests. All the networks are trained on the same data sets. It is noticeable that the YOLO detectors process the data with higher FPS. Which means that they are faster than others. Some YOLOv3 models contain the numbers in the name. This numbers means the size of the images on which that model was trained. The same size should be specified while preparing the blob for that net in order to achieve good performance. Is will be described below.



Performance on the COCO Dataset

Model	Train	Test	mAP	FLOPS	FPS	Cfg	Weights
SSD300	COCO trainval	test-dev	41.2	-	46		link
SSD500	COCO trainval	test-dev	46.5	-	19		link
YOLOv2 608x608	COCO trainval	test-dev	48.1	62.94 Bn	40	cfg	weights
Tiny YOLO	COCO trainval	test-dev	23.7	5.41 Bn	244	cfg	weights
SSD321	COCO trainval	test-dev	45.4	-	16		link
DSSD321	COCO trainval	test-dev	46.1	-	12		link
R-FCN	COCO trainval	test-dev	51.9	-	12		link
SSD513	COCO trainval	test-dev	50.4	-	8		link
DSSD513	COCO trainval	test-dev	53.3	-	6		link
FPN FRCN	COCO trainval	test-dev	59.1	-	6		link
Retinanet-50-500	COCO trainval	test-dev	50.9	-	14		link
Retinanet-101-500	COCO trainval	test-dev	53.1	-	11		link
Retinanet-101-800	COCO trainval	test-dev	57.5	-	5		link
YOLOv3-320	COCO trainval	test-dev	51.5	38.97 Bn	45	cfg	weights
YOLOv3-416	COCO trainval	test-dev	55.3	65.86 Bn	35	cfg	weights
YOLOv3-608	COCO trainval	test-dev	57.9	140.69 Bn	20	cfg	weights
YOLOv3-tiny	COCO trainval	test-dev	33.1	5.56 Bn	220	cfg	weights
YOLOv3-spp	COCO trainval	test-dev	60.6	141.45 Bn	20	cfg	weights

Figure 5.11 The results of different tests of various neural networks which are available on the Darknet webpage (4)

After all files have been passed, a separate color has to be assigned to each class in order to make a distinguished boundary box for each class subsequently. It can be done by using random uniform function from NumPy module. This function requires upper and lower bounds (which has to be from 0 to 255), and the size of the output matrix. (20) The next step is to read a pre-trained net, which is defined with a downloaded YOLO detection weights file and configuration file.

```
net = cv2.dnn.readNet(weights, config)
```

Since the neural network is initialized, the code can go ahead. As well as the color detection, the input image (in case of real-time detection it is a frame from video stream) should be pre-processed in order to facilitate the detection process for the neural network.

```
blob = cv2.dnn.blobFromImage(image, scalefactor, size, mean, swapRB=True, crop=False)
```

The function above outputs the processed blob which is subsequently used as an input to the net. It takes six arguments:

- Input image or frame from video
- Scale factor
- Spatial size for the output image
- Mean values which are subtracted from channels. Values are intended to be in RGB order
- SwapRB option. It is required to be True if the input image is presented in BGR format
- Crop option. This flag indicates whether the image will be cropped after resizing or not

All parameters which are related to the image modification, directly depend on the network architecture. For instance, the spatial size preferable should be the same as the size of images on which the network was trained. The scale factor is 1.0 as a default, but reducing this parameter leads to increasing the processing speed and performance. Usually the developers make that parameter equal to 1/255 to have the pixel values between 0 and 1. The mean values are a 3-tuple consisting of the mean of the Red, Green, and Blue channels respectively. The mean subtraction is used to combat illumination in the input images or frames. It intended to reduce the influence of changing light conditions.

After the input image has been converted to the blob, it can be passed to the network. The developer passes an input to the neural network with net.setInput() function in Python and then runs the net with net.froward() function. The important thing is that the forward function requires the ending layer until which it should run in the net. Since we want to achieve good performance, we need to go through the full net and define the last layer. If the ending layer is not specified, the net outputs the results from all layers in which we are not interested. Getting the ending layer can be performed with the function getUnconnectedOutLayers() that gives the names of unconnected layers which are apparently the last layers in the net. The YOLO detector outputs the boundary boxes of predicted objects with a confidence that the boundary box encloses an object. The post-process step should be performed as well as pre-processed, in order to eliminate the objects with low probability by thresholding the results and assigning the class to each boundary box. For each detected object the net also outputs the confidence associated with each class. So, the object can be named by assigning to the class with a higher probability. Then the objects could be added to the dictionary with their coordinates on the frame to use this data later for mapping. Finally, the developer can also draw them in the image to get better visualization. However, for the robot, it is not required.

```
# Sets the input to the network
net.setInput(blob)
# Runs the forward pass to get output of the output layers
outs =    net.forward(getOutputsNames(net))
```

This section is mainly devoted to the interface and how YOLO detector can be used. However, it is possible to train a custom YOLOv3 model on a custom dataset. With that, the neural network can be prepared for any object. This process requires a powerful computer because it will influence the training speed. A huge training data set should also be prepared. It has to be mentioned that the YOLO detector will only detect the objects, but not track them. It means that in case of this project, the robot will not understand what it should track. The detector will detect, for instance, all human beings in front of it. The program can assign the unique ID number for each detected object, but when the operator will move out of the frame and back the new ID will be assigned. The combination of approaches can be used to solve that problem. Each operator can be marked with a vest with a unique color, and the color detection in conjunction with YOLO detection can be applied. In that case, the logic can be such that if the specific color is detected inside the operator boundary box, then the robot found the correct operator and will start to follow him. The additional integrated OpenCV tracking algorithms can be used for the tracking purpose as well. Their descriptions will be in the next section.

5.2 Tracking algorithms integrated into OpenCV

One of the most important fields of studies in Machine Vision is the object tracking. The previous chapters describe how to detect an operator or an object which distinguishes the operator on each frame. However, does it mean that the object on the previous frame is the same object on the current frame? The tracking process can be defined as assigning a unique ID number to each detected object on the first frame of a video stream and keeping those numbers on the subsequent frames until the tracked object will not appear on the video. So, tracking preserves the identity. An excellent example of a simple tracking algorithm is a centroid tracker. It is based on calculating Euclidean distances between the center points of detected objects from the previous frame and the new frame. It assumes that the nearest center point in the new frame compared to the previous frame is the same object being tracked. If a newly detected object appears, it is registered with a new ID. The code detects a new tracked object, if there is one boundary box more in the current frame, compared to the previous frame. It was just an example to better understand the purpose of tracking and why it is required.

With OpenCV 3.0, the developers of the module introduced a tracking API as well. In OpenCV 3.4.1 version there are eight trackers available. Each of them works as trackers and detectors at the same time. Moreover, tracking algorithms are usually faster than detection algorithms. The reason is that when the program tracks an object which was detected in the previous frame, the program knows a lot about the appearance of the object. It also knows the direction of its motion, which is important for most of the tracking algorithms. The direction and velocity of the tracked object can be called a motion model. However, there is more information: the program knows the appearance of the object which can be called an appearance model. In other words, appearance model encodes what the object looks like. In many modern trackers, this appearance model is a classifier that is trained in an online manner. That classifier aims to classify an area of the image either as an object or not. The online classifier is trained online by feeding it positive (object) and some negative (background) examples. The robustness of that classifiers is much lower compare to offline neural networks. So, based on the motion model and appearance model, the trackers predict the location in the current frame. However, sometimes trackers can fail, and that's why the trackers have to be applied in conjunction with a detection algorithm. In the beginning, OpenCV trackers require only an initial boundary box of the tracked object. The important point is that the original boundary box can be specified manually by the user or another detection algorithm. Each tracker has its own approach with its advantages and disadvantages. This section is about tracking algorithms, how they can be used, and which one of them gives the best performance. (7) As it was mentioned, there are eight trackers available in OpenCV: BOOSTING, MIL, KCF, TLD, MEDIANFLOW, GOTURN, MOSSE and CSRT. Brief information about some trackers is provided below.

BOOSTING tracker is based on the online version of AdaBoost. The AdaBoost algorithm is used in HAAR cascade face detector. That tracker uses the online classifier which is described previously. Since the tracker receives a new frame, the classifier is run on each pixel in the neighborhood of the previous location and scores of each result is recorded. The area with a higher score is the tracked object. Additionally, one more positive image for the online classifier is received. This method is almost ten years old, and there is no reason to use it when other advanced trackers are available.

MIL tracker is a much more sophisticated tool compared to BOOSTING tracking system. The approach is quite similar but MIL trackers, instead of considering only one location as positive sample, take a small neighborhood around the object location to generate a set of positive examples. It might be argued that the object on those positive examples will be not centered and this is where Multiple Instance Learning (MIL) comes to the rescue. This tracker creates positive and negative “Bags”, instead of specifying positive and negative examples. Only one image in the positive bag has to be the positive example. MIL tracker provides a pretty good performance, and it does a reasonable job even with partial occlusion. (21)

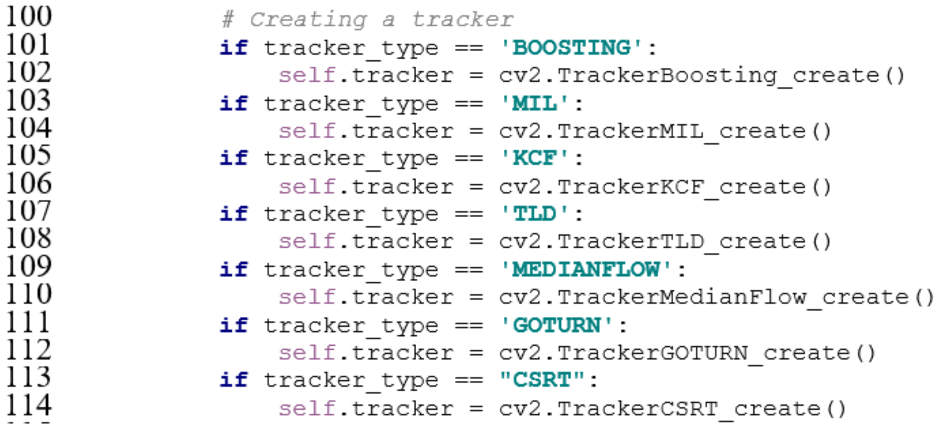
KCF tracker stands for a Kernelized Correlation Filter. This tracking algorithm is built on the ideas from the previous two trackers. Additionally, this tracking system takes the information from overlapping regions from the positive bag. This information allows to improve the speed and quality of the tracking system. It can be said that this one is the last stage of development of previous trackers. KCF was tested in this project and showed one of the best results. It also reports tracking failure better than previous trackers. However, this one still does not recover from full occlusion.

Next tracker is TLD which stand for Tracking, Learning, and Detection. This tracker localizes all appearances that have been observed before and corrects the detector if it is needed. It is an excellent example of online classifiers, which are learning while working. On the one hand, the drawback of this algorithm is that it can jump around similar objects, like cars. On the other hand, because of the online correction, this tracker is able to track the objects over a larger scale, motion or occlusion. This tracker can detect the object even if it completely disappeared for several frames. However, as it was mentioned, the huge number of false detections makes the TLD almost unusable.

The 5th tracking algorithm is a MEDIANFLOW tracker. Internally, this one used in order to track an object in both forward and backward directions. The work is related to real-time object tracking, so this tracker is unusable. This algorithm works almost perfectly when the motion is predictable and slow. The good feature of this algorithm is that it knows when the tracking has failed and therefore has the best tracking failure reporting.

GOTURN tracker is the only one which is based on Convolutional Neural Network (CNN). From OpenCV literature, it can be stated that this tracker is stable under viewpoint changes, light changes, and deformations. However, it is not good in overcoming the occlusions. This approach was tested with the tennis ball, but it did not give a good result.

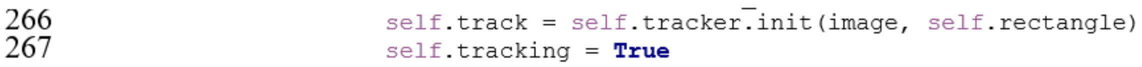
In this project, the tracking system was used in conjunction with a detection system. As it was mentioned before, almost all trackers require only an initial boundary box of the tracked object. Therefore, the logic was that the detection algorithm detects the object only once and sends its coordinates to the tracker. Since the tracker gets the boundary box, the program initiates it. In case tracking fails, the code goes back to the detection phase and tries to detect the object again. The critical point is that the program has to recreate a tracker object every time the tracking fails. It is needed because a lot of trackers are training online and changing themselves during work. The information below will show how the tracker was used in the project. The best results were achieved with KFC tracker, this one will be used as an example (line 106). Firstly, the developer has to create a tracker.



```
100        # Creating a tracker
101        if tracker_type == 'BOOSTING':
102            self.tracker = cv2.TrackerBoosting_create()
103        if tracker_type == 'MIL':
104            self.tracker = cv2.TrackerMIL_create()
105        if tracker_type == 'KCF':
106            self.tracker = cv2.TrackerKCF_create()
107        if tracker_type == 'TLD':
108            self.tracker = cv2.TrackerTLD_create()
109        if tracker_type == 'MEDIANFLOW':
110            self.tracker = cv2.TrackerMedianFlow_create()
111        if tracker_type == 'GOTURN':
112            self.tracker = cv2.TrackerGOTURN_create()
113        if tracker_type == "CSRT":
114            self.tracker = cv2.TrackerCSRT_create()
```

Figure 5.12 Creating the tracker object.

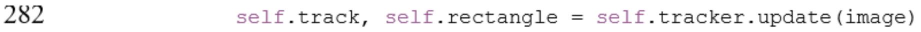
After the tracker has been created, it has to be initialized. Initialization is like a first launch, the image, and the first boundary box of the tracking object have to be passed as arguments. In the code below, self.rectangle is a variable which contains the boundary box of the tracked image. Self.tracking variable is used to stop color detection. When it is True the program skips the detection part and uses only tracking algorithm. Otherwise, it goes back to detection part. The program changes this variable to False value if the tracker returns failure detection.



```
self.track = self.tracker.init(image, self.rectangle)
self.tracking = True
```

Figure 5.13 The tracker initialization. The primal boundary box the program gets from color detection.

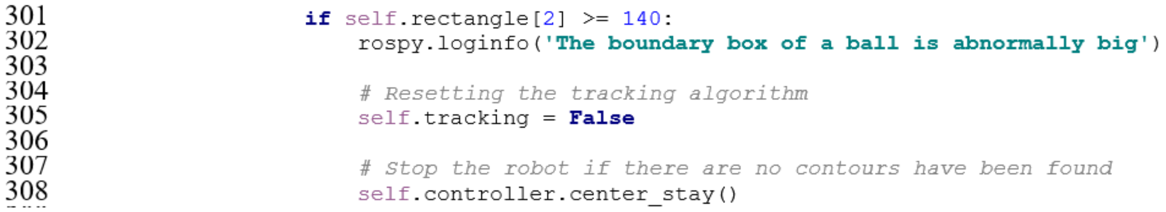
Then the tracker needs to update the coordinates of the object. Function self.tracker.update() is intended to do that. The new frame has to be passed as an argument. This function returns two outputs: the first one reports a tracking failure and the second one is a new boundary box of the tracked image.



```
282                 self.track, self.rectangle = self.tracker.update(image)
```

Figure 5.14 The tracker update function

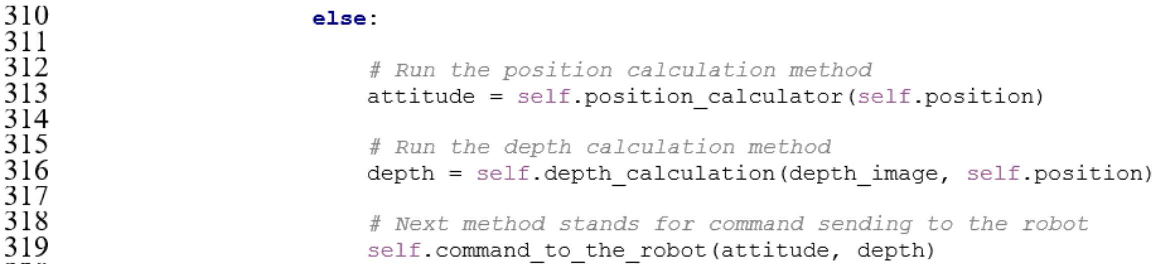
The first output is essential due to the fact that the program redetects the object depending on it. So, as it was mentioned if self.track is False, the program goes back to the detection part, detects the object again, recreates the tracker and supplies it subsequently with new boundary box. The same operations are performed if the tracking boundary box increases abnormally. The program checks the width of the box, and if it is more than 140 pixels (line 301), then it makes self.track equal to False as well as the tracking failure.



```
301                     if self.rectangle[2] >= 140:
302                         rospy.loginfo('The boundary box of a ball is abnormally big')
303
304                         # Resetting the tracking algorithm
305                         self.tracking = False
306
307                         # Stop the robot if there are no contours have been found
308                         self.controller.center_stay()
```

Figure 5.15 The program check the output boundary box

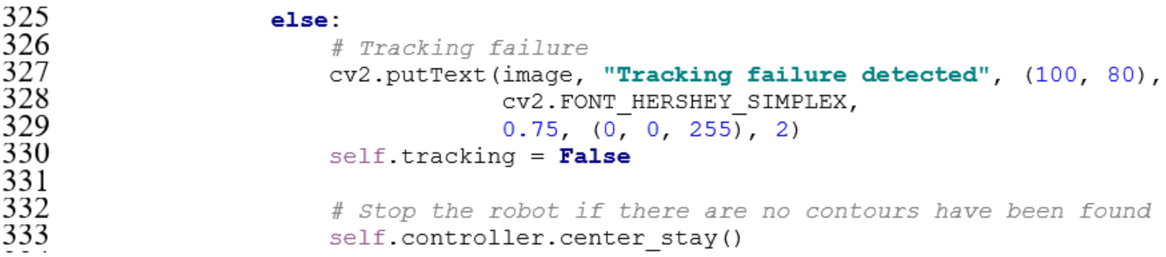
Otherwise, if self.track is True and the boundary box is not too big, the program performs the position calculation, depth calculation, and it sends the command to the robot according to it. The position calculation was mentioned before, and the depth calculation will be described later.



```
310                     else:
311
312                         # Run the position calculation method
313                         attitude = self.position_calculator(self.position)
314
315                         # Run the depth calculation method
316                         depth = self.depth_calculation(depth_image, self.position)
317
318                         # Next method stands for command sending to the robot
319                         self.command_to_the_robot(attitude, depth)
```

Figure 5.16 The program runs the position and depth calculation and then uses command_to_the_robot function to make a decision. This part runs only if self.track is True and the size of the boundary box is okay

The next part of code will show how the program reacts if self.track is False. It simply writes on the screen that the tracking failure and changes self.tracking variable to False. If self.tracking variable is False, the programs goes to the detection part of code. In that situation the program stops the robot by publishing the message with 0 linear and angular speed.

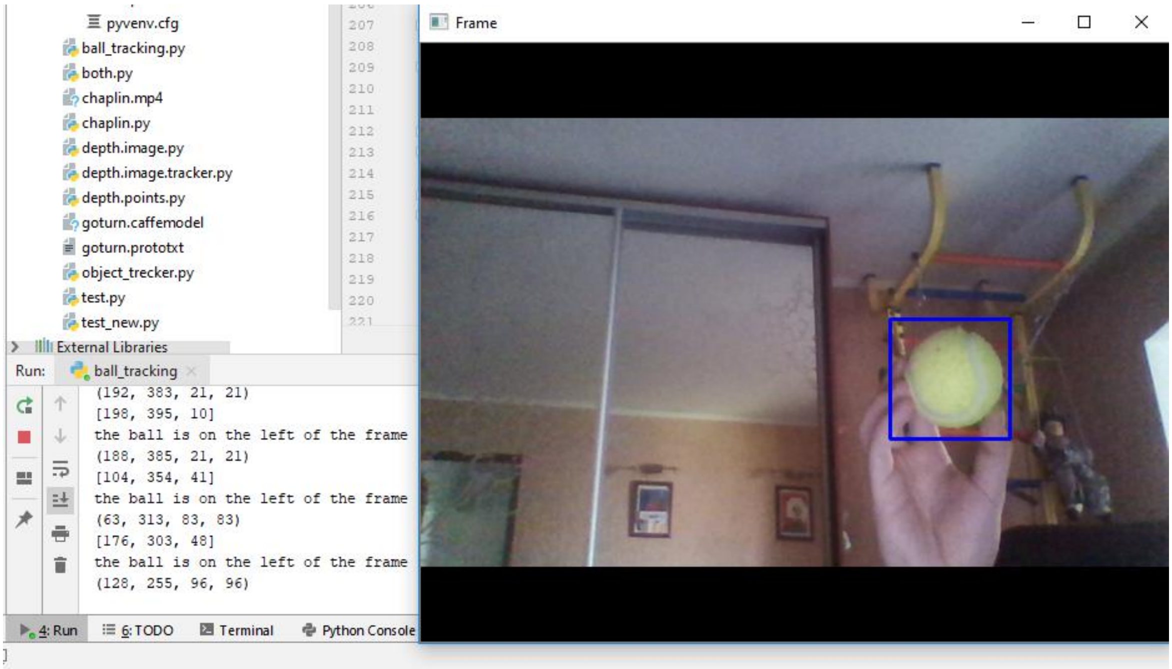


```
else:
    # Tracking failure
    cv2.putText(image, "Tracking failure detected", (100, 80),
                cv2.FONT_HERSHEY_SIMPLEX,
                0.75, (0, 0, 255), 2)
    self.tracking = False

    # Stop the robot if there are no contours have been found
    self.controller.center_stay()
```

Figure 5.17 The program executes this part of code in case of failure detection

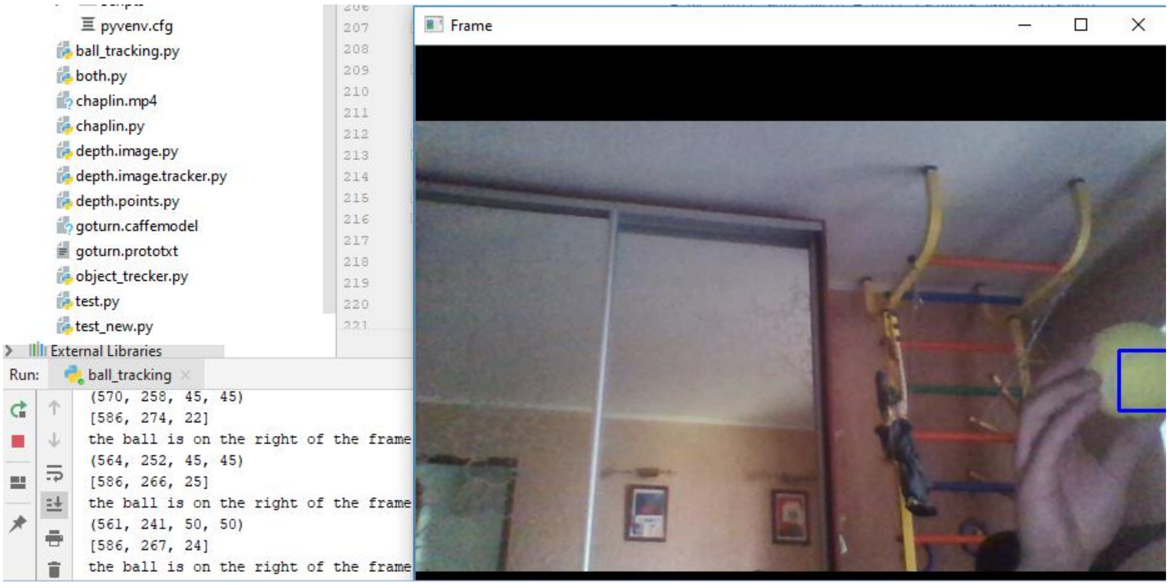
Integrating a tracking algorithm increased performance significantly. Unfortunately, the program with tracking part was tested only on a personal laptop. It was not possible to test it with the robot because as it was mentioned in the Accessories part, the ODROID board with ARM architecture was used to control the robot. These trackers do not support the ARM architecture. Therefore, in order to implement the tracking part with Pioneer 3-DX, the ODROID board has to be replaced with a PC.

The set of images below presents the results of object tracking with KCF tracker. The detection works under partial occlusion, with different distances to the object and when the object moves fast. The program proceeds the code quite fast and ensures more than 30 FPS. On the left bottom part of the images, the program outputs the coordinate of a detected ball (x, y, radius), the position of the ball in the frame, and the coordinates of a boundary box from tracking algorithm(x, y, height, width).



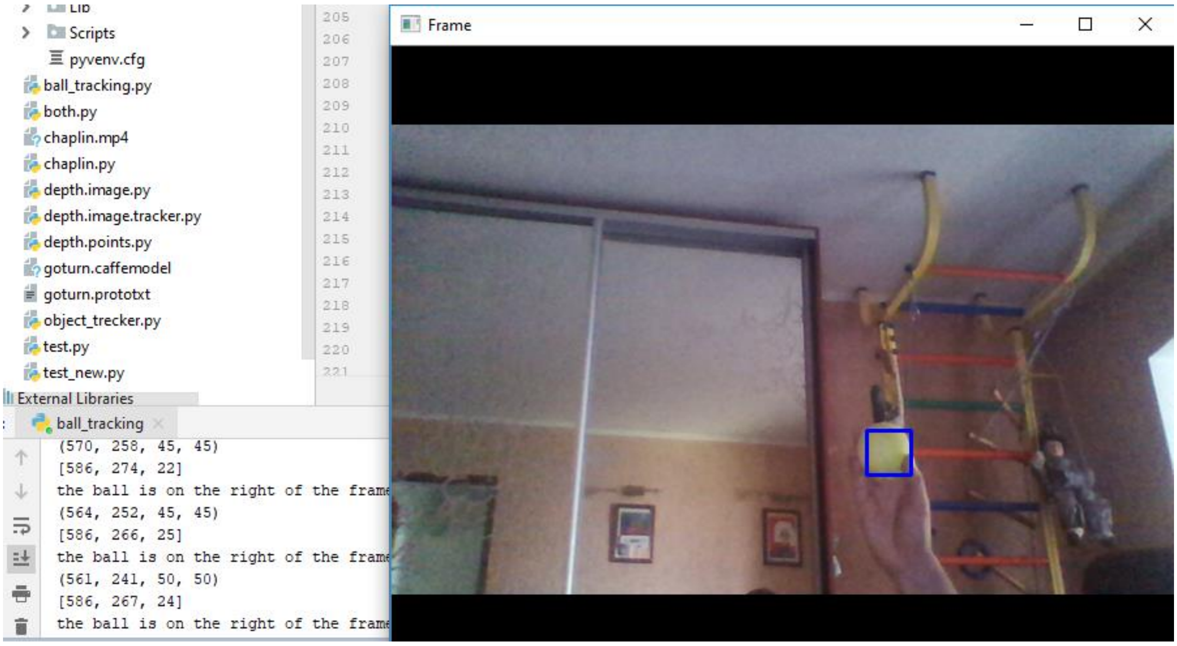
```
(192, 383, 21, 21)
[198, 395, 10]
the ball is on the left of the frame
(188, 385, 21, 21)
[104, 354, 41]
the ball is on the left of the frame
(63, 313, 83, 83)
[176, 303, 48]
the ball is on the left of the frame
(128, 255, 96, 96)
```

Figure 5.18 Tennis ball detection with KCF tracker



```
(570, 258, 45, 45)
[586, 274, 22]
the ball is on the right of the frame
(564, 252, 45, 45)
[586, 266, 25]
the ball is on the right of the frame
(561, 241, 50, 50)
[586, 267, 24]
the ball is on the right of the frame
```

Figure 5.19 Tennis ball detection with KCF tracker under partial occlusion



```
(570, 258, 45, 45)
[586, 274, 22]
the ball is on the right of the fram
(564, 252, 45, 45)
[586, 266, 25]
the ball is on the right of the fram
(561, 241, 50, 50)
[586, 267, 24]
the ball is on the right of the fram
```

Figure 5.20 Tennis ball tracking with KCF tracker. The ball is moved away from a camera and the tracker tracks it without any problems.

In order to see the difference between detection and tracking, the boundary boxes are displayed with different colors. The program outputs the boundary box from tracker with blue line. The boundary from color detection part is displayed with a yellow color. It was difficult to capture the detection because it appears for a sec and then the tracker starts working.

5.3 Depth calculation with Astra depth sensor

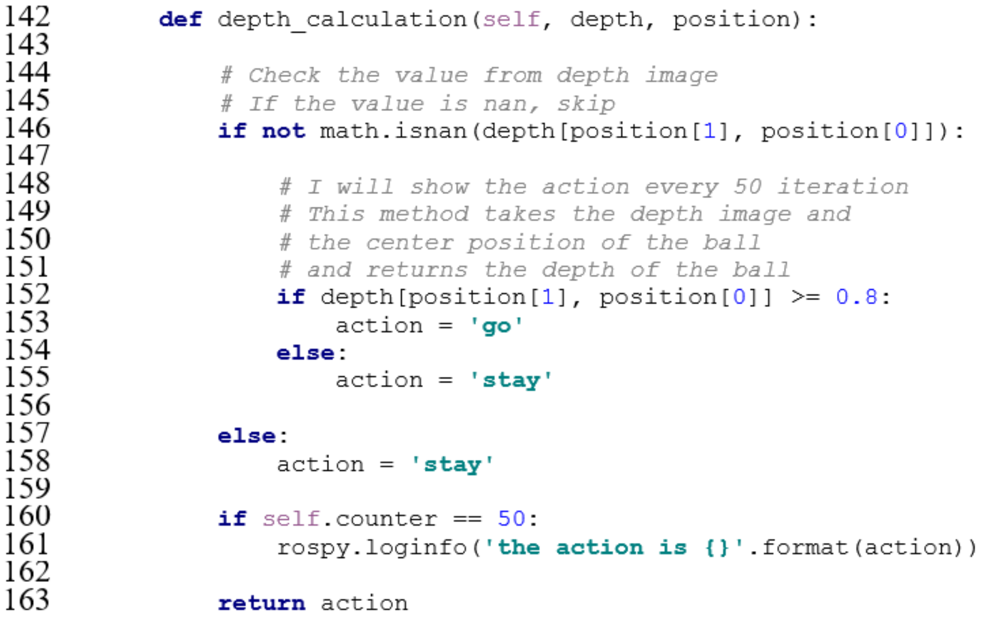
The depth sensor is one the best devices for mechatronic projects where the distance has to be calculated. The Astra depth sensor is built into a camera and has a quite simple working principle. The sensor sends a massive number of laser beams, and then the specific detector detects them. The main point is to estimate the distance between the two beams. It is simple to understand that if the object is close to the sensor, the distance between two beams on that object will be small. Otherwise, if the object is quite far from the camera, then the distance between two beams which have reached this object will be larger. The sensor has the function inside that calculates the distance to the object based on the distance between two neighborhood beams. The main disadvantage of this sensor is that sometimes the laser beams may be badly reflected from the object and the sensor will not be able to detect them. Also, the laser beam cannot be reflected from metals. It makes the distance calculating for metal object impossible.

Distance calculation is as crucial as the position calculation in this project. The program sends messages with the linear and angular speed. The angular motion depends on the object position in the frame, and the linear motion depends on how far is the object. There are a lot of methods on how the distance can be estimated. For example, the distance can be estimated even without a depth sensor by calculating the number of pixels of a specific color. Moreover, the speed is not essential, the main purpose is to determinate whether the object is moving away or not. The distance to the objects which are in front of the robot can be also estimated by using the robots sonars. In this project, in order to get the distance to the object, two methods were performed. The first stands for getting the distance from /depth/image topic. The executable file receives the depth image with the same size as the BGR image. Each pixel in that image represents the distance in meters. Another option is calculating the distance from the point cloud. Both of these methods will be described in the following sections.

5.3.1 Depth calculation from /depth/image topic

The easiest way to get the distances is to get the data from /depth/image topic. The Astra camera node creates this topic. The node publishes the information as an image where each pixel represents the distance to the object on that image in meters. Depth image topic updates less often than a topic with BGR image. In order to synchronize both of topics, the message_filters package, which was described in ROS related chapter, has been used. Additionally, the data conversion should be performed by using CV_Bridge package. How this is done can be observed in the figure 4.1 line 213.

The conversion function for depth image takes two arguments, like the BGR conversion. The first one is the input ROS message. The second argument can be either ‘32FC1’ or ‘passthrough’. It will be the same because ‘32FC1’ is the format where every pixel value is stored as one channel floating point, which is exactly what will be received if the data is passed with ‘passthrough’ option. Since the object is detected and the position is known, the center point of the boundary box can be used to check the depth in that point. The fragment of the code below presents the method which takes the depth image and the coordinate of the detected object and returns the command to the robot. It was set that if the distance is more than 0.8 meter, then the robot should move. Otherwise, it should stay still. This value has been set randomly just to test the program. It can be adjusted for the developers’ needs. The counter (line 160), which is used in a lot of methods of the machine vision class, is added to display the detection results every 50 iteration. Otherwise, there will be to many log outputs in the terminal and it will be hard to read. The counter value is the same for each method in the class because it is stored into the class variable. It allows to show the detection outputs like position and depth at the same time.



```
142     def depth_calculation(self, depth, position):
143 
144         # Check the value from depth image
145         # If the value is nan, skip
146         if not math.isnan(depth[position[1], position[0]]):
147 
148             # I will show the action every 50 iteration
149             # This method takes the depth image and
150             # the center position of the ball
151             # and returns the depth of the ball
152             if depth[position[1], position[0]] >= 0.8:
153                 action = 'go'
154             else:
155                 action = 'stay'
156 
157         else:
158             action = 'stay'
159 
160         if self.counter == 50:
161             rospy.loginfo('the action is {}'.format(action))
162 
163         return action
```

Figure 5.21 The function which takes the depth image and object position and return the action which must performed based on the depth data

Additionally, one more line of code should be added to the depth_calculation method. This line of code will check the depth value for Nan (line 146). Nan output may appear if the depth sensor could not read the depth value for some reason. In case Nan has detected, the command for the robot will be ‘stay’. This approach is very simple and does not require any additional calculation or functions compare to the second method.

5.3.2 Depth calculation from Point Cloud

The second method is much more sophisticated and in my opinion this method is used under the coverage of the first method. The point cloud data structure has to be understood firstly in order to understand the second method. This data structure is getting more and more popular nowadays. It can be used, for example, if the developer wants to eliminate a background. In the point cloud data structure, each pixel contains at least three values which are (x,y,z) coordinates. In that array, z value is the depth value which represents how far is the object, x and y coordinates represent the position of the pixel on the frame relative to the camera. By using a point cloud data structure, the surface in front of the camera can be presented. So, now it is easy to understand that the developer can eliminate the background by thresholding the depth value.

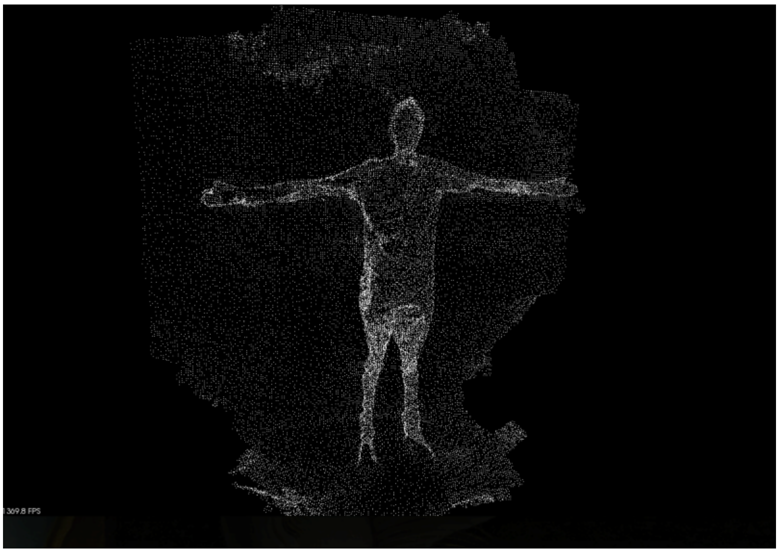
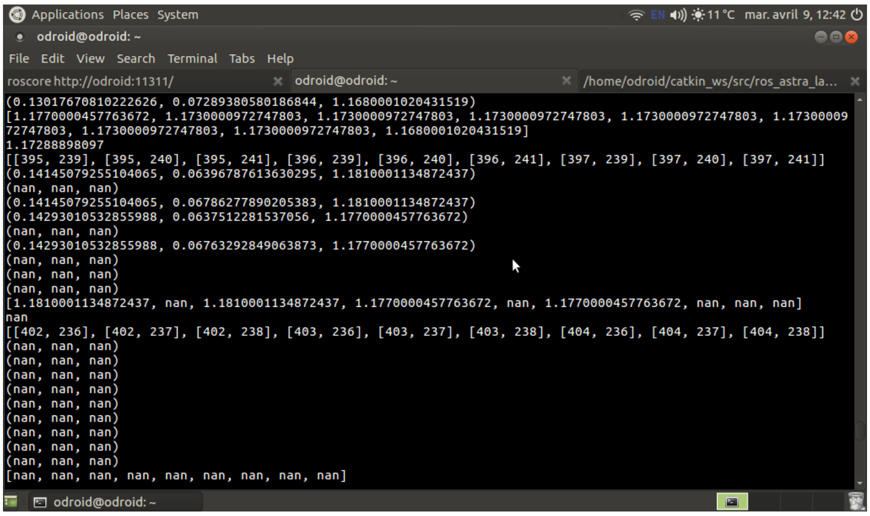


Figure 5.22 Representation of the pint cloud data structure (22)

While using that method with point cloud data structure, the only z value is required. However, after several experiments, it became clear that it is not possible to achieve robust depth calculation by using only one pixel. It is because the sensor does not always detect the laser and sometimes it returns the Nan value. There are many reasons why it could happen; one of them is bad beam reflection. In order to get a stable depth calculation, instead of one pixel, a kernel of pixels should be used.


Applications Places System
EN 11 °C mar. avril 9, 12:42
odroid@odroid: ~
File Edit View Search Terminal Tabs Help
roscore http://odroid:11311/
odroid@odroid: ~
/home/odroid/catkin_ws/src/ros_astra_la...
odroid@odroid: ~

```
(0.13017670810222626, 0.07289380580186844, 1.1680001020431519)
[1.1770000457763672, 1.1730000972747803, 1.1730000972747803, 1.1730000972747803, 1.1730000972747803, 1.17300009
72747803, 1.1730000972747803, 1.1730000972747803, 1.1680001020431519]
1.17288898097
[[395, 239], [395, 240], [395, 241], [396, 239], [396, 240], [396, 241], [397, 239], [397, 240], [397, 241]]
(0.14145079255104065, 0.06396787613630295, 1.1810001134872437)
(nan, nan, nan)
(0.14145079255104065, 0.06786277890205383, 1.1810001134872437)
(0.14293010532855988, 0.0637512281537056, 1.1770000457763672)
(nan, nan, nan)
(0.14293010532855988, 0.06763292849063873, 1.1770000457763672)
(nan, nan, nan)
(nan, nan, nan)
(nan, nan, nan)
[1.1810001134872437, nan, 1.1810001134872437, 1.1770000457763672, nan, 1.1770000457763672, nan, nan, nan]
nan
[[402, 236], [402, 237], [402, 238], [403, 236], [403, 237], [403, 238], [404, 236], [404, 237], [404, 238]]
(nan, nan, nan)
(nan, nan, nan)
(nan, nan, nan)
(nan, nan, nan)
(nan, nan, nan)
(nan, nan, nan)
(nan, nan, nan)
(nan, nan, nan)
(nan, nan, nan)
[nan, nan, nan, nan, nan, nan, nan, nan, nan]
```

Figure 5.23 This screenshot presents how often the sensor cannot calculate the distance to the object by using only one pixel. The value is got from a point cloud message. Test was performed with a tennis ball.

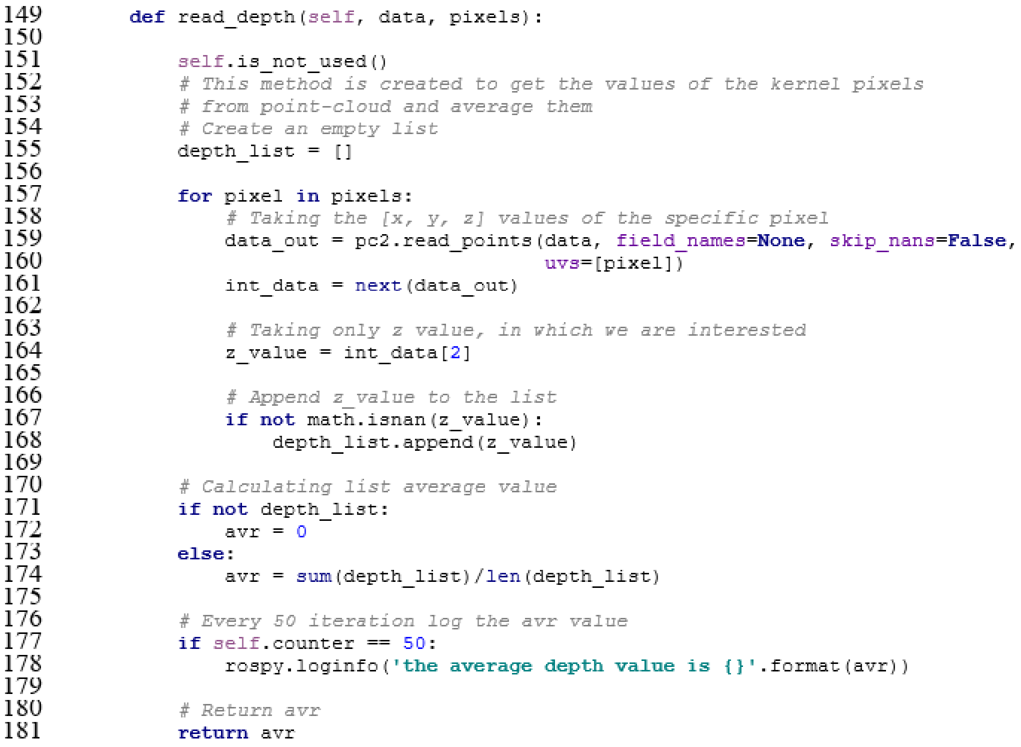
So, the method which creates the kernel by taking only the center pixel was prepared (line 121). The method takes the coordinate of a pixel as an input and returns the 3x3 kernel with the input pixel in the middle. This kernel is represented as a list (line 129), which subsequently passing to another method which is described on the next page. This method firstly defines the coordinates of the first pixel in the kernel (line 132 and 133) and then goes one by one adding the next pixels to the list by using FOR loop structure (line 136, 138). The for loop takes three pixels from the first row, then from second and third. The order is not essential in this situation.

```
121     def array_of_pixels(self, position):
122 
123         self.is_not_used()
124         # This method is created to get list of pixels of the 3x3 kernel.
125         # Subsequently we will use that kernel
126         # in order to get the depth values and average them
127 
128         # Reset the list of pixel
129         list_of_pixels = []
130 
131         # Take the position of start pixel for 3x3 kernel
132         start_x = position[0] - 1
133         start_y = position[1] - 1
134 
135         # Create a loop to go over the pixels, and append them to the array
136         for x in range(3):
137             x_pos = start_x + x
138             for y in range(3):
139                 y_pos = start_y + y
140                 # creating pixel position
141                 pixel = [x_pos, y_pos]

142 
143                  # Append pixel to the list of pixels
144                  list_of_pixels.append(pixel)
145 
146          # Return list of pixels
147          return list_of_pixels
148 
```

Figure 5.24 Shows the array_of_pixels method, which takes the center pixel as an input and outputs 3x3 kernel

The second function is a depth calculator. It should be noted that the point cloud data structure has to be read by using specific library pc2 in Python and its method read_points (line 159). This method converts the point cloud to the iterator. The iterator returns the values one by one when the program subsequently calls it by using next() (line 161) function in Python. It allows saving lots of memory compared to the list structure where all values are saved together. The read_points method also allows the developer to specify the pixel which he wants to retrieve by adding the specific argument uvs. The developer can also skip nan values or retrieve only a specific value from a pixel by applying additional arguments. It makes point cloud data treatment easy and convenient. In the code below, the method, read_depth (line 149), which retrieves all pixels in the predefined kernel by using a loop (line 157), is presented. It also checks the value for nan (line 167), and if it is not a Nan, it adds the value to another list (line number 168). The list mean value is calculated (line 174) at the end and returned as an output of the method (line 181).

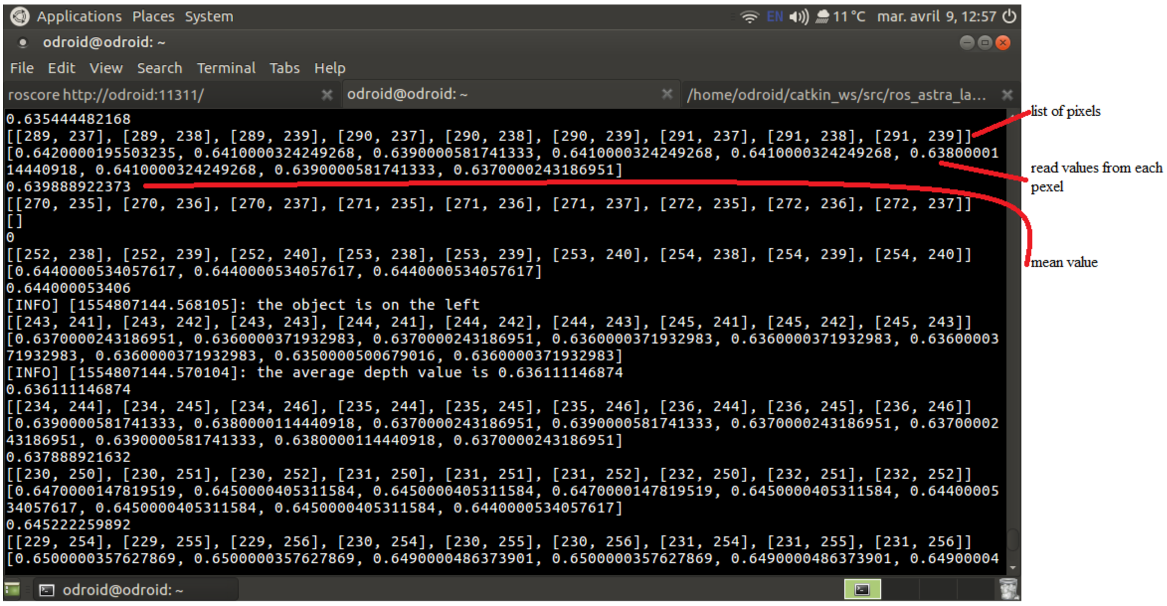
Afterwards, the average depth is passed to the robot controlling method like in the previous section where the program defines what command should be sent to the Pioneer 3-DX robot (Figure 5.19). The only difference that there is no need to check the input value for nan because it is already done in the read_depth method. This approach increases the robustness of the depth calculation due to the fact that it checks several points of the object.



```
149         def read_depth(self, data, pixels):
150 
151             self.is_not_used()
152             # This method is created to get the values of the kernel pixels
153             # from point-cloud and average them
154             # Create an empty list
155             depth_list = []
156 
157             for pixel in pixels:
158                 # Taking the [x, y, z] values of the specific pixel
159                 data_out = pc2.read_points(data, field_names=None, skip_nans=False,
160                                            uvs=[pixel])
161                 int_data = next(data_out)
162 
163                 # Taking only z value, in which we are interested
164                 z_value = int_data[2]
165 
166                 # Append z_value to the list
167                 if not math.isnan(z_value):
168                     depth_list.append(z_value)
169 
170             # Calculating list average value
171             if not depth_list:
172                 avr = 0
173             else:
174                 avr = sum(depth_list)/len(depth_list)
175 
176             # Every 50 iteration log the avr value
177             if self.counter == 50:
178                 rospy.loginfo('the average depth value is {}'.format(avr))
179 
180             # Return avr
181             return avr
```

Figure 5.25 This figure presents the read_depth method which retrieving the z values for specific pixels from a point cloud and calculates the average depth.

The results of that approach are presented in the figure below.


Applications Places System
EN 11 °C mar. avril 9, 12:57
odroid@odroid: ~
File Edit View Search Terminal Tabs Help
roscore http://odroid:11311/
odroid@odroid: ~
/home/odroid/catkin_ws/src/ros_astra_la...
odroid@odroid: ~
list of pixels
read values from each pexel
mean value

```
0.635444482168
[[289, 237], [289, 238], [289, 239], [290, 237], [290, 238], [290, 239], [291, 237], [291, 238], [291, 239]]
[0.6420000195503235, 0.6410000324249268, 0.6390000581741333, 0.6410000324249268, 0.6410000324249268, 0.63800001
14440918, 0.6410000324249268, 0.6390000581741333, 0.6370000243186951]
0.639888922373
[[270, 235], [270, 236], [270, 237], [271, 235], [271, 236], [271, 237], [272, 235], [272, 236], [272, 237]]
[]
0
[[252, 238], [252, 239], [252, 240], [253, 238], [253, 239], [253, 240], [254, 238], [254, 239], [254, 240]]
[0.6440000534057617, 0.6440000534057617, 0.6440000534057617]
0.644000053406
[INFO] [1554807144.568105]: the object is on the left
[[243, 241], [243, 242], [243, 243], [244, 241], [244, 242], [244, 243], [245, 241], [245, 242], [245, 243]]
[0.6370000243186951, 0.6360000371932983, 0.6370000243186951, 0.6360000371932983, 0.6360000371932983, 0.63600003
71932983, 0.6360000371932983, 0.6350000500679016, 0.6360000371932983]
[INFO] [1554807144.570104]: the average depth value is 0.636111146874
0.636111146874
[[234, 244], [234, 245], [234, 246], [235, 244], [235, 245], [235, 246], [236, 244], [236, 245], [236, 246]]
[0.6390000581741333, 0.6380000114440918, 0.6370000243186951, 0.6390000581741333, 0.6370000243186951, 0.63700002
43186951, 0.6390000581741333, 0.6380000114440918, 0.6370000243186951]
0.637888921632
[[230, 250], [230, 251], [230, 252], [231, 250], [231, 251], [231, 252], [232, 250], [232, 251], [232, 252]]
[0.6470000147819519, 0.6450000405311584, 0.6450000405311584, 0.6470000147819519, 0.6450000405311584, 0.64400005
34057617, 0.6450000405311584, 0.6450000405311584, 0.6440000534057617]
0.645222259892
[[229, 254], [229, 255], [229, 256], [230, 254], [230, 255], [230, 256], [231, 254], [231, 255], [231, 256]]
[0.6500000357627869, 0.6500000357627869, 0.6490000486373901, 0.6500000357627869, 0.6490000486373901, 0.64900004
```

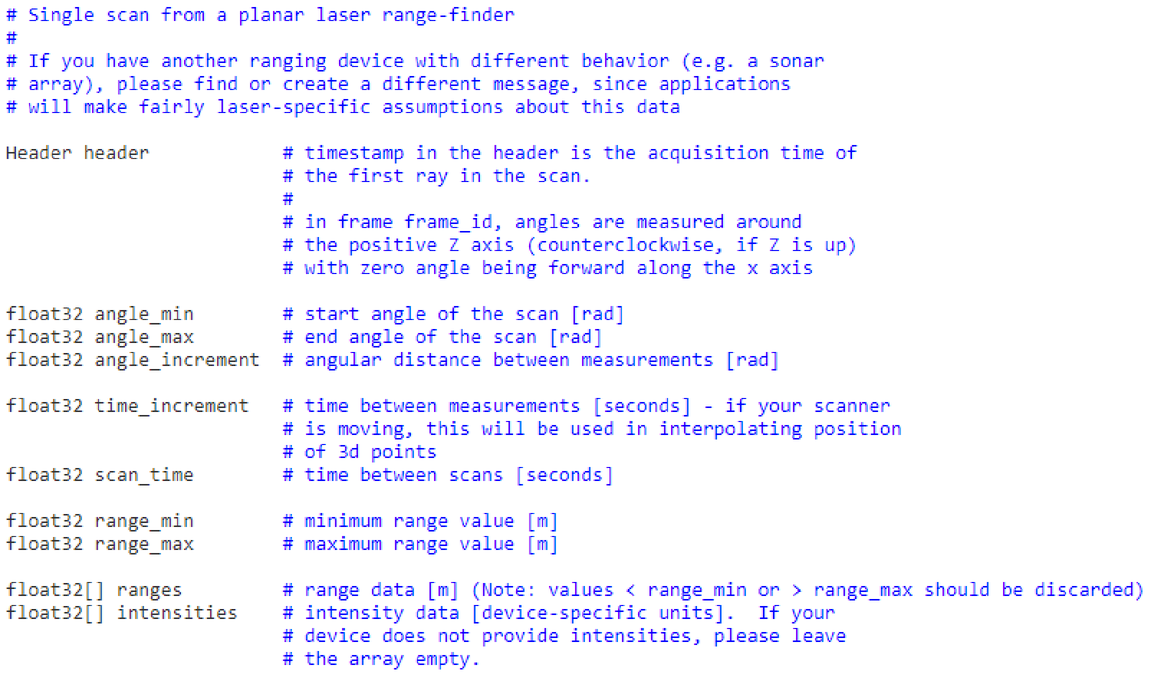
Figure 5.26 This screenshot shows the results of depth calculation. It outputs the list of pixels of a kernel, z values of each pixel (if it is not nan), and the mean value. The size of the list with z values changes.

The program displays three important variables. The first one is the list of pixels which is passed to the read_depth function. This variable is described as a list which contains nine other lists. Each of them includes the coordinates (x, y) of the pixel. The second variable is the depth values which was read by using the list of pixels. It is noticeable that the size of that list is not constant. It is because the Nan values might be read sometimes. The last variable presents the average depth value in which we are interested in the end. The full program code which contains this approach of depth calculation is presented in the Appendix B.

6. MAPPING WITH RPLIDAR

The last but not the least part of the conducted work is related to the mapping by using RPLIDAR. This device, which is mounted on the top of the robot, is the perfect tool which can be used in Mechatronics projects to detect the obstacles around the robot. It is a suitable sensor for indoor SLAM applications. Detecting all objects around the robot can be directly called mapping. It can be called like this because the graphical visualization is not useful for the robot. The robot only needs to know the values of distances to the objects and their sizes. The RPLIDAR used is a quite cheap and robust device. The price is around 400$ which is not a high price for such a good detection. This lidar provides 2D detection, and this is the main drawback. The motor provides the 360-degree rotation motion in one space. So, the position of the lidar depends on at what height the developer wants to perform the detection. The working principle and specification of the lidar were mentioned in the Accessories chapter. This chapter is about how can the device be used in ROS. Due to the fact that this lidar is quite popular for mapping purposes, there are a lot of prepared packages on the Internet for the lidar. The main package is rplidar. (23)

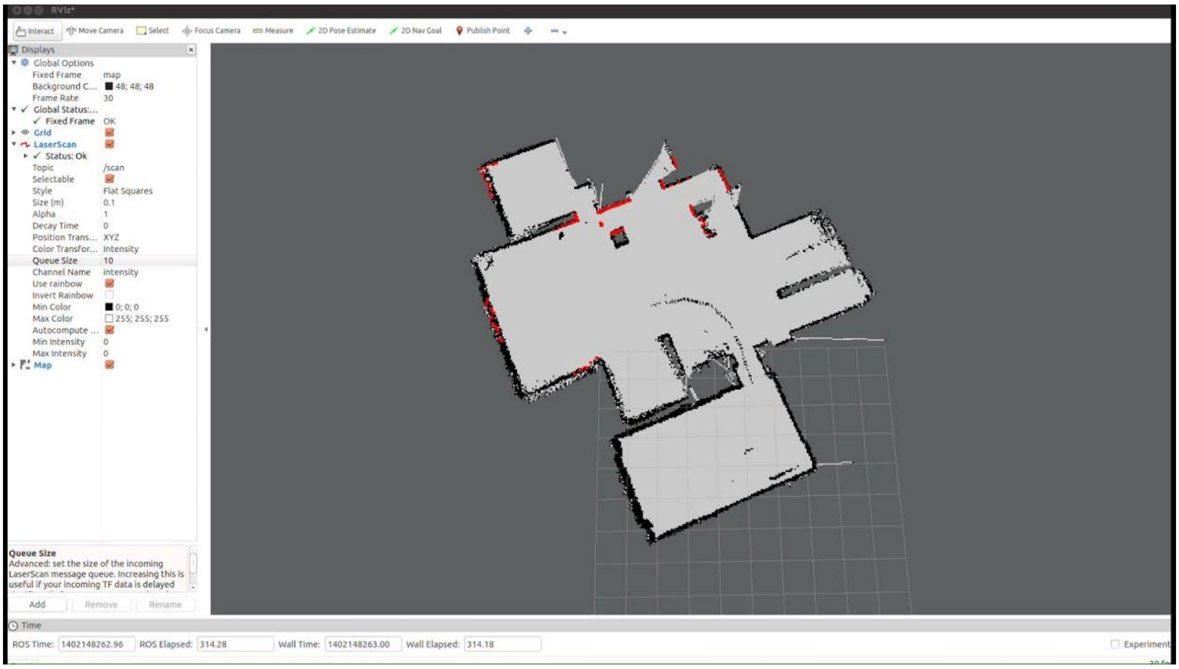
The rplidar node reads an RPLIDAR raw scan data using RPLIDAR's SDK and converts it to ROS message. The node publishes the data sensor_msgs/LaserScan message. This message includes information about start angle, end angle, angle increment, ranges, scan time and so on. The angle is presented in radians and the ranges are presented in meters.



```
# Single scan from a planar laser range-finder
#
# If you have another ranging device with different behavior (e.g. a sonar
# array), please find or create a different message, since applications
# will make fairly laser-specific assumptions about this data

Header header            # timestamp in the header is the acquisition time of
                         # the first ray in the scan.
                         #
                         # in frame frame_id, angles are measured around
                         # the positive Z axis (counterclockwise, if Z is up)
                         # with zero angle being forward along the x axis

float32 angle_min        # start angle of the scan [rad]
float32 angle_max        # end angle of the scan [rad]
float32 angle_increment  # angular distance between measurements [rad]

float32 time_increment   # time between measurements [seconds] - if your scanner
                         # is moving, this will be used in interpolating position
                         # of 3d points
float32 scan_time        # time between scans [seconds]

float32 range_min        # minimum range value [m]
float32 range_max        # maximum range value [m]

float32[] ranges         # range data [m] (Note: values < range_min or > range_max should be discarded)
float32[] intensities    # intensity data [device-specific units].  If your
                         # device does not provide intensities, please leave
                         # the array empty.
```

Figure 6.1 Structure of geometry_msgs/LaserScan message (6)

So, as it can be seen in the figure 5.1 ranges data component is presented as a list. Angles and ranges are connected, so the developer can easily estimate the location of the obstacles knowing the start angle. That data can be visualized by using rviz tool integrated into ROS which was described in the ROS related chapter. These ranges in meter can be used in conjunction with any detection algorithm. For example, YOLO detection can be performed in order to recognize the objects around and to understand their locations. Then the data from scanning can be used to estimate the distances between each object. All that information can be stored into the dictionary in Python where each key will be the name of the object and the value will be its position. As a result, the developer achieves quite a good map for the robot. Additionally, rplidar node also provides two services to stop and start the motor. There are a lot of already existed packages for obstacle avoiding in ROS developed based on the sensor_msgs/LaserScan data. The colossal work was conducted by Mats Håkon Follestad where the good explanation of how RPDILAR can be used is presented. (24)



```
▾ Global Options
    Fixed Frame          map
    Background C...      ■ 48; 48; 48
    Frame Rate           30
▾ ✓ Global Status:...
    ✓ Fixed Frame        OK
▸ Grid                   ☑
▾ LaserScan              ☑
  ▸ ✓ Status: Ok
    Topic                /scan
    Selectable           ☑
    Style                Flat Squares
    Size (m)             0.1
    Alpha                1
    Decay Time           0
    Position Trans...    XYZ
    Color Transfor...    Intensity
    Queue Size           10
    Channel Name         intensity
    Use rainbow          ☑
    Invert Rainbow       □
    Min Color            ■ 0; 0; 0
    Max Color            □ 255; 255; 255
    Autocompute ...      ☑
    Min Intensity        0
    Max Intensity        0
▸ Map                    ☑
```

Queue Size
Advanced: set the size of the incoming LaserScan message queue. Increasing this is useful if your incoming TF data is delayed

ROS Time: 1402148262.96 ROS Elapsed: 314.28 Wall Time: 1402148263.00 Wall Elapsed: 314.18

Figure 6.2 The visualization of ranges from RPLIDAR in the rviz plugin (6)

The good thing is that the RPLIDAR’s SDK can be launched at all operational systems. Unfortunately, the lidar which is attached to the top of the robot had some problems with USB adapter. This adapter is needed because initially lidar uses a COM port communication. Due to this issue another RPLIDAR was tested separately.

7. CONCLUSION

Initially, the work was devoted to creating the cobot system. The system had to detect the object and track that object. However, during the work, a lot of additional options and developing perspectives have been discovered. In the beginning, only color detection was performed, but at the end of the work, the much more sophisticated detection algorithms were considered. The color detection or pattern matching were good detection algorithms ten years ago. Nowadays, with new state-of-the-art neural network technologies, such a task like an object tracking can be implemented with much better performance. Moreover, the detection by using YOLO detector is not limited to a number of objects. Contrarily, the color detection is restricted by the color and the pattern matching is restricted by the pattern. This allows using the new detection technologies in developing sophisticated applications for autonomous vehicles which was not possible before or had very low robustness. However, in case of single object tracking, the old methods still can be useful. They intended to help the network to understand which one out of a list of similar objects the application should track. In conjunction with neural networks, such applications can achieve an excellent result. The essential part of the work was also the robot motion control in ROS. Studying ROS and acquiring knowledge allows understanding ROS working principle completely. Easily implemented and very convenient communications methods allow to establish communication between several executable files regardless of which programming languages were used to write those files. On this stage, the performed executable files can be used with several robots, and that is the additional benefit of the ROS.

Additionally, the ways of how the project can be improved are defined. The project can be continuously developed in a different direction. By using the PC instead of the single-board computer, the tracking performance can be increased. It should be noticed that with powerful PC the YOLOv3 detector can be trained for a specific object. This neural network which is trained to detect specific object can be subsequently used for detection and tracking in conjunction with other methods. Another direction of developing is performing the map for autonomous vehicles. With these technologies, the operator is not required at all. In that case, by using GPS systems, the robot will know the destination, and by using the detection system, the robot will avoid the obstacles.

REFERENCES

1. Ferrari, M. Tarokh* and P. *Case Study: Robotic Person Following Using Fuzzy Control and Image Segmentation.* San Diego, USA : s.n., 2003.

2. Jun Miura, Junji Satake, Masaya Chiba. *Development of a Personal Following Robot and Its Experimental Evaluation.* Ottawa, Canada : s.n., 2010.

3. Hiroshi Takemura, Zentaro Nemoto, Keita Ito and Hiroshi Mizoguchi. *Development of Vision Based Person Following Module for Mobile Robots in RT-Middleware.* Tokyo, Japan : s.n., 2010.

4. *Darknet official Web site.* [Online] https://pjreddie.com/darknet/.

5. Bao Xin Chen, Raghavender Sahdev, and John K. Tsotsos. *Integrating Stereo Vision with a CNN Tracker for a Person-Following Robot.* York University, Canada : s.n.

6. *ROS documentation.* [Online] https://www.ros.org/.

7. *OpenCV documentation.* [Online] https://opencv.org/.

8. *Hardkernel.* [Online] https://www.hardkernel.com/shop/odroid-xu4-special-price/.

9. ODROID XU-4 single board manual. *Magazine Odroid.* [Online] https://magazine.odroid.com/wp-content/uploads/odroid-xu4-user-manual.pdf.

10. Pioneer 3-DX operation manual. [Online] https://www.inf.ufrgs.br/~prestes/Courses/Robotics/manual_pioneer.pdf.

11. *ORBBEC Astra cameras official Web site.* [Online] https://orbbec3d.com/product-astra-pro/.

12. Technical specification. *Astra wiki.* [Online] https://astra-wiki.readthedocs.io/en/latest/about.html.

13. RPLIDAR A2 version. *RPLIDAR official Web Site.* [Online] https://www.slamtec.com/en/Lidar/A2.

14. Actionlib package description. *ROS official documentation.* [Online] http://wiki.ros.org/actionlib.

15. Message_filters package description. *ROS official documentation.* [Online] http://wiki.ros.org/message_filters.

16. RosAria package description. *ROS official documentation.* [Online] [http://wiki.ros.org/ROSARIA.

17. *Robocraft.* [Online] http://robocraft.ru/blog/computervision/402.html.

18. *YOLOv3: An Incremental Improvement.* Joseph Redmon, Ali Farhadi.

19. List of COCO objects. *GitHub.* [Online] https://github.com/arunponnusamy/object-detection-opencv/blob/master/yolov3.txt.

20. NumPy . *Scipy official documentation.* [Online] https://docs.scipy.org/doc/numpy/.

21. *Robust Object Tracking with Online Multiple Instance Learning.* Boris Babenko, Student Member, IEEE, Ming-Hsuan Yang, Senior Member, IEEE.

22. Point cloud — Post-processing of point cloud data. [Online] https://www.prototechsolutions.com/point-cloud-visualization/].

23. RPLIDAR package description. *ROS official documentation.* [Online] http://wiki.ros.org/rplidar.

24. Follestad, Mats Håkon. *Autonomous Path-Planning and* -. Trondheim : s.n., 2017.

APPENDIX A

Here is the code which includes object color detection, object tracking, depth calculating from depth image and subsequent ROS communication is presented. This code was tested without ROS communication on a personal laptop. In order to test it with Pioneer 3-DX robot, the ODROID single board must be replaced with standard PC.

```python
#! /usr/bin/env python

# import necessary libraries
import cv2
import numpy as np
import argparse
import rospy
import math
import message_filters
from cv_bridge import CvBridge, CvBridgeError
from sensor_msgs.msg import Image
from geometry_msgs.msg import Twist


# The program is controlling the robot in real-time by tracking the ball.
# Depends on the ball position on the screen and the data from depth sensor,
# the program sends signals to the robot motion controller.
# For the program execution Pioneer 3-DX robot, Astra RGBD camera were used.


class Motion(object):

    # This class responsible for sending commands to the robot
    # by using ROS topic /RosAria/cmd_vel
    # In the topic the linear and angular velocity are passing

    def __init__(self):
        self.motion = Twist()
        self.pub = rospy.Publisher('/RosAria/cmd_vel', Twist, queue_size=1)

    def left_go(self):
        self.motion.angular.z = .2
        self.motion.linear.x = .2
        self.pub.publish(self.motion)

    def left_stay(self):
        self.motion.angular.z = .2
        self.motion.linear.x = 0
        self.pub.publish(self.motion)

    def right_go(self):
        self.motion.angular.z = -.2
        self.motion.linear.x = .2
        self.pub.publish(self.motion)

    def right_stay(self):
        self.motion.angular.z = -.2
        self.motion.linear.x = 0
        self.pub.publish(self.motion)

    def center_go(self):
        self.motion.angular.z = 0
        self.motion.linear.x = .2
        self.pub.publish(self.motion)

    def center_stay(self):
        self.motion.angular.z = 0
        self.motion.linear.x = 0
        self.pub.publish(self.motion)


class BallTracker(object):

    # This class is responsible for the object detection and tracking.
    # It is also passing the output to the motion control class

    # Initiate class variables in the constructor
    def __init__(self, hsv_array):

        # Variables
        self.counter = 0
        self.position = []
        self.rectangle = ()
        self.lower_bound = np.array(hsv_array[0:3])
        self.upper_bound = np.array(hsv_array[3:6])
        self.image = None
        self.tracking = False
        self.track = None
        self.controller = Motion()
        self.bridge = CvBridge()

        # Next two lines create subscribers from image and depth_image topics
        # which will be synchronized subsequently
        # in the ApproximateTimeSynchronizer,
        # in order to analyze frames with the same timestamps
        image_sub = message_filters.Subscriber("/camera/rgb/image_raw", Image)
        depth_sub = message_filters.Subscriber("/camera/depth/image", Image)

        # Creating time synchronizer with the queue_size = 1
        # and slop = 1, initiate a callback
        self.ts = message_filters.ApproximateTimeSynchronizer([image_sub,
                                                               depth_sub],
                                                              1, 1)
        self.ts.registerCallback(self.callback)

        tracker_types = ['BOOSTING', 'MIL', 'KCF', 'TLD', 'MEDIANFLOW',
                         'GOTURN', 'CSRT']
        tracker_type = tracker_types[0]

        # Creating a tracker
        if tracker_type == 'BOOSTING':
            self.tracker = cv2.TrackerBoosting_create()
        if tracker_type == 'MIL':
            self.tracker = cv2.TrackerMIL_create()
        if tracker_type == 'KCF':
            self.tracker = cv2.TrackerKCF_create()
        if tracker_type == 'TLD':
            self.tracker = cv2.TrackerTLD_create()
        if tracker_type == 'MEDIANFLOW':
            self.tracker = cv2.TrackerMedianFlow_create()
        if tracker_type == 'GOTURN':
            self.tracker = cv2.TrackerGOTURN_create()
        if tracker_type == "CSRT":
            self.tracker = cv2.TrackerCSRT_create()

        self.prime_tracker = self.tracker

    def is_not_used(self):
        pass

    def position_calculator(self, position):

        # I will show the position every 50 iteration,
        # for that purpose I need a counter
        # This method calculate the position of the object, log it and return
        # if 320 + position[2] >= position[0] >= 320 - position[2]:
        if 440 >= position[0] >= 220:
            attitude = 'center'

            # 320 - position[2]
        elif position[0] < 220:
            attitude = 'left'

        else:
            attitude = 'right'

        if self.counter == 50:
            rospy.loginfo('the object is on the {}'.format(attitude))

        return attitude

    def depth_calculation(self, depth, position):

        # Check the value from depth image
        # If the value is nan, skip
        if not math.isnan(depth[position[1], position[0]]):

            # I will show the action every 50 iteration
            # This method takes the depth image and
            # the center position of the ball
            # and returns the depth of the ball
            if depth[position[1], position[0]] >= 0.8:
                action = 'go'
            else:
                action = 'stay'

        else:
            action = 'stay'

        if self.counter == 50:
            rospy.loginfo('the action is {}'.format(action))

        return action

    def command_to_the_robot(self, attitude, action):

        # This method will make a decision about the robot motion
        # The robot moves straight and turns left
        if attitude == 'left' and action == 'go':
            self.controller.left_go()

        # The robot only turns left
        elif attitude == 'left' and action == 'stay':
            self.controller.left_stay()

        # The robot moves straight and turns right
        elif attitude == 'right' and action == 'go':
            self.controller.right_go()

        # The robot only turns right
        elif attitude == 'right' and action == 'stay':
            self.controller.right_stay()

        # The robot moves straight
        elif attitude == 'center' and action == 'go':
            self.controller.center_go()

        # The robot stays on the same place
        elif attitude == 'center' and action == 'stay':
            self.controller.center_stay()

    def rectangle_calculation(self, position):

        self.is_not_used()

        x = position[0] - position[2]
        y = position[1] - position[2]

        xw = position[2]*2
        yw = xw

        rect = (x, y, xw, yw)

        return rect

    def callback(self, rgb_data, depth_data):

        # This method getting the data from both topics and analyze them
        try:
            # take the image as 'bgr8' format and the depth as '32FC1',
            # store them into variables
            image = self.bridge.imgmsg_to_cv2(rgb_data, "bgr8")
            depth_image = self.bridge.imgmsg_to_cv2(depth_data, "32FC1")
        except CvBridgeError as error:
            # If there is an error, log this error
            rospy.loginfo('the error: {}'.format(error))

        if not self.tracking:
            # Making preprocess of the frame
            # Applying GaussianBlur convolution kernel
            # to reduce the number the noise in the frame
            blurred = cv2.GaussianBlur(image, (11, 11), 0)
            # Changing BRG to HSV
            hsv = cv2.cvtColor(blurred, cv2.COLOR_BGR2HSV)
            # Applying the mask by using inRange function
            mask = cv2.inRange(hsv, self.lower_bound, self.upper_bound)
            # Now we firstly erode and the do opposite (dilate)
            # to remove small detections from the frame
            mask = cv2.erode(mask, None, iterations=2)
            mask = cv2.dilate(mask, None, iterations=2)

            # Looking for the contours, by using cv2.findContours function
            _, contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL,
                                              cv2.CHAIN_APPROX_SIMPLE)

            if contours:
                self.position = []
                # Going ahead and looking for the max contour in the contours
                c = max(contours, key=cv2.contourArea)
                # By using minEcnlosingCircle taking the coordinates
                # of the approximate center of the area
                ((x, y), radius) = cv2.minEnclosingCircle(c)
                # Append the data to new array,
                # we will use this array subsequently
                # in the def position calculation
                self.position.extend([int(x), int(y), int(radius)])

                # print(depth_image[self.position[0], self.position[1]])

                # If the object is big enough, it means that the operator
                # is close to the robot, go ahead
                if 80 > radius > 10:
                    # draw the circle and centroid on the frame,
                    cv2.circle(image, (int(x), int(y)), int(radius),
                               (0, 255, 255), 2)
                    # cv2.circle(image, (int(x), int(y)), 5, (0, 255, 255), -1)
                    self.rectangle = self.rectangle_calculation(self.position)
                    cv2.rectangle(image, (self.rectangle[0], self.rectangle[1]),
                                  (self.rectangle[0] + self.rectangle[2],
                                  self.rectangle[1] + self.rectangle[3]),
                                  (0, 255, 255), 2)

                    # Initialize the tracker with first frame and
                    # bounding box from detection part
                    self.tracker = self.prime_tracker
                    self.track = self.tracker.init(image, self.rectangle)
                    self.tracking = True

            else:

                # Print the msg to the operator
                if self.counter % 25 == 0:
                    rospy.loginfo("No contours have been detected")

                # Reset the counter
                if self.counter == 51:
                    self.counter = 0

        else:

            # Update tracker
            self.track, self.rectangle = self.tracker.update(image)

            # Draw bounding box
            if self.track:
                self.position = []

                # Tracking is okay, extract the coordinates of the
                # object and store them into self.position
                p1 = (int(self.rectangle[0]), int(self.rectangle[1]))
                p2 = (int(self.rectangle[0] + self.rectangle[2]),
                      int(self.rectangle[1] + self.rectangle[3]))

                cv2.rectangle(image, p1, p2, (255, 0, 0), 2, 1)

                self.position.extend([int(self.rectangle[0] +
                                          self.rectangle[2]/2),
                                      int(self.rectangle[1] +
                                          self.rectangle[3]/2)])

                if self.rectangle[2] >= 140:
                    rospy.loginfo('The boundary box of a ball is abnormally big')

                    # Resetting the tracking algorithm
                    self.tracking = False

                    # Stop the robot if there are no contours have been found
                    self.controller.center_stay()

                else:

                    # Run the position calculation method
                    attitude = self.position_calculator(self.position)

                    # Run the depth calculation method
                    depth = self.depth_calculation(depth_image, self.position)

                    # Next method stands for command sending to the robot
                    self.command_to_the_robot(attitude, depth)

                    # reset the self.counter
                    if self.counter == 51:
                        self.counter = 0

            else:
                # Tracking failure
                cv2.putText(image, "Tracking failure detected", (100, 80),
                            cv2.FONT_HERSHEY_SIMPLEX,
                            0.75, (0, 0, 255), 2)
                self.tracking = False

                # Stop the robot if there are no contours have been found
                self.controller.center_stay()

        # show the image
        self.counter += 1
        cv2.imshow('image', image)
        cv2.waitKey(1)


def arg_parser():

    # Taking an argument from command line, creating an arg parser
    parser = argparse.ArgumentParser(description=
                                     'To choose the way of defining HSV')

    # analyse an argument
    parser.add_argument("-o", "--option", type=str, required=True, metavar='',
                        help='1 - by typing the values, '
                             '2 - use default values', )
```

```
352     # store the argument in arg
353     arg = parser.parse_args()
354 
355     # return arg
356     return arg
357 
358 
359 def main():
360 
361     # Initiate the node named 'camera_parser'
362     rospy.init_node('camera_parser', anonymous=True)
363 
364     # Creating some local variables
365     condition = False
366     hsv_values = []
367 
368     # Choose the option for HSV settings
369     arg = arg_parser()
370     option = arg.option
371 
372     # Here we launched the program in two different ways,
373     # by choosing the default HSV values
374     # or typing them manually
375     if option == '1':
376         for i in ['MIN', 'MAX']:
377             for j in 'HSV':
378                 value = input('%s_%s value:' % (i, j))
379                 hsv_values.append(int(value))
380         # Condition True launched the BallTracking program, log HSV values.
381         # The same operations in the line 198-199
382         condition = True
383         rospy.loginfo('The HSV values are next: {}'.format(hsv_values))
384 
385     elif option == '2':
386         hsv_values = [29, 75, 6, 65, 255, 255]
387         condition = True
388         rospy.loginfo('The HSV values are next: {}'.format(hsv_values))
389 
390     if condition:
391         # Running ball tracking application
392         main_process = BallTracker(hsv_values)
393 
394         try:
395             rospy.spin()
396         except KeyboardInterrupt:
397             rospy.logtinfo('The error has been occurred, '
398                            'shutting down the program')
399 
400         cv2.destroyAllWindows()
401 
402 
403 # the program starts here, if the module have been launched
404 # by itself then the part of the code will run
405 if __name__ == '__main__':
406     main()
```

APPENDIX B

The code presented in Appendix B includes object color detection, depth calculation by using Point Cloud data structure, and ROS communication. This code was successfully tested with Pioneer 3-DX robot.

```
#! /usr/bin/env python

# import necessary libraries
import cv2
import numpy as np
import argparse
import rospy
import math
import message_filters
import sensor_msgs.point_cloud2 as pc2
from cv_bridge import CvBridge, CvBridgeError
from sensor_msgs.msg import Image
from geometry_msgs.msg import Twist
from sensor_msgs.msg import PointCloud2


# The program is controlling the robot in real-time by tracking the ball.
# Depends on the ball position on the screen and the data
# from depth sensor the program sends signals to the robot motion controller.
# For the program execution Pioneer 3-DX robot, Astra RGBD camera were used.


class Motion(object):

    # This class responsible for sending commands to the robot
    # by using ROS topic /RosAria/cmd_vel
    # In the topic the linear and angular velocity are passing

    def __init__(self):
        self.motion = Twist()
        self.pub = rospy.Publisher('/RosAria/cmd_vel', Twist, queue_size=1)

    def left_go(self):
        self.motion.angular.z = .2
        self.motion.linear.x = .2
        self.pub.publish(self.motion)

    def left_stay(self):
        self.motion.angular.z = .2
        self.motion.linear.x = 0
        self.pub.publish(self.motion)

    def right_go(self):
        self.motion.angular.z = -.2
        self.motion.linear.x = .2
        self.pub.publish(self.motion)

    def right_stay(self):
        self.motion.angular.z = -.2
        self.motion.linear.x = 0
        self.pub.publish(self.motion)

    def center_go(self):
        self.motion.angular.z = 0
        self.motion.linear.x = .2
        self.pub.publish(self.motion)

    def center_stay(self):
        self.motion.angular.z = 0
        self.motion.linear.x = 0
        self.pub.publish(self.motion)


class BallTracker(object):

    # This class is responsible for the object detection and tracking.
    # It is also passing the output to the
    # motion control class

    # Initiate class variables in the constructor
    def __init__(self, hsv_array):

        # Variables
        self.list_of_pixels = []
        self.position = []
        self.counter = 0
        self.lower_bound = np.array(hsv_array[0:3])
        self.upper_bound = np.array(hsv_array[3:6])
        self.image = None
        self.bridge = CvBridge()
        self.controller = Motion()

        # Next two lines create subscribers from image and
        # depth_image topics which will be synchronized subsequently
        # in the ApproximateTimeSynchronizer,
        # in order to analyze frames with the same timestamps
        image_sub = message_filters.Subscriber("/camera/rgb/image_raw", Image)
        depth_sub = message_filters.Subscriber("/camera/depth/points",
                                               PointCloud2)

        # Creating time synchronizer with the queue_size = 1
        # and slop = 1, initiate a callback
        self.ts = message_filters.ApproximateTimeSynchronizer([image_sub,
                                                               depth_sub],
                                                              10, 0.5)

        self.ts.registerCallback(self.callback)

    def is_not_used(self):
        pass

    def position_calculator(self, position):

        # I will show the position every 50 iteration, for that purpose
        # I need a counter
        # This method calculate the position of the object, log it and return
        if 320 + position[2] >= position[0] >= 320 - position[2]:
            attitude = 'center'

        elif position[0] < 320 - position[2]:
            attitude = 'left'

        else:
            attitude = 'right'

        if self.counter == 50:
            rospy.loginfo('the object is on the {}'.format(attitude))

        return attitude

    def array_of_pixels(self, position):

        self.is_not_used()
        # This method is created to get list of pixels of the 3x3 kernel.
        # Subsequently we will use that kernel
        # in order to get the depth values and average them

        # Reset the list of pixel
        list_of_pixels = []

        # Take the position of start pixel for 3x3 kernel
        start_x = position[0] - 1
        start_y = position[1] - 1

        # Create a loop to go over the pixels, and append them to the array
        for x in range(3):
            x_pos = start_x + x
            for y in range(3):
                y_pos = start_y + y
                # creating pixel position
                pixel = [x_pos, y_pos]

                # Append pixel to the list of pixels
                list_of_pixels.append(pixel)

        # Return list of pixels
        return list_of_pixels

    def read_depth(self, data, pixels):

        self.is_not_used()
        # This method is created to get the values of the kernel pixels
        # from point-cloud and average them
        # Create an empty list
        depth_list = []

        for pixel in pixels:
            # Taking the [x, y, z] values of the specific pixel
            data_out = pc2.read_points(data, field_names=None, skip_nans=False,
                                       uvs=[pixel])
            int_data = next(data_out)

            # Taking only z value, in which we are interested
            z_value = int_data[2]

            # Append z_value to the list
            if not math.isnan(z_value):
                depth_list.append(z_value)

        # Calculating list average value
        if not depth_list:
            avr = 0
        else:
            avr = sum(depth_list)/len(depth_list)

        # Every 50 iteration log the avr value
        if self.counter == 50:
            rospy.loginfo('the average depth value is {}'.format(avr))

        # Return avr
        return avr

    def depth_calculation(self, depth):

        # I will show the action every 50 iteration
        # This method takes the depth image and the center position of the ball
        # and returns the depth of the ball
        if depth >= 0.8:
            action = 'go'
        else:
            action = 'stay'

        if self.counter == 50:
            rospy.loginfo('the action is {}'.format(action))

        return action

    def command_to_the_robot(self, attitude, action):

        # This method will make a decision about the robot motion
        # The robot moves straight and turns left
        if attitude == 'left' and action == 'go':
            self.controller.left_go()

        # The robot only turns left
        elif attitude == 'left' and action == 'stay':
            self.controller.left_stay()

        # The robot moves straight and turns right
        elif attitude == 'right' and action == 'go':
            self.controller.right_go()

        # The robot only turns right
        elif attitude == 'right' and action == 'stay':
            self.controller.right_stay()

        # The robot moves straight
        elif attitude == 'center' and action == 'go':
            self.controller.center_go()

        # The robot stays on the same place
        elif attitude == 'center' and action == 'stay':
            self.controller.center_stay()

    def callback(self, rgb_data, depth_data):

        # This method getting the data from both topics and analyze them
        try:
            # take the image and 'bgr8' format
            image = self.bridge.imgmsg_to_cv2(rgb_data, "bgr8")
        except CvBridgeError as error:
            # If there is an error, log this error
            rospy.loginfo('the error: {}'.format(error))

        # Making preprocess of the frame
        # Applying GaussianBlur convolution kernel to reduce
        # the number the noise in the frame
        blurred = cv2.GaussianBlur(image, (11, 11), 0)
        # Changing BRG to HSV
        hsv = cv2.cvtColor(blurred, cv2.COLOR_BGR2HSV)
        # Applying the mask by using inRange function
        mask = cv2.inRange(hsv, self.lower_bound, self.upper_bound)
        # Now we firstly erode and the do opposite (dilate)
        # to remove small detections from the frame
        mask = cv2.erode(mask, None, iterations=2)
        mask = cv2.dilate(mask, None, iterations=2)

        # Looking for the contours, by using cv2.findContours function
        _, contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL,
                                          cv2.CHAIN_APPROX_SIMPLE)

        if contours:
            self.position = []
            # Going ahead and looking for the max contour in the contours
            c = max(contours, key=cv2.contourArea)
            # By using minEcnlosingCircle taking the coordinates of
            # the approximate center of the area
            ((x, y), radius) = cv2.minEnclosingCircle(c)
            # Append the data to new array, we will use this array subsequently
            # in the def position calculation
            self.position.extend([int(x), int(y), radius])

            # print(depth_image[self.position[1], self.position[0]])

            # If the object is big enough, it means the the operator
            # is close to the robot, go ahead
            if radius > 10:
                # draw the circle and centroid on the frame,
                # then update the list of tracked points
                cv2.circle(image, (int(x), int(y)), int(radius),
                           (0, 255, 255), 2)
                cv2.circle(image, (int(x), int(y)), 5, (0, 255, 255), -1)

                # Calculate the position of the ball
                attitude = self.position_calculator(self.position)

                # Create the list of pixels
                self.list_of_pixels = self.array_of_pixels(self.position)

                # Calculate the depth value of the ball
                depth = self.read_depth(depth_data, self.list_of_pixels)

                # Next method stands for command sending to the robot
                self.command_to_the_robot(attitude, depth)

                # reset the self.counter
                if self.counter == 51:
                    self.counter = 0

        else:
            # Stop the robot if there are no contours have been found
            self.controller.center_stay()

            # Print the msg to the operator
            if self.counter % 25 == 0:
                rospy.loginfo("No contours have been detected")

            # Reset the counter
            if self.counter == 51:
                self.counter = 0

        # show the image
        self.counter += 1
        cv2.imshow('image', image)
        cv2.waitKey(1)


def arg_parser():

    # Taking an argument from command line, creating an arg parser
    parser = argparse.ArgumentParser(description=
                                     'To choose the way of defining HSV')

    # analyse an argument
    parser.add_argument("-o", "--option", type=str, required=True, metavar='',
                        help='1 - by typing the values, 2 - use default values',)

    # store the argument in arg
    arg = parser.parse_args()

    # return arg
    return arg


def main():
    # Initiate the node named 'camera_parser'
    rospy.init_node('camera_parser')

    # Creating some local variables
    condition = False
    hsv_values = []

    # Choose the option for HSV settings
    arg = arg_parser()
    option = arg.option

    # Here we launched the program in two different ways,
    # by choosing the default HSV values
    # or typing them manually
    if option == '1':
        for i in ['MIN', 'MAX']:
            for j in 'HSV':
                value = input('%s_%s value:' % (i, j))
                hsv_values.append(int(value))
        # Condition True launched the BallTracking program, log HSV values.
        # The same operations in the line 198-199
        condition = True
        rospy.loginfo('The HSV values are next: {}'.format(hsv_values))

    elif option == '2':
        hsv_values = [29, 75, 6, 65, 255, 255]
        condition = True
        rospy.loginfo('The HSV values are next: {}'.format(hsv_values))

    if condition:
        # Running ball tracking application
        main_process = BallTracker(hsv_values)

        try:
            rospy.spin()
        except KeyboardInterrupt:
            rospy.logtinfo('The error has been occurred, '
                           'shutting down the program')

        cv2.destroyAllWindows()


# the program starts here, if the module have been launched
# by itself then the part of the code will run
if __name__ == '__main__':
    main()
```