# DORA - Dexterous Robot Assistant

Autonomous door opening with a mobile manipulator

Felix Bramsved
Lukas Gunterberg-Klase
Vincent Hendriksen
Ellen Schuchert

Bachelor's Thesis

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden
May 10, 2024

# Abstract

This thesis has focused on building a ROS based system for autonomous door detection and opening using a mobile manipulator. The mobile manipulator in question, DORA (Dexterous Robot Assistant), is based on the MiR200 mobile platform along with an UR10 robotic arm. The robot is equipped with an Intel realsense D435i depth camera. DORA had a navigation system and motion planning system for the arm already implemented and these systems were used in the process of this project.

Object detection was achieved through a YOLO algorithm on the image stream from the depth camera, and pose estimation was done using the depth values obtained. The program was tested and found to be consistent and accurate when the robot was located in front of the door, but becomes unreliable in edge cases when the door is located at the edge of the field of vision or on an angle.

The door opening program creates a trajectory for opening a door with unknown kinematics and an uncertain hinge location by utilizing readings from a force/torque sensor that is located between the arm and the end-effector, and the position of said end-effector. By employing the MoveIt framework, DORA was able to follow that trajectory and successfully open the door. Testing has confirmed that the force/torque sensor enables the program to operate successfully even in the presence of a 16% error in the estimation of the placement of the hinge position.

In conclusion it has been determined that although further development in safety systems and increased versatility is required for real world use, a solid foundation for further work has still been created in this thesis and the capabilities of DORA to interact with the world has been greatly expanded.

# Contents

**1 Introduction** **5**
1.1 Aim . . . 6
1.2 Objectives . . . 6
1.3 System Outline . . . 6
1.3.1 Door and door handle detection . . . 6
1.3.2 Door opening . . . 7
1.4 Deliminations . . . 7

**2 Background** **8**
2.1 Related Work . . . 9
2.2 Theory . . . 9
2.2.1 ROS . . . 9
2.2.2 Simulation . . . 10
2.2.3 Computer vision . . . 12
2.3 Ethical aspects . . . 13

**3 Methodology** **14**
3.1 Testing environments . . . 14
3.2 Object detection . . . 14
3.2.1 Image handling . . . 15
3.2.2 Object detection using YOLO . . . 16
3.3 Pose estimation . . . 17
3.3.1 Image coordinates of door . . . 17
3.3.2 Conversion to Cartesian coordinates . . . 17
3.3.3 Door pose creation . . . 18
3.3.4 Image coordinate of handle . . . 19
3.3.5 Handle pose creation . . . 21
3.3.6 Hinge pose creation . . . 22
3.4 Grasp movement . . . 22
3.4.1 Algorithm implemented . . . 22

**4 Results** **25**
4.1 Object detection . . . 25
4.1.1 Test: Door pose estimation . . . 25
4.1.2 Results: Door pose estimation . . . 27
4.1.3 Analysis: Door pose estimation . . . 28
4.2 Door opening . . . 29
4.2.1 Test: Hinge position error . . . 31
4.2.2 Results: Hinge position error . . . 32
4.2.3 Analysis: Hinge position error . . . 33

**5 Conclusion & Discussion** **34**
5.1 Object Detection . . . 34
5.2 Door opening . . . 34
5.3 Discussion . . . 35
5.4 Future work . . . 36
5.4.1 Extensive real world testing . . . 36
5.4.2 ROS 2.0 . . . 36
5.4.3 Image Segmentation . . . 37
5.4.4 Right or left opening door . . . 37
5.4.5 Filter for FT-sensor readings . . . 37
5.4.6 Rotating the end effector . . . 38

**6 References** **39**

**7 Appendix** **41**
7.1 Disclaimer on usage of ChatGPT and other LLMs . . . 41
7.2 Images . . . 41
7.2.1 Simulation scenes from 3.1 . . . 41
7.3 Code . . . 43

# Terminology

**DORA** Dexterous Robot Assistant, the name of the robot in this project. It is a mobile manipulator developed at Chalmers university.

**ROS** Robot Operating System, a set of software libraries and tools that help you build robot applications.

**Force/torque sensor (FT-sensor)** An electronic device that is designed to monitor, detect, record and regulate linear and rotational forces exerted upon it.

**YOLO** You Only Look Once, an object detection algorithm that can be used in real time.

**End-effector** A tool or grasper attached at the end of a robotic arm to perform a task or manipulate objects.

**Dummies** Dummies are collidable, measurable and detectable objects within the used simulation software CoppeliaSim.

**Waypoint** A waypoint is a specific point in space along a specified route.

# 1 Introduction

Robots have revolutionized our industry and are quickly expanding from the factories into different use cases, such as assistive or logistical helpers in spaces traditionally used by humans. An example of this is the robotic vacuum. In these environments a high degree of autonomy is required since it is often an unstructured and ever-changing space, where human intervention is not always possible.

Mobile manipulators are an interesting area of research due to their versatility. They combine the robotic manipulator, a precise and versatile arm, with a mobile robot base that gives it the ability to move freely. Their applications are just as broad as they are impressive, ranging from agricultural work [1] to disaster-response [2] (Figure 1). Even moon rovers could be considered mobile manipulators if one stretches the definition somewhat.

This thesis regards DORA (Dexterous Robot Assistant), a mobile manipulator developed at Chalmers University of Technology. DORA already has the ability to navigate and perform simple pick-and-place tasks. The work of this thesis aims to make DORA able to autonomously open doors, without prior knowledge of its environment or the kinematics of the door, in order to be a realistic application. This is an important function since many tasks in a home or work environment require the ability to open doors, drawers or cabinets.

Figure 1: The mobile manipulation robot Momaro used in [2] as a disaster-responder.

## 1.1 Aim

The aim of this thesis is to equip DORA with the ability to autonomously detect and open doors. DORA is deemed successful in its task if it can detect a door from a suitable distance, navigate to it, and open it at least 90 degrees.

## 1.2 Objectives

In order to reach the aim, the following tasks has been accomplished:

- **Object detection** is a reliable method of detecting objects in an image and has been used to detect doors and door handles in DORA's camera feed.
- **Grasping point generation** is needed to give the gripper on DORA a secure point to hold on to.
- Utilizing an **FT-sensor** (Force/Torque sensor) to enhance the arm's trajectory to effectively open the door.

## 1.3 System Outline

DORA is composed of many subsystems. The two developed for this thesis are the *door and door handle detection* and *door opening* systems.

### 1.3.1 Door and door handle detection

To extract useful data about its environment DORA uses an Intel realsense D435i depth camera, an RGB-D camera, mounted at the end of its arm, and generates a point cloud of everything the camera sees. Bounding boxes around doors and door handles are generated in the RGB image with the object detection algorithm YOLO 3 using the weights from [3]. The bounding boxes are used to find relevant points in the point cloud. The current implementation assumes that the corners of the bounding box corresponds to the doors corners, and that the grasp position of the handle is the point closest to the camera within the door handle's bounding box.

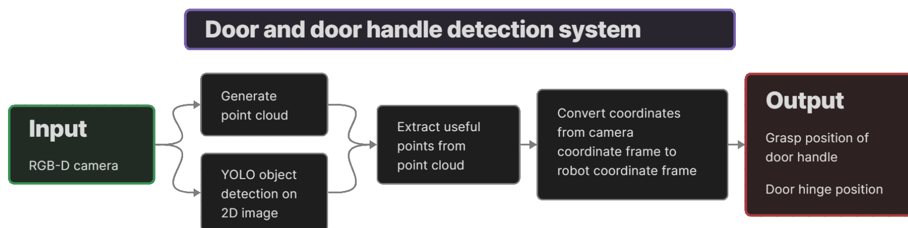



Figure 2: System outline of DORA's door and door handle detection

### 1.3.2 Door opening

To be able to open a door, DORA uses an FT-sensor mounted between the UR10 arm and the end-effector (an RG2 gripper in the simulation environment) that provides a data stream of force readings. The end-effector moves by providing waypoints to the MoveIt Motion Planning Python API. The force reading is used in the door opening algorithm when computing the next waypoint in the trajectory. When the door is in a position that is defined as open, an output is delivered that signals that the task is complete, and the program stops.

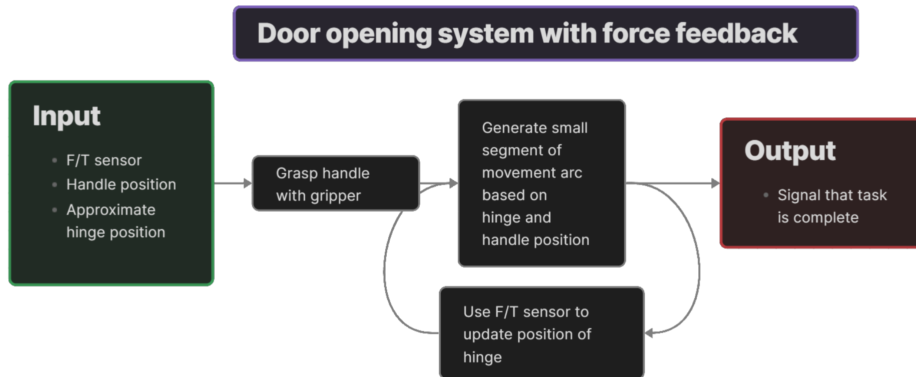



Figure 3: System outline of DORA's door opening function

## 1.4 Deliminations

The subject of door detection and opening and mobile manipulators covers a broad range of subjects and environments and there is only a limited time available for the work and a limited budget of 5000 SEK. It is thus necessary to place limitations on the work. The largest limitation placed is that the thesis is focused primarily on working within the simulation. This is both due to reported problems with the real robot and the time constraints placed upon the work.

The door that is to be detected cannot be partially obstructed by obstacles, or have properties that limit detection ability. These properties include, abnormal shape, transparent material or abnormal opening mechanism (sliding doors). Another limitation is to focus exclusively on doors with lever-type door handles, meaning that the program is not going to be able to detect or open other types of door handles, such as doorknobs.

Another limitation is that the program is only able to open doors outwards and not inwards. This is because the inwards door opening movement is more complex and requires the robot to move during the motion, as opposed to the outwards motion that can be done without moving the robot.

# 2 Background

This project builds upon previous bachelor projects using DORA, and will further develop and expand its capabilities. DORA is an ongoing project and its development will be continued.

DORA consists of the following components as seen in Figure 4:

- The mobile platform, MiR200 which use two laser scanners, two 3D cameras and four ultrasonic sensors.
- The robotic arm, UR10 with six degrees of freedom.
- A two-finger OnRobot gripper mounted at the end of the arm.
- An Intel realsense D435i depth camera mounted near the gripper.

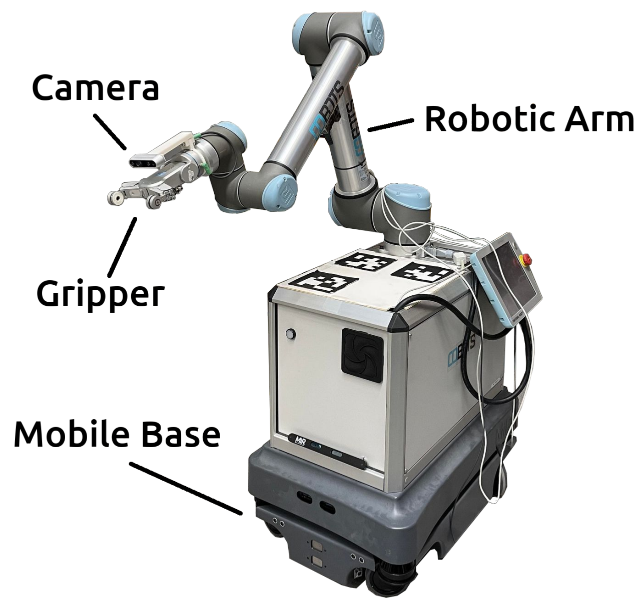



Figure 4: The main components of the mobile manipulator DORA.

## 2.1 Related Work

Opening a door with an autonomous mobile manipulator is an established and well researched topic. A paper [4] from 1995 aimed to realize a behavior of opening a door by finding the doorknob and pushing the door open. Similar to this work they divide the behavior routine into several subtasks.

Successfully opening a door mainly includes two subtasks. The first is to detect the position of the door and the door handle. A method of door handle detection proposed in [5] is using a CNN (Convolutional Neural Network) to generate ROI (Regions Of Interest) in which more computationally intensive algorithms are used to accurately find the position of handles. In [6] a CNN called YOLO (You Only Look Once) is trained to detect doors and door handles in the same way. The authors also provide the weights of the trained network [3] which are used in DORA's object detection system 4.1.

The second subtask is generating and executing the correct trajectory for the arm to follow in order to open the door. In [7] a FT-sensor is implemented with a method that is proposed to open doors without prior knowledge about their kinematics, with the help of velocity-controlled manipulators with force sensors at the end-effector. This is the method that suits the needs of this project best and it is this paper that forms the basis of the door opening algorithm described in this thesis. A study [8] regarding robot interaction with mechanisms with one degree of freedom such as doors, with no prior knowledge of the kinematics, gives proof that with the help of a velocity controller, a FT-sensor, and estimates consisting of motion direction, distance, and the orientation of the rotational axis converge to the true values. Another paper[9] proposes a method for controlling robotic interactions with simple mechanisms, such as doors or cranks, without requiring precise kinematic models. Traditional control methods often struggle with internal forces and modeling errors. Instead, this approach focuses on following the path of least resistance by learning the mechanism's shape while in motion, thus minimizing internal forces.

## 2.2 Theory

### 2.2.1 ROS

ROS stands for Robotic Operating System and is an open-source framework for designing and running robotic systems. ROS has three levels of concepts: the Filesystem level, the Computation Graph level, and the Community level.

The filesystem level covers resources such as packages, where the software in ROS is organized. It contains software such as nodes, libraries and configurations files.

The computation graph level is the peer-to-peer network that processes data through concepts such as nodes, topics and messages. Nodes send and receive

messages from topics and process the data contained within the messages. Several nodes can subscribe and publish to the same topic allowing for easy communication between different nodes. Within this project for example one node controls the object detection, another node will handle the image segmentation and yet another node controls the end-effector. This structure makes it easier to handle communication between different parts of the project and is well suited for a robot like DORA which has many sensors that have a continuous stream of data and information.

ROS is an open-source framework and the community level concepts are resources within ROS that makes it possible for different communities to contribute with both software and knowledge. ROS has different distributions similar to Linux that makes it easier to install a collection of software. In this project the ROS Noetic Ninjemys distribution is used. Different institutions develop and share their own repositories containing robot software components that others can make use of.

**TF2** is a library in ROS used to keep track of the physical relationships between objects in the ROS environment. This is done through transforms that include data on the position and rotation of an object relative it's parent frame. These transforms linked together make up a /tf tree which can be used to quickly look up the relationship of two objects, provided that they're linked through the /tf hierarchy.

**MoveIt** is a software framework within ROS used for motion planning and grasp generation in robots with multiple degrees of freedom.

### 2.2.2 Simulation

During development of a complex mechatronical system such as DORA it is crucial to have an accurate simulation environment, to enable easier testing without the risk of damaging the real robot and in order to avoid the time costly process of reseting the robot and uploading changes.

**CoppeliaSim** is the simulation program primarily used in this project. It features extensive tools for sensor simulation and Forward/Inverse kinematics calculations among other things.

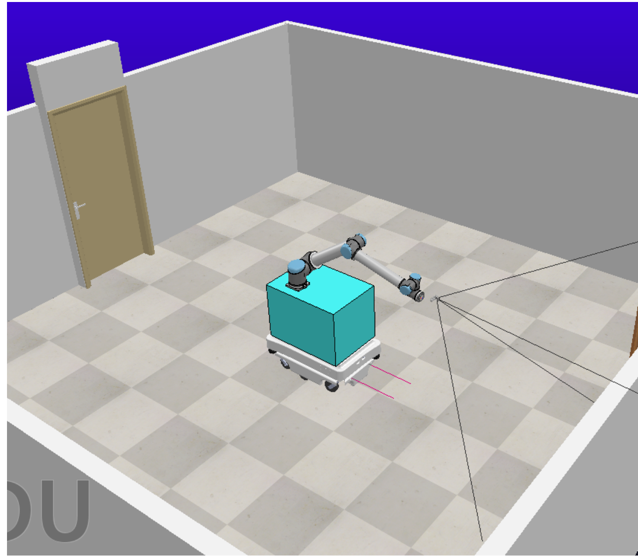


Figure 5: DORA within the CoppeliaSim environment

**RViz** is an open-source 3D ROS visualizer. It is used to visualize the robot model, as well as any sensor data the robot is using. It can also be used to send messages to the ROS system, such as navigation goals or motionplanning poses.

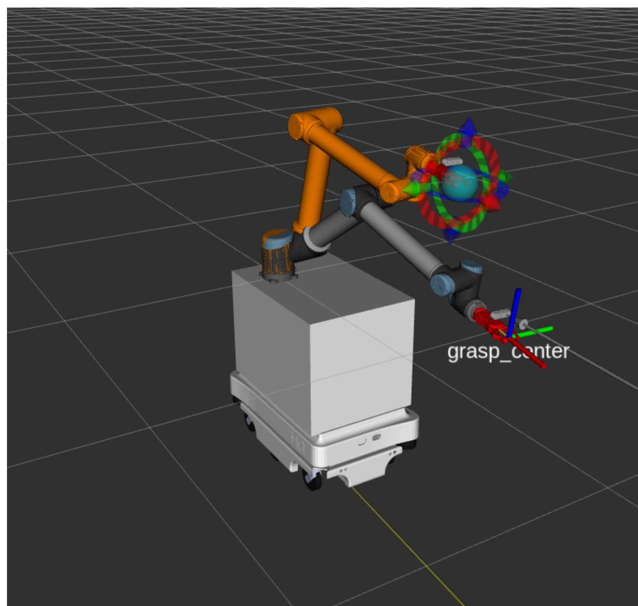


Figure 6: Motionplanning interface in RViz

### 2.2.3 Computer vision

Computer vision is a field of artificial intelligence which focuses on enabling a computer to analyze, interpret and understand visual information in the form of images. Computer vision plays a large role within robotics, as it allows the robot to autonomously perceive and analyze visual information from it's environment, and thus be able to detect objects of interest and potential obstacles.

**OpenCV** stands for Open Source Computer Vision Library and is an open-source library for use within computer vision and machine learning. OpenCV provides a large amount of functions for use in image analysis, object detection and image segmentation. It is a very powerful toolset and can be used across multiple platforms and languages.

**YOLO** stands for You Only Look Once and is a real-time object detection algorithm. It is notable for its speed and efficiency in classifying objects, as well as it's ease of use. There are many pre-trained datasets for different objects that can be downloaded which removes the need to train new datasets.

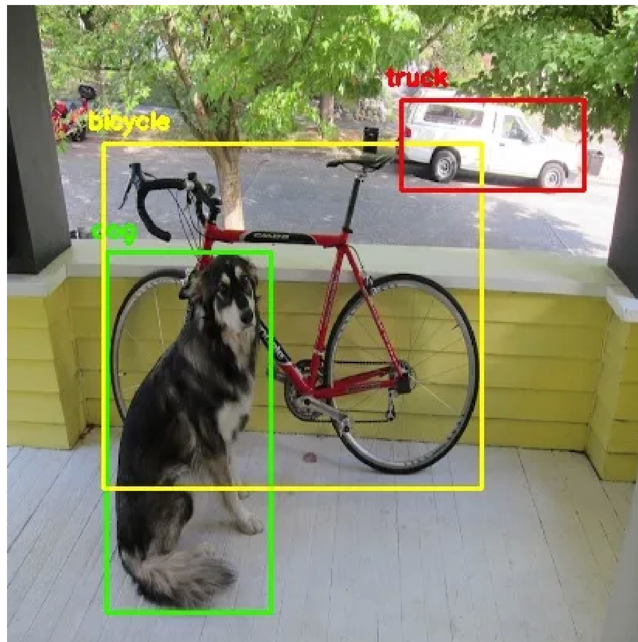



Figure 7: Example of object detection using YOLO [10]

## 2.3 Ethical aspects

When developing a robot with capacity for mobile manipulation, considering the potential use cases and ethical aspects of those cases needs to be done. Mobile manipulators have traditionally been used primarily within healthcare and assisted living [11], where they're primarily used to assist the user in different tasks such as grasping objects that would otherwise be out of reach for the user. With this usage there is the considerable problem of safety. If the robot is to be used in an environment such as a hospital or around already very vulnerable people then the safety standards needs to be very high. Although fully implementing this functionality is outside the scope of the project it is nevertheless important that safety features are kept in mind.

The other common use case is as an assistant to humans in a workshop or factory-like environment [12], where they perform menial tasks such as assisting in fetching and handing tools to the user. This has the potential to replace existing jobs, but as fetching tools and other menial tasks is rarely the sole task of anyone working in these environments, and this thesis is unlikely to make significant improvements over existing technologies, it likely won't have much of an impact on the livelihoods of the people working in these environments. The integration of mobile manipulators in these work-spaces can also lead to more jobs [13] as the introduction of robots will introduce the need for highly skilled jobs in their maintenance and installation.

# 3 Methodology

## 3.1 Testing environments

Several simulation environments were created and used in the process of developing and testing new features. Images of the environments described here can be found in the appendix at 7.2.1.

The primary testing environment utilized in this study was the **workshop** (Figure 25), a moderately sized, square room equipped with a table and a single door. Its primary function revolved around serving as a versatile testing space.

In order to evaluate door detection, a modified version of the workshop, named **door detect** (Figure 26), was created. This variant retains the workshop's layout but removing the table and introducing an additional door on the opposite wall. This configuration was used to assess the algorithm's proficiency in detecting doors of varying appearances.

For the development of the door opening algorithm, a dedicated **door opening** environment (Figure 27) was created. Unlike the 'door detect' setup, this environment is a compact variation of the workshop, featuring a small box with a vertical lid placed atop the table. This choice was made due to the superior reliability of the small box's physics compared to the simulated doors available in CoppeliaSim.

## 3.2 Object detection

The object detection program has two primary tasks: identifying the position of doors, and locating the position of their handle and hinge once a door has been detected. This is accomplished by combining an object detection algorithm, specifically YOLO with a model trained for detection of doors, with the navigation stack for navigating the robot closer to the door and the ROS tf2 library in order to upload the transforms extracted to the /tf tree for use in the manipulation program.

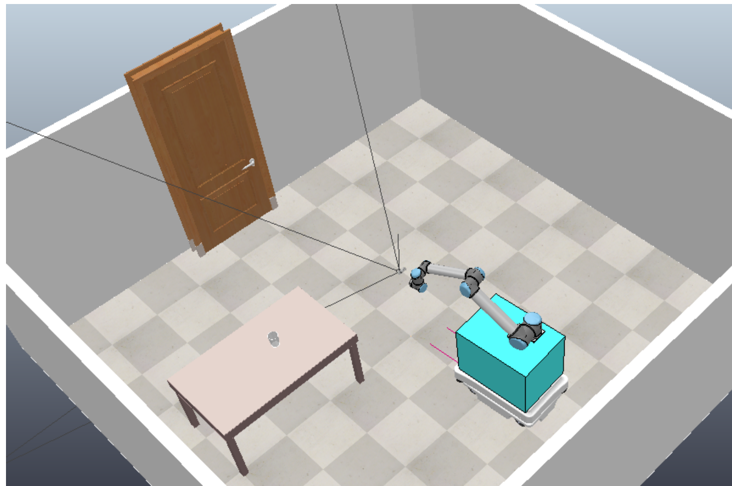


Figure 8: Example of the initial state of the object detection program

The following outlines the program flow:

1. Utilize object detection to analyze the image flow.
2. Upon identifying a door:
   (a) Calculate the door's position relative to the robot.
   (b) Navigate the robot to a distance within 2 meters in front of the door.
3. Analyze the image stream to detect the handle's location.
4. Compute the coordinates of the handle and derive the hinge position from the door coordinates.
5. Upload both points to /tf for use in the manipulation program.

#### 3.2.1 Image handling

When the object detection is running the color camera and the depth camera each publish their corresponding image stream, along with their camera information, to ROS topics, as illustrated in Figure 9.

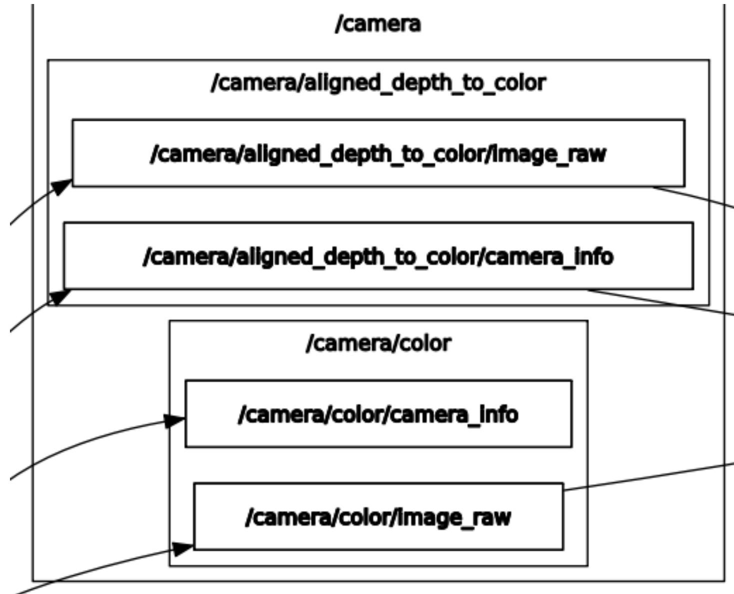



Figure 9: RQT graph of the camera ROS topics

The object detection program subscribes to these topics and receives the raw images and associated camera info. The color image gets encoded using OpenCV into a .jpg image for use in object detection, while the depth image gets encoded as a numpy array to be used to calculate the coordinates of detected objects.

### 3.2.2 Object detection using YOLO

The object detection programs works with a YOLO algorithm adapted from Arun Ponnusamy's article in Towards Data Science [14]. The image detection part of the algorithm remains mostly unaltered, though the handling of the detected object has been reworked. The dataset used is from Miguel Arduengo et al research paper Robust and adaptive door operation with a mobile robot [3]. Arduengo's dataset is trained on detecting doors, door handles, cabinet doors and refrigerator doors, and thus works very well for the purposes of this project.

The processed images are sent to the YOLO function where it uses the provided dataset to detect the location of the different object. Cabinet and refrigerator doors get ignored by the algorithm as it is not relevant to this project. Detected doors and door handles get handled in different ways depending on the state of the program.

## 3.3 Pose estimation

After an object has been detected using YOLO, the next step is to extract a pose in 3D space from the information provided by the object detection and the depth image. This is accomplished by finding the correct coordinates in the image and extracting the depth value of the point. From this depth value a coordinate in 3D space can then be extracted.

### 3.3.1 Image coordinates of door

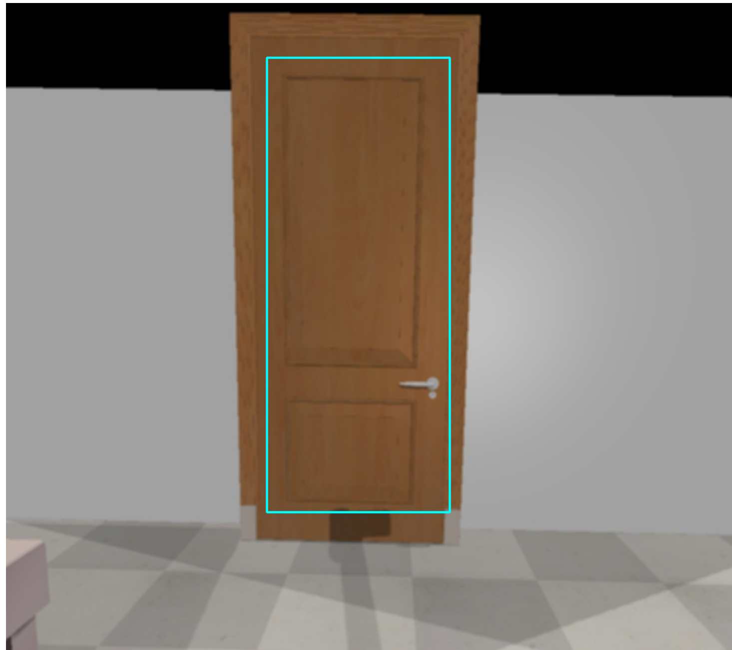


Figure 10: A detected door object with bounding box within the simulation

When YOLO detects a door object it returns the point in the top left corner of the object, along with width and height. From these values the coordinates of all four corners of the door can be calculated.

### 3.3.2 Conversion to Cartesian coordinates

After acquiring the coordinates of the object in the image space the next step is to convert them to Cartesian coordinates. This is done using the depth image, the focal length of the camera, and the principal point of the camera. The focal length is the distance from the camera lens to the camera sensor, and the principal point defines the center of the image plane and is used for correcting lens distortions.

In the following equations fx and fy represent the focal length in x and y directions and cx and cy represent the principal point in respective direction. Z represent the depth value of the coordinate (x,y) in the depth image.

$$X_{Cartesian} = \frac{(x - cx)Z}{fx} \tag{1}$$

$$Y_{Cartesian} = \frac{(y - cy)Z}{fy} \tag{2}$$

From the equations we get the coordinate (X,Y,Z) which represents the Cartesian coordinate of the point relative to the optical frame of the depth camera.

#### 3.3.3 Door pose creation

The coordinates of the door's corners are relative to the optical frame of the robot, which is not stationary and can be expected to move around slightly, even when the robot is still. Thus in order to get a more accurate pose multiple coordinates are collected over several cycles, outliers are sorted out and an average of the rest is taken.

When calculating the position of the door the center is acquired from the average value of the four coordinates of the corners. The orientation of the door is calculated by finding the normal of the plane created by the vectors derived from the corners of the detected door as demonstrated in Figure 11.

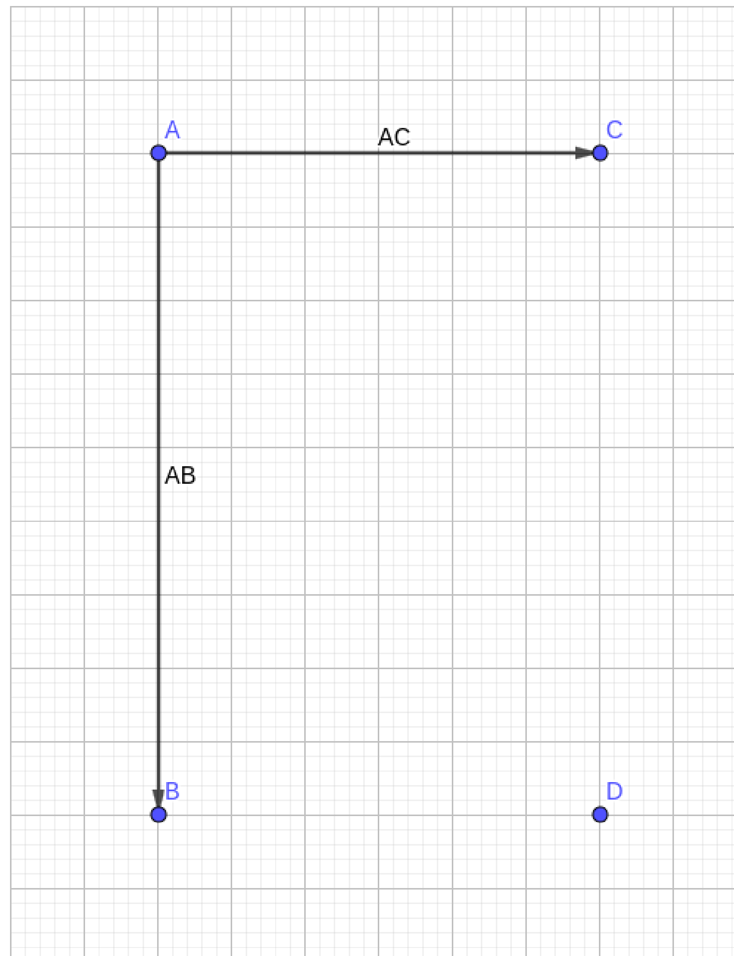



Figure 11: Illustration of the door plane

The edges of the door are also recorded and published to the /tf tree for use in estimating the position of the door hinge.

#### 3.3.4 Image coordinate of handle

When a pose has been successfully estimated for a door the program will move the robot closer to the door to allow for more accurate estimation of the handle's pose as demonstrated in Figure 13.

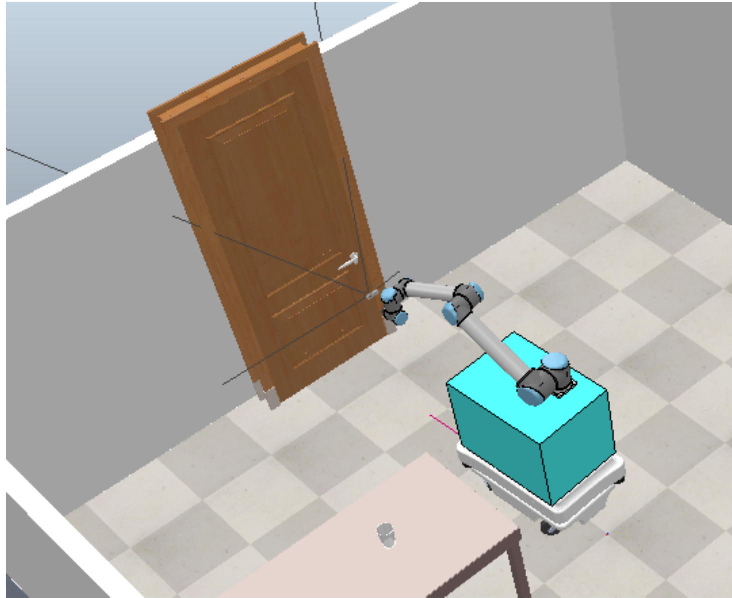


Figure 12: State of the program after DORA has moved closer for handle detection

When YOLO detects a handle object it returns the point in the top left corner of the detected object, along with width and height. From these values the depth image can be cropped to only contain the detected handle object.

Figure 13: The closest point on a detected handle object represented by a blue dot

It is assumed that the point closest to the camera will always lie on the handle itself, and thus the algorithm will find the point in the cropped depth image with the lowest depth value, and this point will be the point used to calculate the pose of the handle.

### 3.3.5 Handle pose creation

The handle's coordinate, acquired in the conversion to Cartesian coordinates is relative to the optical frame of the camera, which is located on the mobile robotic arm. This means that any pose created from this coordinate will move along with the robot. This is not sufficient for the grasp movement program, which requires a static pose, relative to the map frame.

Thus the pose is translated so it is relative to the map frame before it is published to the /tf tree. This transform is acquired with the lookup_transform function, which acquires the transform between two frames in the /tf tree, along with do_transform_pose, which applies a transform to a pose.

The orientation of the handle is set to the same as the door, as the handle is expected to always have the same orientation as the door. The handle transform is then published to /tf.

### 3.3.6 Hinge pose creation

The hinge position is acquired from the door edge positions published during the door pose creation. The locations of the edges of the door are compared to the position of the handle, and the one furthest away from it is assumed to be the edge that the hinge is located on. The hinge object is assigned the same X and Y coordinate as the door edge, but the same Z value and orientation as the handle. The hinge transform is then published to /tf.

## 3.4 Grasp movement

The following outlines the larger concept of how the manipulator moves and each part will be thoroughly discussed. In Figure 14 an overview of the door is illustrated along with a vector between the handle position and estimated hinge position. The gripper moves by providing waypoints to the MoveIt Motion Planning Python API. MoveIt then utilizes Cartesian path planning to calculate and execute the desired trajectory towards the waypoint needed for the robot. This task was completed with help from the offical Moveit tutorial[15]. An adapted algorithm, discussed in Section 3.4.1 computes the next waypoint using two estimated points in space, one for the door handle and one for the door hinge, where the initial estimates are given from the door detection program. However, two dummy waypoints were used instead while testing the algorithm, until the full solution including the pose estimation was implemented. The program then continues to calculate new waypoints until the door is estimated to be opened. The waypoints work as a trajectory for how the door needs to move to be opened.

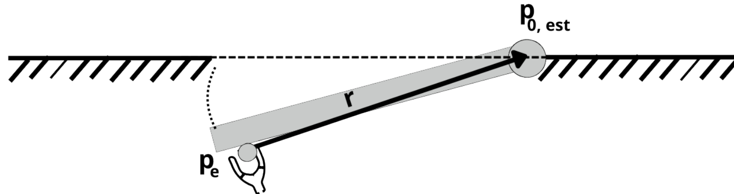



Figure 14: Illustration of the door

### 3.4.1 Algorithm implemented

The algorithm implemented is mainly based on the one developed by [7]. Since no velocity controller was used, the implementation in this work has been adapted to calculate waypoints instead. This was done by using a hypothetical velocity to calculate the distance to the next waypoint.

Assuming that the end-effector has already moved to and is gripping the handle, the algorithm to calculate the next waypoint is outlined as follows:

1. Fetch the end-effector position, and save it as the current door handle position, $\mathbf{p}_{\mathrm{e}}$.
2. Fetch the force sensor readings, $\mathbf{F}$, but set the force measurements in the z-axis to zero.
3. Calculate the hypothetical velocity, $\mathbf{v}_{\mathrm{ref}}$, by using an estimate of the door hinge position, $\mathbf{p}_{0,\mathrm{est}}$, and force sensor readings and save the next handle position coordinates. Also save the new velocity as the current velocity.
4. Rotate the handle around the z-axis with a fixed set of degrees to prevent the gripper from letting go of the handle.
5. Move the end-effector to the next waypoint.
6. Repeat steps one to five until the door angle is estimated to have turned 90 degrees.

As in [7], this paper's hypothetical velocity uses the force sensor reading to change the direction of the pull and adapt to eventual errors in the estimation of the hinge position, as well as update the estimated hinge location. This is done in multiple steps. In Table 1 an overview of the variables used in the calculations is presented.

Table 1: Variable descriptions

| **Variable** | **Description** |
|---|---|
| $\mathbf{p}_{0,\mathrm{est}}$ | Estimated hinge position |
| $\mathbf{p}_e$ | End-effector position |
| $\mathbf{r}$ | Vector representing the difference between the estimated hinge position ($\mathbf{p}_{0,\mathrm{est}}$) and the door handle position ($\mathbf{p}_e$) |
| $\hat{\mathbf{r}}$ | Unit vector in the direction of $\mathbf{r}$ |
| $F_{\mathrm{radial_est}}$ | Estimated radial force, obtained by projecting force ($\mathbf{F}$) onto $\hat{\mathbf{r}}$ |
| $\Delta F_{\mathrm{radial_est}}$ | Difference between the estimated radial force ($F_{\mathrm{radial_est}}$) and the desired radial force ($F_{\mathrm{radial_desired}}$) |
| $\int \Delta F_{\mathrm{radial_est}}$ | Integral of the change in estimated radial force over time |
| $F_{\mathrm{feedback}}$ | Feedback force based on $\Delta F_{\mathrm{radial_est}}$ and its integral |
| $\mathbf{v}_{\mathrm{ref}}$ | Hypothetical velocity |
| $\hat{\mathbf{r}}_{\perp\mathrm{est}}$ | Unit vector orthogonal in the door opening direction $\hat{\mathbf{r}}$ |
| $\mathrm{v}_{\mathrm{desired}}$ | Desired tangent velocity |
| $\alpha$ | Coefficient for how much impact the force feedback has on the $\mathbf{v}_{\mathrm{ref}}$ |
| $\beta$ | Coefficient for integral feedback |
| $\Delta t$ | Time step |
| $\gamma$ | Coefficient for estimating $\mathbf{p}_{0,\mathrm{est}}$ |
| $\mathbf{v}_{\mathrm{curr}}$ | Current velocity |

The first step in computing the hypothetical velocity in step 3, is determining the normalized vector extending from the end-effector to the estimated hinge (see Equations 3 and 4).

$$\mathbf{r} = \mathbf{p}_{0,\text{est}} - \mathbf{p}_{\mathbf{e}} \tag{3}$$

$$\hat{\mathbf{r}} = \frac{\mathbf{r}}{\|\mathbf{r}\|} \tag{4}$$

Then the force projection, $\text{F}_{\text{radial_est}}$ onto $\hat{\mathbf{r}}$ is compared to a desired constant value (Equation 5), and then used to calculate the difference between the estimated radial force and the desired radial force (Equation 6), which is then integrated over time along with the change in estimated radial force to obtain the force feedback (Equations 7 and 8.).

$$\text{F}_{\text{radial_est}} = \hat{\mathbf{r}} \cdot \mathbf{F} \tag{5}$$

$$\Delta F_{\text{radial est}} = F_{\text{radial_est}} - F_{\text{radial_desired}} \tag{6}$$

$$\int \Delta F_{\text{radial_est}} = \int \Delta F_{\text{radial_est}} + \Delta t \cdot \Delta F_{\text{radial_est}} \tag{7}$$

$$F_{feedback} = \Delta F_{\text{radial_est}} + \int \Delta F_{\text{radial_est}} \cdot \beta \tag{8}$$

Finally, the $\mathbf{v}_{\text{ref}}$ is calculated using the unit vector orthogonal in the door opening direction, a desired velocity (essentially determining the distance), and with a scaling factor($\alpha$) multiplied by the estimated radial vector ($\hat{\mathbf{r}}_{\text{est}}$) multiplied with the force feedback (Equation 9).

$$\mathbf{v}_{\text{ref}} = \hat{\mathbf{r}}_{\perp\text{est}} \cdot \text{v}_{\text{desired}} + \alpha \cdot \hat{\mathbf{r}}_{\text{est}} \cdot F_{feedback} \tag{9}$$

Since the hinge's exact location is unknown and only estimated, the position is continuously updated based on the force sensor as shown by Equation 10 and 11.

$$\dot{\mathbf{p}}_{0,\text{est}} = \frac{\gamma}{\|\mathbf{r}_{\text{est}}\|} \left( \text{k}_f \cdot \Delta F_{\text{radial_est}} + \text{k}_I \cdot \int \Delta F_{\text{radial_est}} \right) \cdot \mathbf{v}_{\text{curr}} \tag{10}$$

$$\mathbf{p}_{0,\text{est}} = \mathbf{p}_{0,\text{est}} + \dot{\mathbf{p}}_{0,\text{est}} \cdot \Delta t \tag{11}$$

# 4 Results

## 4.1 Object detection

During the development of the object detection software it was noticed that the door pose estimator wasn't completely accurate in all situations. When detecting from an angle it would sometimes get the door's position wrong, and thus be unable to get an accurate estimation of the handle.

### 4.1.1 Test: Door pose estimation

In order to test the accuracy of the door pose estimator the simulation environment was modified to publish the actual pose of the door object to /tf. This enables an direct comparison between the measured door pose and the actual location of the door.

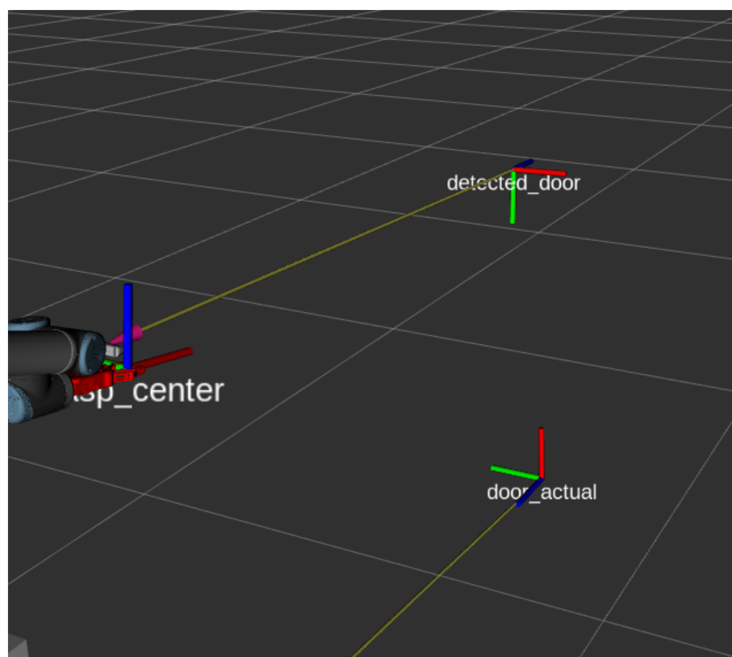



Figure 15: Pose comparison in RViz, illustrating the difference in pose height

As seen in Figure 15 however there is an height differential between the two poses, as the pose estimator places the pose in the center of the door, while the door publisher publishes the pose at the bottom of the door. But as the height of the pose matters little to the function of the program, and this height differential can be expected to be constant it can be safely ignored.

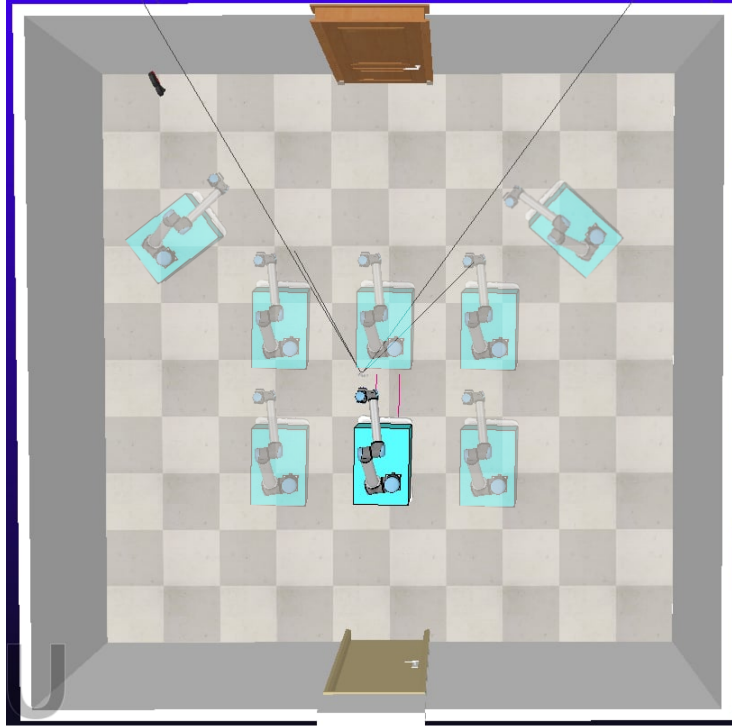


Figure 16: Illustration of how the test was performed

The test consisted of doing pose estimations from different positions and angles from the door, as illustrated in Figure 16 (note that the image is for illustrational purposes and does not accurately reflect the actual positions the test was performed with).

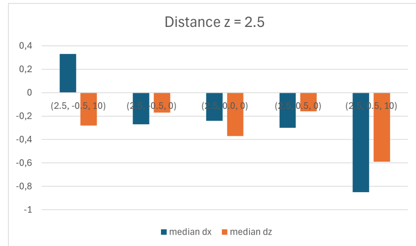
The lower bound for the distance to the door on the Z axis is limited at around 2 meters which is where the door becomes too close for YOLO to detect it accurately. The upper bound was set at 4 meters, as the depth camera has an ideal range of 3 meters and going further would thus risk the data being inaccurate outside of the simulation. The range of distance on the X axis is limited by where the door exits the camera frame for the different Z values.

For each position the test performed 100 estimations and recorded $\Delta X$, $\Delta Y$ and $\Delta Z$ for each one. This data would then be processed and the median deviation in all three coordinates would be collected, the percentage of outliers from the median and the mean deviation from the median would be collected. The median was chosen instead of the mean to avoid outliers contaminating the

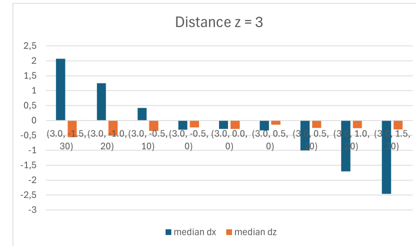
data.

### 4.1.2 Results: Door pose estimation

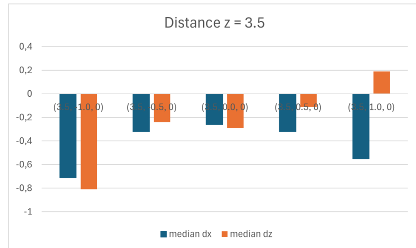
The results presented here illustrate the median delta X and median delta Z for different measuring positions. The layout of the graph corresponds with the measuring positions, with the left and right parts of the graph corresponding to the edge cases of the measurement and the center corresponding to the measurements from the center.



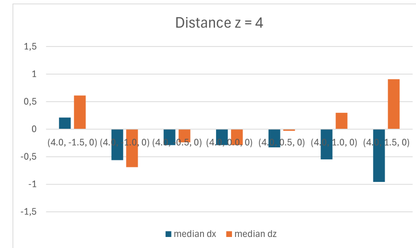

(a) Median dx and dz with distance to target 2.5 m




(b) Median dx and dz with distance to target 3 m




(c) Median dx and dz with distance to target 3.5 m




(d) Median dx and dz with distance to target 4 m

Figure 17: Analysis of robot positions relative to the door at different distances

The positions illustrated in these charts represent the base of the robot positioned at varying distances from the door in the Z direction. Each chart spans a range of distances in the X direction, with the far edge cases including different angles relative to the door.

The results consistently show a deviation in both the X and Z directions from the real position of the door. In the center cases, the deviation is approximately -0.3 meters in the X direction and approximately -0.2 meters in the Z direction. This deviation remains consistent across different distances from the door. However, in the edge cases, where the angle between the robot and the door increases, the deviation grows larger, indicating decreased accuracy in these scenarios.

#### 4.1.3 Analysis: Door pose estimation

As observed from the results, there is a consistent error in the X direction of around -0.3 meters and around -0.2 meters in the Z direction. This pattern holds as long as the door is centered in the image, and as long as there is no angle between the robot and the door. Though when these conditions are not met the median values diverge from the true position.

This pattern suggests there's a systematic error in the measurement of approximately -0.3 meters in the X direction and -0.2 meters in the Z direction. This systematic error can be countered by applying an offset to the estimated pose.

The larger deviation observed in cases where the object is located on the edge of the camera frame or the object is on an angle from the camera, can be explained primarily in two ways. There is the possibility that the camera distortion on the edges in the simulation is not accurate to the camera info given and that the conversion to Cartesian coordinates thus fails to give accurate values in these cases. In this case the problem can be solved by obtaining the correct camera values and testing in the real world.

The other primary explanation is that the YOLO algorithm gives a warped bounding box that not only includes the object of interest but also parts of the background as demonstrated by Figure 18. In cases where this is the primary problem this could potentially be solved by using a more sophisticated method for detecting the doors.

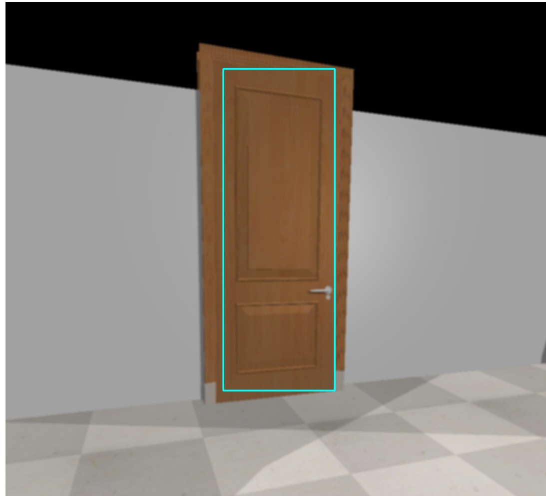


Figure 18: A door detected from an angle, illustrating the potential warping of the bounding box

As can be seen from the results the pose estimation is only accurate when DORA

is in front of the door being detected. A potential stopgap solution for this that would make the program more robust is that once a door has been detected in an non-ideal position, DORA will move to a more ideal position and the redo the detection algorithm. This can be achieved by analyzing the detected doors position in the camera frame and checking if it is close to the edges where the distortion becomes a problem. As for the angles this can be achieved by checking the direction of the calculated quaternion.

## 4.2 Door opening

After integrating an FT-sensor between the arm and end-effector on DORA within the CoppeliaSim simulation environment, as well as implementing the door opening algorithm, DORA successfully managed to open a door in the simulation. However this was managed with exact estimations of the positions of the door handle and hinges in a controlled simulation environment as seen in Figure 19. In Figure 19 there is a box with a lid that is flipped on its side to resemble a door that was used to test our door opening algorithm. DORA can consistently open the door as demonstrated in Figure 20.

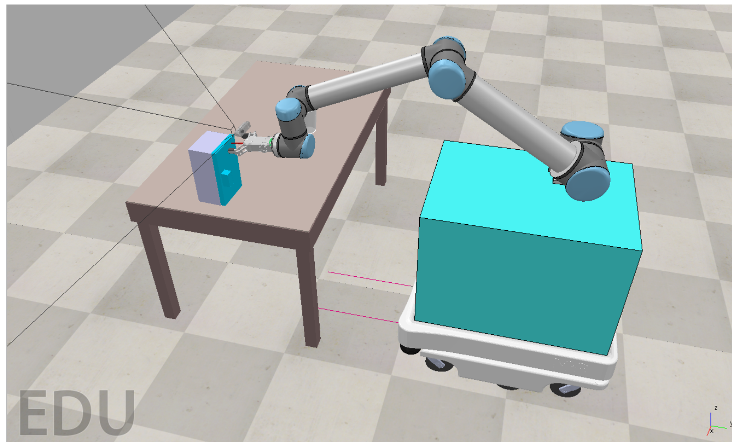
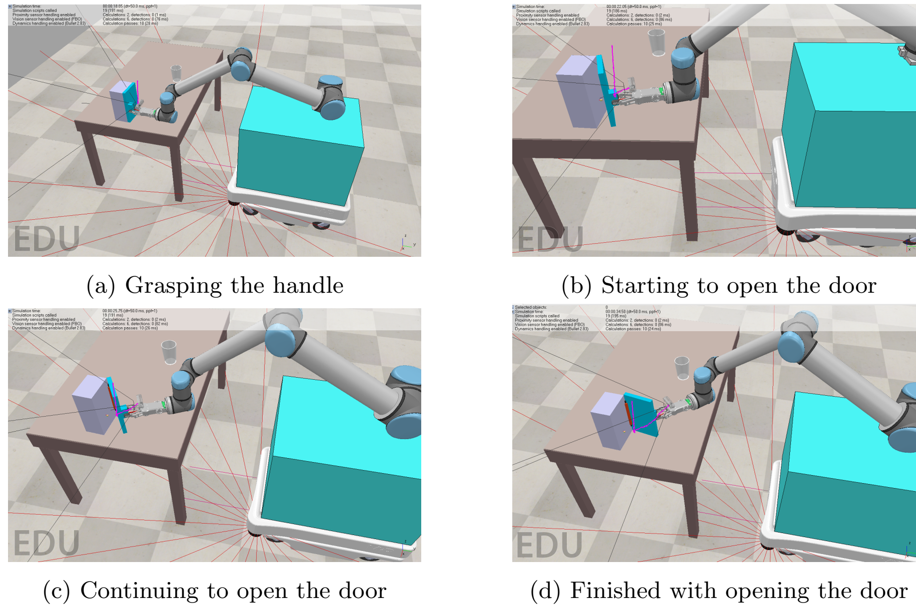


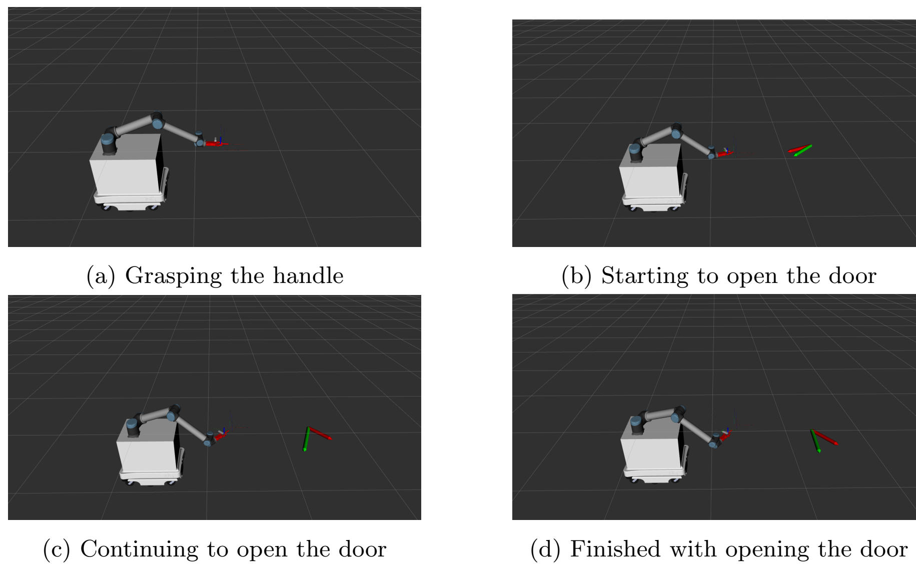

Figure 19: Starting point for door opening sequence

In Figure 21 there can be sees a complete view of the door opening sequence this time in RViz with force vectors shown to visualize how the force feedback effects the trajectory when it is included when calculating the hypothetical velocity (green: with force feedback, red: without force feedback). The effect of the force feedback is discussed more in depth in Section 4.2.3 but it resulted in a smoother trajectory and less force exerted on DORA's arm and the door when opening it.



(a) Grasping the handle
(b) Starting to open the door
(c) Continuing to open the door
(d) Finished with opening the door

Figure 20: Door opening sequence in CoppeliaSim



(a) Grasping the handle
(b) Starting to open the door
(c) Continuing to open the door
(d) Finished with opening the door

Figure 21: Door opening sequence in RViz with force vectors (green: with force feedback, red: without force feedback)

### 4.2.1 Test: Hinge position error

With the current method to generate a trajectory to open a door, the biggest uncertainty is the initial estimated hinge position. The algorithm's robustness was tested by providing a small error when giving the initial estimated hinge position. This test was implemented and observed by moving the dummy in CoppeliaSim which gives the initial position of the hinge and then running the simulation, seeing if DORA could still successfully open the door.

The tuning parameters in the algorithm were set to the values seen in Table 2 during the tests.

Table 2: Tuning parameters in door opening algorithm

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $\alpha$ | = | 0.05 | $k_f$ | = | 0.1 | $\Delta t$ | = | 0.1 |
| $\beta$ | = | 0.1 | $k_I$ | = | 0.05 | $\Delta\theta$ | = | $\frac{\pi}{20}$ |
| $\gamma$ | = | 0.5 | $\mathrm{v}_{\mathrm{desired}}$ | = | 0.5 | | | |
| | | | $\mathrm{F}_{\mathrm{radial_desired}}$ | = | 2 | | | |

The box is 0.15 meters wide and the position of the dummy was moved in intervals of 0.006 meters up until what would correspond to an 20% error in the estimation of the initial hinge position. The position of the dummy was only moved in along the x-axis as illustrated in Figure 22. The dummy was moved to the left (denoted as negative in Table 3) as well as to the right (denoted as positive in Table 3) to simulate that the estimated distance between the door handle and hinge could both be smaller and bigger than what it is in reality. The bounds of the testing are based on the results that was observed in the tests regarding door pose estimation in Section 4.1.2, this is to be able to analyse if the solutions will work together even with error in the estimations. This test was made to be able to conclude the impact of using force feedback versus not using force feedback when calculating the hypothetical velocity, which is controlling the estimated direction of the end-effector. However the force feedback is still being used to calculate the position of the door hinge.

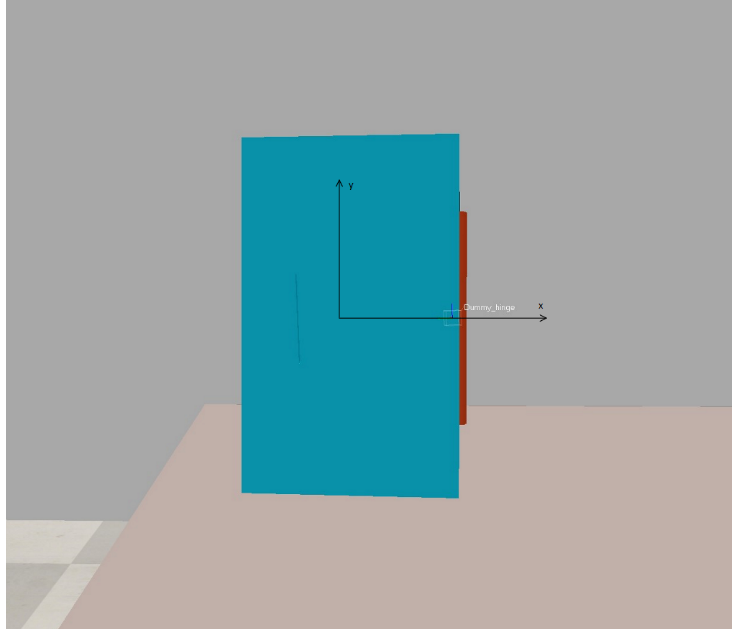



Figure 22: Illustration of hinge position error test

#### 4.2.2 Results: Hinge position error

The results presented in Table 3 display the offset of the initial door hinge position and if the task of opening the door was still successful or not.

The results shows that with the force feedback DORA can still successfully open the door more times than without the force feedback when an offset occurs with the initial position of the door hinge. However these tests were not 100% consistent which is discussed in Section 4.2.3, however it is still possible to conclude that the success rate of opening the door is better when the force feedback is included to calculate the hypothetical velocity in the algorithm.

Table 3: Results: Hinge position error

| | | | |
|---|---|---|---|
| + 0.006 | Success | - 0.006 | Success |
| + 0.012 | Success | - 0.012 | Success |
| + 0.018 | Success | - 0.018 | Fail |
| + 0.024 | Success | - 0.024 | Success |
| + 0.030 | Fail | - 0.030 | Fail |

Table 4: With force feedback

| | | | |
|---|---|---|---|
| + 0.006 | Fail | - 0.006 | Fail |
| + 0.012 | Success | - 0.012 | Fail |
| + 0.018 | Fail | - 0.018 | Fail |
| + 0.024 | Success | - 0.024 | Success |
| + 0.030 | Fail | - 0.030 | Success |

Table 5: Without force feedback

### 4.2.3 Analysis: Hinge position error

As observed from the results, when the initial position of the hinge on the door had a varying offset, DORA successfully opened the door seven out of ten times when force feedback was included and four times out of ten when it was not. As mentioned in the results, the consistency of the tests was not 100% and this was counteracted by redoing the tests that were inconsistent several times and using the result that was most frequent. An improvement that would make the door opening algorithm more robust would be to implement a filter for the FT-sensor reading, which is discussed in Section 5.4.5, since a lot of noise is received from the readings.

During the tests it was also noticed in some incidents when the force feedback was not included that the arc of the trajectory of the end-effector would be quite tight, which means that the end-effector is trying to be too close to the hinge while simultaneously holding the handle, in some of the successful attempts of opening the door. This works in simulation but in reality would cause a lot of force to be exerted on both DORA's arm as well as the door, which could potentially result in opening the door in an manor which would not classify as an successful operation, or completely fail to open the door.

# 5 Conclusion & Discussion

The aim for this thesis was to implement autonomous door detection and opening with DORA. This objective has been successfully accomplished, as DORA is now capable of detecting a door, navigating towards it, and opening it without human intervention. Although the program is not entirely fault-tolerant and requires human oversight to operate safely, the main functionality has nevertheless been implemented.

## 5.1 Object Detection

The accuracy of the object detection and pose estimation is very variable and far from robust in many cases, although the algorithm is accurate and consistent when the robot is within an ideal range of distances from the door being detected. Whether the robot is located within this range can easily be measured and thus the accuracy of the measured pose can be verified within the algorithm, allowing for the DORA to move into the desired area and redo the measurement in cases where the data is inaccurate.

Due to time constraints this re-measurement function has not been implemented in this project, however due to the ease of checking the validity of the measured pose and the fact that the functionality for moving the robot already exists within the algorithm implementing the function will be trivial, although it would have to be tested and calibrated, hence why it is deferred for future work.

The risks when the measurement is not accurate is negligible. The navigation system will prevent DORA from colliding with a wall or other obstacle, and the biggest problem is that the program will not be able to detect the handle and thus fail its task. When DORA is located within the safe area for measurements the pose estimation is very consistent and accurate, allowing for successful operation of the rest of the systems.

Overall DORA is mostly able to accurately detect doors and their handles, although improvements to the pose estimator is needed in order to make it truly robust. For the purpose of this thesis its current performance is deemed acceptable.

## 5.2 Door opening

The door opening sequence works, but it could be more consistent when an error in the hinge position exists. The success rate of opening the doors is inconsistent due to the noisy force sensor readings and fine-tuning of parameters. Before being able to guarantee that the FT-sensor implementation is indeed helping DORA to open the door, further and more extensive testing would be needed. For instance, more tests could be conducted to observe the outcome if the force feedback was excluded entirely from the algorithm so that the hinge position

was not updated depending on the force feedback. That would have enabled a more decisive conclusion about the results of implementing an FT-sensor for the door opening task and could have helped to optimise the different parameters. This test could not be conducted in this thesis due to time constraints.

However, from the test that has been conducted, it is possible to conclude that updating the hypothetical velocity, when including the force sensor readings in the door opening algorithm, allows DORA to open the door with larger offsets in the position of the hinge than when only updating the estimated hinge position based on the force feedback.

Overall the implementation of a FT-sensor and the door opening algorithm has been successful. DORA is, in most cases, able to autonomously open a door where the kinematics of the door are unknown; however, if the position of the initial door hinge is too inaccurate, the risk of not succeeding with opening the door exists.

## 5.3 Discussion

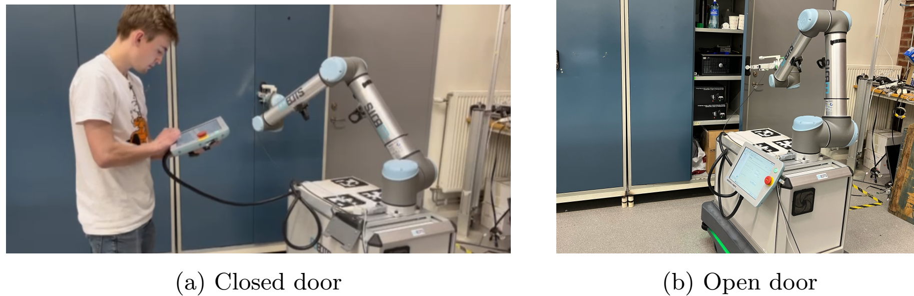
In its current state DORA would not be very suitable for work within the environments described in Section 2.3. There are very few safety precautions which would make DORA a potential hazard. The foundations for a robust and secure system has been created and implementing safety systems for work within sensitive areas can be done rather easily. The UR10 robotic arm has many safety measures built in [16], such as a force limiter that will automatically stop the arm if it encounters an unexpected amount of force, though this shouldn't be entirely relied upon as there could still be a risk of crushing injuries in certain spots, such as the door hinge. There is also the navigation system which though it has an obstacle detection system built in, the tests performed during previous years thesis [17] indicated that it can only detect obstacles in 2D space, and thus had trouble detecting some objects such as tables and stairs. These issues would need to be resolved if DORA is to be used in a real life setting.

The issues with the object detection described in Section 5.1 would need to be resolved and the door opening algorithm would need to be generalized so it can open doors in all directions, as well as all different types of doors. This step would necessitate the object detection system being able to differentiate between different types of doors which is beyond the capabilities of the current dataset being used, so this would require finding a new dataset or training one for this purpose if it cannot be found.

## 5.4 Future work

### 5.4.1 Extensive real world testing

Considering this thesis has been focused primarily on the simulation environment over the real world, there is a need to test the programs developed in a real world environment, and potentially adapt the programs so that they work in a real world setting. During this project one real world test has been performed, which came to the conclusion that some work will need to be done in order to adapt the system for proper real world usage.



(a) Closed door (b) Open door

Figure 23: Images from real life test

The adaptation of the system into a real world setting requires configuration of the camera constants in order to make sure the pose estimator is accurate. On the real robot the FT-sensor is located on the fingertips and not on the wrist of the robot as it is in the simulation, which necessitates adaptation in order to work properly.

The primary areas that require testing is the camera distortion and whether the inaccuracies seen in the simulation remains in the real world. or the camera constants need to be changed to reflect the properties of the real camera.

The door opening algorithm does also require more real world testing and tuning of the parameters. This is because it was observed during simulation that the trajectory of the end-effector could sometimes be a bit tight, which could result in a lot of force exerted on both DORA's arm as well as the door. In the real world this could potentially cause more problems than it did during simulation.

### 5.4.2 ROS 2.0

The current program has been written entirely in ROS Noetic Ninjemys which is the last long-term support release of ROS 1. Support for ROS Noetic is scheduled to end in May 2025, meaning that Noetic will no longer receive new updates after that date. This can lead to security vulnerabilities, missing out on

new features and compatibility issues. Thus it will be necessary to research and establish a plan for updating the project to ROS 2.0 within a timely manner.

### 5.4.3 Image Segmentation

The YOLO algorithm currently in use is very limited in it's scope. It can only output a single point with a width and height, creating a rectangular box. This doesn't work properly in cases where there is an angle between the object and the camera as seen in Figure 18. A potential fix to this would be to implement an image segmentation algorithm that can detect exactly where an object is located as demonstrated in Figure 24. Research will have to be done to determine if the extra processing time required for image segmentation is acceptable or whether another solution is better for this use case.

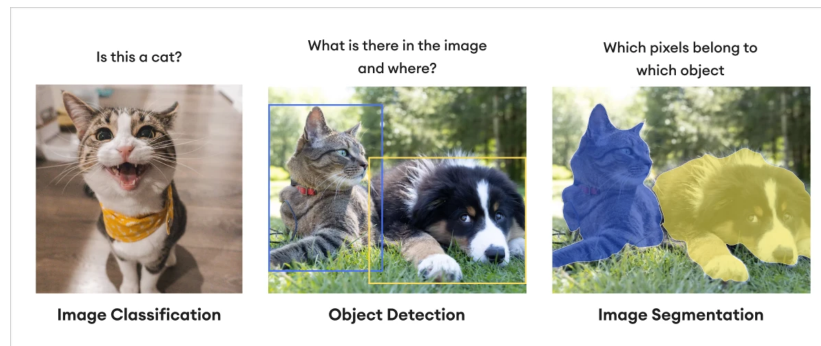



Figure 24: Examples of different image analysis methods [18]

### 5.4.4 Right or left opening door

With the current algorithm, in order to switch the way the door will open, left or right, the code must be manually adjusted. This is an future improvement that could be made to the algorithm, to make it decide on which side on the door the hinges are located and the adjust accordingly.

### 5.4.5 Filter for FT-sensor readings

The readings from the FT-sensor in the simulation environment CoppeliaSim contains a lot of noise which is having an effect on the door opening algorithm that is using said reading in several calculations. This could be resolved by implementing a filter to reduce the noise in the sensor and thus improving the accuracy of the calculations in the door opening algorithm as well as making it more consistent.

### 5.4.6 Rotating the end effector

Currently, the end effector rotates by a fixed number of degrees each time it moves to a new waypoint, ensuring it maintains its grasp on the door handle. However, this approach requires recalibration of the angle for each new door, and there is still a possibility that it will not rotate enough. Ideally, the rotation of the end effector should precisely align with the handle's orthogonal direction. Since the distance to each new waypoint varies, the amount the gripper should turn to maintain alignment should also vary accordingly. Unfortunately, this adaptive rotation mechanism could not be implemented due to time constraints. If this mechanism is added in the future, it could significantly improve the amount of times the door is successfully opened.

# 6 References

## References


[1] S Vineet et al. "Dynamic Analysis of Tracked Mobile Manipulator Used in Agriculture". In: *2021 IEEE 18th India Council International Conference (INDICON)*. 2021, pp. 1–6. DOI: 10.1109/INDICON52576.2021.9691678.

[2] Max Schwarz et al. "NimbRo Rescue: Solving Disaster-response Tasks with the Mobile Manipulation Robot Momaro". In: *Journal of Field Robotics* 34.2 (Nov. 2016), pp. 400–425. ISSN: 1556-4967. DOI: 10.1002/rob.21677. URL: http://dx.doi.org/10.1002/rob.21677.

[3] Arduengo M. "Labelled image dataset for door and handle detection." In: 2019. URL: https://github.com/MiguelARD/DoorDetect-Dataset.

[4] K. Nagatani and S.I. Yuta. "An experiment on opening-door-behavior by an autonomous mobile robot with a manipulator". In: *Proceedings 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human Robot Interaction and Cooperative Robots*. Vol. 2. Aug. 1995, 45–50 vol.2. DOI: 10.1109/IROS.1995.526137.

[5] Adrian Llopart, Ole Ravn, and Nils. A. Andersen. "Door and cabinet recognition using Convolutional Neural Nets and real-time method fusion for handle detection and grasping". In: *2017 3rd International Conference on Control, Automation and Robotics (ICCAR)*. Apr. 2017, pp. 144–149. DOI: 10.1109/ICCAR.2017.7942676.

[6] M. Arduengo, L. Sentis, and C. Torras. "Robust and adaptive door operation with a mobile robot." In: *Intelligent Service Robotics* 14.3 (2021), pp. 409-425 –425. ISSN: 18612784. URL: https://search.ebscohost.com/login.aspx?direct=true&db=edselc&AN=edselc.2-52.0-85106266022&site=eds-live&scope=site&authtype=guest&custid=s3911979&groupid=main&profile=eds.

[7] Y. Karayiannidis et al. ""Open sesame!" adaptive force/velocity control for opening unknown doors." In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on* (2012), pp. 4040–4047. ISSN: 978-1-4673-1737-5. URL: https://search.ebscohost.com/login.aspx?direct=true&db=edseee&AN=edseee.6385835&site=eds-live&scope=site&authtype=guest&custid=s3911979&groupid=main&profile=eds.

[8] "An Adaptive Control Approach for Opening Doors and Drawers Under Uncertainties". In: *DiVA* (March 22, 2016). URL: https://www.diva-portal.org/smash/record.jsf?dswid=6004&pid=diva2%3A914309&c=1&searchType=SIMPLE&language=en&query=An+Adaptive+Control+Approach+for+Opening+Doors+and+Drawers+under+Uncertainties&af=%5B%5D&aq=%5B%5B%5D%5D&aq2=%5B%5B%5D%5D&aqe=%5B%5D&

noOfRows=50&sortOrder=author_sort_asc&sortOrder2=title_sort_asc&onlyFullText=false&sf=all.

[9] G. Niemeyer and J.-J.E. Slotine. “A simple strategy for opening an unknown door”. In: *Proceedings of International Conference on Robotics and Automation*. Vol. 2. Apr. 1997, 1448–1453 vol.2. DOI: 10.1109/ROBOT.1997.614341.

[10] *Image of dog and bike object detection*. Accessed: May 10, 2024. URL: https://github.com/pjreddie/darknet/blob/master/data/dog.jpg.

[11] Hongjun Xing et al. “An admittance-controlled wheeled mobile manipulator for mobility assistance: Human–robot interaction estimation and redundancy resolution for enhanced force exertion ability”. In: *Mechatronics* 74 (2021), p. 102497. ISSN: 0957-4158. DOI: https://doi.org/10.1016/j.mechatronics.2021.102497. URL: https://www.sciencedirect.com/science/article/pii/S0957415821000076.

[12] “Mobile manipulators: The intelligent production for your factory.” In: (February 16, 2022). URL: https://robotnik.eu/mobile-manipulators-the-intelligent-production-for-your-factory/.

[13] Jayant Menon. “Why the Fourth Industrial Revolution could spell more jobs – not fewer”. In: *Worl Economic Forum* (September 17, 2019). URL: https://www.weforum.org/agenda/2019/09/fourth-industrial-revolution-jobs/.

[14] Arun Ponnusamy. “YOLO Object Detection with OpenCV and Python”. In: *Towards Data Science* (Aug. 2018). URL: https://towardsdatascience.com/yolo-object-detection-with-opencv-and-python-21e50ac599e9.

[15] Motion Planning API. *moveit_tutorials Noetic documentation*. Accessed May 10, 2024. Year Accessed. URL: https://moveit.github.io/moveit_tutorials/doc/motion_planning_api/motion_planning_api_tutorial.html.

[16] Universal Robots. *UR10 User Manual*. Accessed: 2024-05-10. URL: https://automationdistribution.com/content/Universal-Robots-UR10-User-Manual.pdf.

[17] Lydia Andersson et al. “DORA - Dexterous Robot Assistant Software Implementation of Mobile Manipulators for Pick-and-Place Tasks”. Examensarbete på kandidatnivå. Bachelor Thesis. Chalmers University of Technology, 2022.

[18] *Image Segmentation Detailed Overview*. Nov. 2023. URL: https://www.superannotate.com/blog/image-segmentation-for-machine-learning.

# 7 Appendix

## 7.1 Disclaimer on usage of ChatGPT and other LLMs

ChatGTP, a LLM (Large Language Models), was used when troubleshooting code and to answer general questions about the topic of the work. It was used much like one might use stack overflow or google. No code was written by the LLM, it only corrected mistakes.

LLMs were not used to write any part of this report.

## 7.2 Images

### 7.2.1 Simulation scenes from 3.1

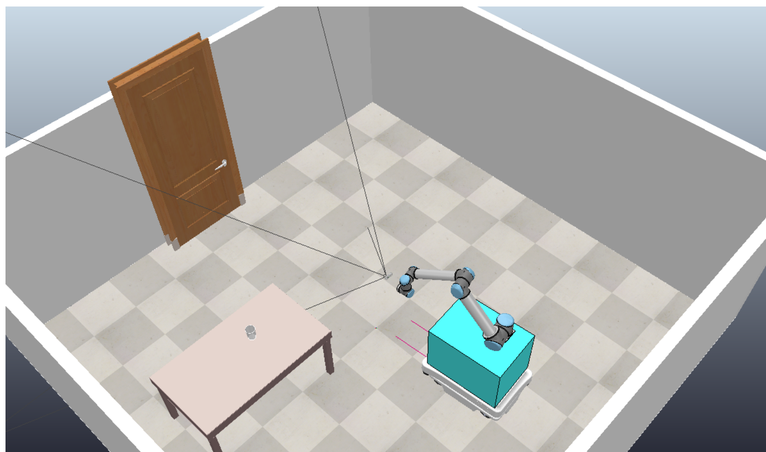


Figure 25: "Workshop" environment, used for small navigation, object detection and manipulation tests.

Figure 26: "Door detect" environment, used for testing DORA's door and door handle detection algorithm.

Figure 27: "Door opening" environment, used for developing the door opening algorithm.

## 7.3 Code

```
def orthogonalize(r):
    R = array([[0, -1, 0],
               [1,  0, 0],
               [0,  0, 1]])
    return R @ r
def get_rot_matrix(theta):
    R = array([[cos(theta), -sin(theta), 0],
               [sin(theta),  cos(theta), 0],
               [0, 0, 1]])
    return R

class DoorOpeningTask:
    alpha = constant
    beta = constant
    gamma = constant

    k_f = constant
    k_I = constant

    dt=constant

    desired_tangent_v = constant
    desired_radial_force = constant

    def init(initial_p0_estimate):
        self.p0_estimate = initial_p0_estimate
        self.delta_estimated_force_radial_integral = 0
        self.curr_vel = 0

    def get_vel(pe, force):
        estimated_r = self.p0_estimate - pe
        estimated_r_normalized = normalize(estimated_r)
        estimated_force_radial = estimated_r_normalized.T @ force

        delta_estimated_force_radial = estimated_force_radial -
    desired_radial_force
        self.delta_estimated_force_radial_integral += dt *
    delta_estimated_force_radial
        force_feedback = delta_estimated_force_radial  + beta *
    self.delta_estimated_force_radial_integral

        vref = orthoganalize(estimated_r_normalized) *
    desired_tangent_v + alpha * estimated_r_normalized *
    force_feedback

        estimated_p0_dot = gamma / norm(estimated_r) * (k_f *
    delta_estimated_force_radial + k_I *
    delta_estimated_force_radial_integral) * self.curr_vel
        self.p0_estimate +=  estimated_p0_dot * dt
        self.curr_vel = vref
        return vref

delta_theta = constant
p0, pe, rot = estimate_door(...)
### Move robot hand to door
move_ee(pe,rot)
task = DoorOpeningTask(p0)
```

```
while is not task.complete:
    pe = get_ee_position()
    force = get_force()
    vref = get_vel(pe,force)
    new_pe = pe + dt * vref # may need adding a gain value here.
    new_rot = rot @ get_rot_matrix(delta_theta) # or -delta_theta
    move_ee(new_pe,new_rot)
    rot = new_rot
```

Listing 1: Pseudocode for door opening algorithm (provided by supervisor)

```
    from geometry_msgs.msg import Pose, PoseArray
from std_msgs.msg import Header
import rospy
import geometry_msgs.msg
import  tf2_ros
import numpy as np
from rg2_control import *
from visualization_msgs.msg import Marker
from geometry_msgs.msg import Point, Vector3
from std_msgs.msg import ColorRGBA
from tf.transformations import quaternion_from_matrix

tf_buffer = None # Create TF buffer
tf_listener = None

#global varibles
pose_array_msg = PoseArray()
global position_Z

# Constants
alpha = 0.05
beta = 0.1
gamma = 0.5
k_f = 0.1
k_I = 0.05
dt = 0.1
desired_tangent_v = 0.5
desired_radial_force = 2

global_orientation_xyz = []

def orthogonalize(r):
    R = np.array([[0, -1, 0],
                  [1,  0, 0],
                  [0,  0, 1]])
    return R @ r

def get_rot_matrix(theta):
    R = np.array([[np.cos(theta), -np.sin(theta), 0, 0],
                  [np.sin(theta),  np.cos(theta), 0, 0],
                  [0, 0, 1, 0],
                  [0, 0, 0, 1]])
    return R

def create_vector_marker(id, pe_pose, vector, color, timestamp):
    header = Header()
    header.stamp = timestamp
    header.frame_id = "base_link_ur10_inertia"
```

```
    marker = Marker()
    marker.header = header
    marker.ns = "vector"
    marker.id = id
    marker.type = Marker.ARROW
    marker.action = Marker.ADD
    marker.pose.position = pe_pose  # Set the base of the arrow to
    the end effector position
    marker.pose.orientation.w = 1
    marker.scale = Vector3(0.05, 0.05, 0.05)  # Arrow dimensions
    marker.color = color

    # Compute the endpoint of the arrow relative to the base (end
    effector position)
    endpoint = Point(pe_pose.x + vector[0], pe_pose.y + vector[1],
    0)
    marker.points.append(pe_pose)
    #marker.points.append(Point(0, 0, 0))  # Start point (base of
    the arrow)
    marker.points.append(endpoint)  # End point

    return marker

class DoorOpeningTask:

    def __init__(self, initial_p0_estimate):
        self.p0_estimate = np.array(initial_p0_estimate)  # Hinge
    initial position
        self.delta_estimated_force_radial_integral = 0
        self.curr_vel = 0  # Current velocity

        self.alpha = alpha
        self.beta = beta
        self.gamma = gamma

        self.k_f = k_f
        self.k_I = k_I
        self.dt = dt

        self.desired_radial_force = desired_radial_force
        self.desired_tangent_v = desired_tangent_v

    def get_vel(self, pe, force):
        print(f'force is {force}')
        estimated_r = self.p0_estimate - pe
        estimated_r[2] = 0
        estimated_r_normalized = estimated_r / np.linalg.norm(
    estimated_r)
        print(estimated_r_normalized)

        estimated_force_radial = np.dot(estimated_r_normalized,
    force)

        print(f'estimated_force_radial is \n {
    estimated_force_radial} estimated_force_radial_test2 {
    estimated_force_radial_test2}')

        delta_estimated_force_radial = estimated_force_radial -
    self.desired_radial_force
        self.delta_estimated_force_radial_integral += self.dt *
```

```
delta_estimated_force_radial
        force_feedback = delta_estimated_force_radial + self.beta *
    self.delta_estimated_force_radial_integral

        vref = -orthogonalize(estimated_r_normalized) *
    desired_tangent_v + alpha * estimated_r_normalized *
    force_feedback
        vref_without_force = -orthogonalize(estimated_r_normalized)
     * desired_tangent_v

        print(f'vref is \n {vref} \n \n vref_without_force \n {
    vref_without_force}')
        #vref = np.cross(estimated_r_normalized, np.array([0, 0,
    1])) * self.desired_tangent_v + self.alpha *
    estimated_r_normalized * force_feedback

        estimated_p0_dot = self.gamma / np.linalg.norm(estimated_r)
     * (self.k_f * delta_estimated_force_radial + self.k_I * self.
    delta_estimated_force_radial_integral) * self.curr_vel
        self.p0_estimate += estimated_p0_dot * self.dt
        self.curr_vel = np.linalg.norm(vref)

        print(f'estimated_p0_dot is {estimated_p0_dot}')

        # Publish vref and vref_without_force vectors as markers
        pe_pose = Point(pe[0], pe[1], pe[2])
        timestamp = rospy.Time.now()  # Get current time stamp
        vref_marker = create_vector_marker(0, pe_pose, vref,
    ColorRGBA(0.0, 1.0, 0.0, 1.0), timestamp)  # Green
        vref_without_force_marker = create_vector_marker(1, pe_pose
    , vref_without_force, ColorRGBA(1.0, 0.0, 0.0, 1.0), timestamp)
      # Red

        # Publish markers
        marker_pub = rospy.Publisher("visualization_marker", Marker
    , queue_size=10)
        marker_pub.publish(vref_marker)
        marker_pub.publish(vref_without_force_marker)

        return vref

    def not_complete(self):
        # Compute the angle between the initial and current vectors
        dot_product = np.dot(initial_handle_to_hinge_vector,
    current_handle_to_hinge_vector)
        unit_vectors = np.linalg.norm(
    initial_handle_to_hinge_vector) * np.linalg.norm(
    current_handle_to_hinge_vector)
        angle = np.arccos(dot_product / unit_vectors)
        angle_threshold = np.pi / 2  # 90 degrees
        print(f'angle is: {angle*180/np.pi}')
        if angle >= angle_threshold:
            print('door is open, task is complete')
            #task complete return false
            return False
        else:
            return True

def get_ee_position():
    '''Get the end effector position (grasp_center) in our case,
    return as an array)'''
```

```
    ee_transform = tf_buffer.lookup_transform('
    base_link_ur10_inertia', 'grasp_center', rospy.Time(0), rospy.
    Duration(1.0))
    ee = ee_transform.transform.translation

    return [ee.x, ee.y, ee.z]

def estimate_door():
    '''Get the intial p0 , pe  where pe her e is the point on the
    handle, p0 is the handles point and rot  '''

    traj_msg = rospy.wait_for_message("/traj", geometry_msgs.msg.
    PoseArray, timeout=10.0)

    wp_init = [pose.position for pose in traj_msg.poses]

    pe = [wp_init[0].x, wp_init[0].y, wp_init[0].z]

    p0 = [wp_init[1].x,wp_init[1].y, wp_init[1].z]
    rot = get_rot_matrix(delta_theta)

    global initial_handle_to_hinge_vector
    global current_handle_to_hinge_vector
    initial_handle_to_hinge_vector = np.array(p0) - np.array(pe)
    current_handle_to_hinge_vector = initial_handle_to_hinge_vector

    global global_orientation_xyz
    global position_z
    position_z = wp_init[0].z
    global_orientation_xyz = [traj_msg.poses[0].orientation.x,
    traj_msg.poses[0].orientation.y, traj_msg.poses[0].orientation.
    z]

    # print(f'rot is {rot} ')
    # print(f'pe {pe}')
    # print(f'po = {p0}')
    # print(f'global_orientation_xyz{global_orientation_xyz}' )

    return pe, p0, rot

def move_ee(pe, rot, first_time=False):
    header = Header()
    header.stamp = rospy.Time.now()
    header.frame_id = "base_link_ur10_inertia"

    pose = Pose()

    if first_time:
        pose.position.x = pe[0]
        pose.position.y = pe[1]
        pose.position.z = position_z
        pose.orientation.x = global_orientation_xyz[0]
        pose.orientation.y = global_orientation_xyz[1]
        pose.orientation.z = global_orientation_xyz[2]
        pose.orientation.w = 1

        pose_array_msg.poses.append(pose)
        pose_array_msg.header = header
    else:
        pose.position.x = pe[0]
        pose.position.y = pe[1]
```

```
        pose.position.z = pe[2]

        # Print the shape of the rotation matrix
      #  print("Shape of rot matrix:", rot.shape)

        # Convert rotation matrix to quaternion
        quaternion = quaternion_from_matrix(rot)
        pose.orientation.x = quaternion[0]
        pose.orientation.y = quaternion[1]
        pose.orientation.z = quaternion[2]
        pose.orientation.w = quaternion[3]

    pose_array_msg.poses.append(pose)
    pose_array_msg.header = header

    execute_wp_movement(pose_array_msg)

def execute_wp_movement(msg):
    print('i am now pub to viz')
    for i in range(3):
        pub = rospy.Publisher('execute_wp_movement', geometry_msgs.
    msg.PoseArray , queue_size=10)
        pubViz = rospy.Publisher('viz_cal_wp', geometry_msgs.msg.
    PoseArray , queue_size=10)
        pubViz.publish(msg)
        pub.publish(msg)
        rospy.sleep(1)
    global pose_array_msg
    pose_array_msg= PoseArray()

def get_force():
    force_msg = rospy.wait_for_message("/force_sensor_data",
    geometry_msgs.msg.Wrench, timeout=10.0)
    #print(f'force msg is {force_msg}')
    return [force_msg.force.x, force_msg.force.y, force_msg.force.z
    ]

delta_theta =  np.pi/20

if __name__ == '__main__':
    rospy.init_node('tf_listener_node')  # Initialize ROS node
    tf_buffer = tf2_ros.Buffer()  # Create TF buffer
    rospy.sleep(0.1)
    tf_listener = tf2_ros.TransformListener(tf_buffer)  # Create TF
     listener
    pe, p0, rot = estimate_door()
   # print('i fetched estimate door')
    move_ee(pe,rot, first_time=True)
    #print(f' rot is {rot}')

     # Call function for gripper here
    ctrlPub = control_pub()
    rospy.sleep(0.1)
    close_gripper(ctrlPub)
    rospy.sleep(2.) #wait for gripper to close

    task = DoorOpeningTask(p0)
    while task.not_complete():
        pe = get_ee_position()  # Update pe with the current end
    effector position
        rospy.sleep(0.1)
```

```
        print(f'Pe is {pe}')

        force = get_force()
        force[2] = 0
        vref = task.get_vel(pe, force)
        new_pe = pe  + (vref * dt) # may need adding a gain value
    here.
        print(f'new Pe is {new_pe}')
        new_rot = rot @ get_rot_matrix(delta_theta)  # Calculate
    new rotation
        move_ee(new_pe, new_rot)  # Move to the new position
        rospy.sleep(1)
        new_pe = get_ee_position()
        current_handle_to_hinge_vector = np.array(p0) - np.array(
    new_pe) # get current vector from ee to hinge
        rot = new_rot #update the old rot.


    rospy.spin()
```

Listing 2: force_feedback_planner.py the door opening program

```
from __future__ import print_function
from transform_interface import TransformInterface
import sys
import rospy
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg

from rg2_control import open_gripper, close_gripper, control_pub

try:
    from math import pi, tau, dist, fabs, cos
except:  # For Python 2 compatibility
    from math import pi, fabs, cos, sqrt

    tau = 2.0 * pi

    def dist(p, q):
        return sqrt(sum((p_i - q_i) ** 2.0 for p_i, q_i in zip(p, q
    )))


from std_msgs.msg import String
from moveit_commander.conversions import pose_to_list


moveit_commander.roscpp_initialize(sys.argv)
rospy.init_node("move_group_python_interface", anonymous=True)


robot = moveit_commander.RobotCommander()

scene = moveit_commander.PlanningSceneInterface()

group_name = "arm"
move_group = moveit_commander.MoveGroupCommander(group_name)
```

```
display_trajectory_publisher = rospy.Publisher(
    "/move_group/display_planned_path",
    moveit_msgs.msg.DisplayTrajectory,
    queue_size=20,
)

def transform_wp_points(wp_list):
    print(f' wp_list is {wp_list} ')

    tf_inter= TransformInterface()
    t1 = tf_inter.lookup_numpy_transform('base_footprint','
    base_link_ur10_inertia')
    converted_poses = []

    for wp in wp_list:
        t2 = tf_inter.ros_pose_to_matrix4x4(wp)
        updated_wp = t1 @ t2
        converted_poses.append(tf_inter.matrix4x4_to_ros_pose(
    updated_wp))
        print(f'converted is {converted_poses}')
    return converted_poses



# Callback function for receiving trajectory messages from
    CoppeliaSim
def move_with_wp(traj_msg):
    waypoints = []

    # Extract waypoints from the trajectory message and transform
    from basefoot print to base_link_inertia_ur10
    wp_l = [p for p in traj_msg.poses]
    wp_converted = transform_wp_points(wp_l)
    waypoints = wp_converted

    # Compute Cartesian path for the wp
    (plan, fraction) = move_group.compute_cartesian_path(
        waypoints, 0.01, 0.0  # waypoints to follow, eef_step,
    jump_threshold
    )

    # Execute the Cartesian path for the first waypoint
    move_group.execute(plan, wait=True)

    # Display planned path in RViz for the second part of the
    trajectory
    display_trajectory = moveit_msgs.msg.DisplayTrajectory()
    display_trajectory.trajectory_start = robot.get_current_state()
    display_trajectory_publisher.publish(display_trajectory)



#Wait for a message on the '/traj' topic with a timeout of 10
    seconds
traj_msg = rospy.wait_for_message("/execute_wp_movement",
    geometry_msgs.msg.PoseArray, timeout=20.0)

traj_old = 0

#Only move once if same message is published multiple times
```

```
while traj_msg != traj_old:
    move_with_wp(traj_msg)
    traj_old = traj_msg
    traj_msg = rospy.wait_for_message("/execute_wp_movement",
    geometry_msgs.msg.PoseArray, timeout=60.0)



rospy.spin()
```

Listing 3: moveit_arm_controll.py the program for controlling the arm

```

import rospy
from std_msgs.msg import Int32
from time import sleep

def open_gripper(pub):
    msg = Int32()
    msg.data = 1
    pub.publish(msg)
    rospy.loginfo("Gripper is opening...")

def close_gripper(pub):
    msg = Int32()
    msg.data = 0
    pub.publish(msg)
    rospy.loginfo("Gripper is closing...")

def control_pub():
    control_pub = rospy.Publisher('gripper_control', Int32,
    queue_size=10)
    rate = rospy.Rate(0.1)  # Adjust the rate as needed

    return control_pub
```

Listing 4: rg2_control.py the program for controlling the gripper

```
    import rospy
    from sensor_msgs.msg import Image, CameraInfo
    from cv_bridge import CvBridge
    from geometry_msgs.msg import Pose, Point, Quaternion,
    TransformStamped, PoseStamped
    import tf2_ros
    import tf2_geometry_msgs
    import cv2
    import numpy as np
    import os
    import transforms3d.quaternions as quat


    #Global variables, classes and COLORS are used to pass info
    between the yolo() and draw_bounding_box functions
    #Delay is the delay between different images in ms
    classes = None
    COLORS = None
    delay = -1
```

```
image_data = None
depth_data = None
average_counter = 0
average_needed = 5
coordinates = [[], [], [], []]
movePub = None
tf_puber = None
tf_listener = None
tf_buffer = None
K = None
doordone = False
latest_image = None



"""
The following functions were adapted from the article "YOLO
Object Detection with OpenCV and Python" by Arun Ponnusamy,
published on Towards Data Science:

get_output_layers():
get_output_layers() function gives the names of the output
layers. An output layer is not connected to any next layer.

yolo():
yolo() short for "You only look once" is an object detection
method. The yolo function has been directly adopted from
Ponnusamys article

You can find the original article here: https://
towardsdatascience.com/yolo-object-detection-with-opencv-and-
python-21e50ac599e9


The dataset used in this project is the DoorDetect dataset. It
consists of weightings for image detection of doors and
doorhandles.
The dataset is publicly available and can be found at https://
github.com/MiguelARD/DoorDetect-Dataset/tree/master.

Full reference in our final report

"""
def get_output_layers(net):
    layer_names = net.getLayerNames()
    #print(net.getUnconnectedOutLayers())
    # Get the indices of the output layers
    output_layer_indices = net.getUnconnectedOutLayers()

    # Convert output layer indices to layer names
    output_layers = [layer_names[i[0] - 1] for i in
output_layer_indices]

    return output_layers



def moveRobothandle():
    global movePub, tf_buffer, doordone
    transform = tf_buffer.lookup_transform('base_link', '
door_handle', rospy.Time(0))
```

```
    robot_pose = tf_buffer.lookup_transform('base_footprint', '
base_link', rospy.Time(0))
    map_pose = tf_buffer.lookup_transform('map', 'base_link',
rospy.Time(0))


    x = transform.transform.translation.x
    y = transform.transform.translation.y
    curdist = np.sqrt(x**2+y**2)
    targetdist = 1.2
    norm_x = x / curdist
    norm_y = y / curdist
    target_x = x - norm_x * targetdist
    target_y = y - norm_y * targetdist - 0.5
    target_z = 0


    moveGoal = PoseStamped()
    moveGoal.header.frame_id = "base_link"
    moveGoal.header.stamp = rospy.Time.now()
    moveGoal.pose.position.x = target_x
    moveGoal.pose.position.y = target_y
    moveGoal.pose.position.z = target_z
    moveGoal.pose.orientation = robot_pose.transform.rotation
    transformed_goal = tf2_geometry_msgs.do_transform_pose(
moveGoal, map_pose)
    transformed_goal.header.frame_id = "map"

    movePub.publish(transformed_goal)

    rospy.loginfo("Handle detected, moving closer")


"""
Moves the robot closer to the door. Liable to chan ge depending
 on how our tests go.
"""
def moveRobot(transform):
    global movePub, tf_buffer, doordone
    robot_pose = tf_buffer.lookup_transform('base_footprint', '
base_link', rospy.Time(0))
    map_pose = tf_buffer.lookup_transform('map', 'base_link',
rospy.Time(0))


    x = transform.transform.translation.x
    y = transform.transform.translation.y
    curdist = np.sqrt(x**2+y**2)
    targetdist = 2
    norm_x = x / curdist
    norm_y = y / curdist
    target_x = x - norm_x * targetdist
    target_y = y - norm_y * targetdist
    target_z = 0


    moveGoal = PoseStamped()
    moveGoal.header.frame_id = "base_link"
    moveGoal.header.stamp = rospy.Time.now()
    moveGoal.pose.position.x = target_x
    moveGoal.pose.position.y = target_y
```

```
        moveGoal.pose.position.z = target_z
        moveGoal.pose.orientation = robot_pose.transform.rotation
        transformed_goal = tf2_geometry_msgs.do_transform_pose(
moveGoal, map_pose)
        transformed_goal.header.frame_id = "map"

        movePub.publish(transformed_goal)

        rospy.loginfo("Door successfully detected, moving closer")
        rospy.sleep(5)
        rospy.loginfo("Starting detection of the handle")
        doordone = True










"""
Publishes the inputed pose to /tf. frame is the name of the
object that will be used in /tf
"""
def tf_broadcast(pose, frame):
    global tf_puber, tf_buffer, tf_listener

    transform = TransformStamped()

    transform.header.stamp = pose.header.stamp
    transform.header.frame_id = pose.header.frame_id
    transform.child_frame_id = frame

    transform.transform.translation = pose.pose.position
    transform.transform.rotation = pose.pose.orientation
    #rospy.loginfo(transform)
    tf_puber.sendTransform(transform)

"""
Calculates a quaternion object from three inputet coordinates
"""
def getQuaternion(coords):
    A = coords[0]
    B = coords[1]
    C = coords[2]
    D = coords[3]

    # create plane
    AB = np.array(B) - np.array(A)
    AD = np.array(D) - np.array(A)
    normal = np.cross(AB, AD)
    normal_unit = normal / np.linalg.norm(normal)

    k = np.array([0, 0, 1])  # Z-axis
    axis = np.cross(k, normal_unit)
    axis_unit = axis / np.linalg.norm(axis)

    cos_theta = np.dot(k, normal_unit)
```

```
        theta = np.arccos(cos_theta)

        w = np.cos(theta / 2)
        xyz = axis_unit * np.sin(theta / 2)

        return Quaternion(w=w,x=xyz[0],y=xyz[1],z=xyz[2])

    """
    Gets an average for the inputet coordinates.
    """

    def getCenter(coords):
        avg_x = sum(p[0] for p in coords) / len(coords)
        avg_y = sum(p[1] for p in coords) / len(coords)
        avg_z = sum(p[2] for p in coords) / len(coords)
        return Point(x=avg_x, y=avg_y, z=avg_z)

    """
    get door pose calculates the position of the door and prepares
    it to be published to /tf.
    It does the same for the door edges to be used later on when
    calculateing the hinge location.
    """
    def getdoorpose(coords):
        global tf_buffer, tf_puber, dooredge1, dooredge2
        quaternion = getQuaternion(coords)
        position = getCenter(coords)

        pose = PoseStamped()
        pose.header.frame_id = "camera_depth_optical_frame"
        pose.header.stamp = rospy.Time.now()
        pose.pose.position = position
        pose.pose.orientation = quaternion
        tf_broadcast(pose, "detected_door")


        map_pose = tf_buffer.lookup_transform('map', '
    camera_depth_optical_frame', rospy.Time(0))

        pose2 = PoseStamped()
        pose2.header.stamp = rospy.Time.now()
        pose2.header.frame_id = "camera_depth_optical_frame"
        pose2.pose.orientation = quaternion
        pose2.pose.position = getCenter([coords[0],coords[2]])   #
    get center of A and C
        #rospy.loginfo(pose2)
        pose2 = tf2_geometry_msgs.do_transform_pose(pose2, map_pose
    )
        pose2.header.frame_id = "map"
        #rospy.loginfo(pose2)
        tf_broadcast(pose2,"door_edge1")
        #dooredge1 = pose2



        pose3 = PoseStamped()
        pose3.header.stamp = rospy.Time.now()
        pose3.header.frame_id = "camera_depth_optical_frame"
        pose3.pose.orientation = quaternion
        pose3.pose.position = getCenter([coords[1], coords[3]])  #
    get center of A and C
```

```
        pose3 = tf2_geometry_msgs.do_transform_pose(pose3, map_pose
)
        pose3.header.frame_id = "map"
        tf_broadcast(pose3, "door_edge2")
        #dooredge2 = pose3



    # check if there is a transform between base_link and door
    rospy.sleep(1)
    if tf_buffer.can_transform('base_link', 'detected_door',
rospy.Time(0)):
        whole_transform = tf_buffer.lookup_transform('base_link
', 'detected_door', rospy.Time(0))
        moveRobot(whole_transform)
        # rospy.loginfo(whole_transform)
    else:
        rospy.loginfo(
            "Unable to find transform between base_link and
detected_door. Make sure dora simulation_moveit.launch is
running")

    return pose
    #rospy.loginfo(str(pose_msg))




"""
cameraValscalc uses the camera info to calculate the cartesian
coordinates from the coordinates in the depth image.
These cartesian coordinates are relative to the
camera_depth_optical_frame in /tf
"""


def cameraValscalc(x,y,depthimg):
    global K
    fx = K[0]  # Focal length in x
    fy = K[4]  # Focal length in y
    cx = K[2]  # Principal point in x
    cy = K[5]  # Principal point in y

    x = x//4        #convert from normal image resolution to
depth image resolution
    y = y//4
    Z = depthimg[x,y]  # Depth value

    # Convert to Cartesian coordinates
    X = (x - cx) * Z / fx
    Y = (y - cy) * Z / fy

    return [X,Y,Z]

def euclidean_distance(p1, p2):
    return np.sqrt((p1.x - p2.x)**2 + (p1.y - p2.y)**2 + (p1.z
- p2.z)**2)


"""
This calculates the location of the handle object from the
```

```
depth image.
It scans the region detected and finds the point closest to the
 camera
Then gets the cartesian coordinates, puts them in a pose and
publishes to /tf

Also finds the hinge of the door by comparing with the door
edges
Note the hinge doesnt mean the actual physical hinge, but the
axis around which the door will turn
Thus it has the same Z value as the handle.
"""
def handleHandleObject(img, class_id, confidence, x, y, w, h,
depth):
    global tf_buffer, tf_puber
    #cv2.rectangle(img, (x, y), (x_plus_w, y_plus_h), (255,
255, 0), 2)
    x_scaled = x // 4
    y_scaled = y // 4
    w_scaled = w // 4
    h_scaled = h // 4
    handleImage = depth[y_scaled:y_scaled + h_scaled, x_scaled:
x_scaled + w_scaled]

    best = 1000
    for i, row in enumerate(handleImage):
        for j, element in enumerate(row):
            if element < best:
                best = element
                [X,Y] = [j,i]



    X = (X + x_scaled) * 4
    Y = (Y + y_scaled) * 4

    cv2.circle(img, (X,Y), 5, (255,0,0), thickness=-1)  #
thickness=-1 to fill the circle

    # Display the image
    cv2.imshow('Red Circle', img)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

    [X,Y,Z] = cameraValscalc(X,Y,depth)


    map_pose = tf_buffer.lookup_transform('map', 'base_link',
rospy.Time(0))
    robot_pose = tf_buffer.lookup_transform('base_link', '
camera_depth_optical_frame', rospy.Time(0))
    dooredge1 = tf_buffer.lookup_transform('map', 'door_edge1',
 rospy.Time(0))
    dooredge2 = tf_buffer.lookup_transform('map', 'door_edge2',
 rospy.Time(0))

    #rospy.loginfo(dooredge1)


    pose = PoseStamped()
    pose.header.stamp = rospy.Time.now()
```

```
    pose.header.frame_id = "camera_depth_optical_frame"

    pose.pose.position.x = X
    pose.pose.position.y = Y
    pose.pose.position.z = Z

    #Placeholder values for orientation, they get accurate
values after transformation
    pose.pose.orientation.x = 0.0
    pose.pose.orientation.y = 0.0
    pose.pose.orientation.z = 0.0
    pose.pose.orientation.w = 1.0

    #transform pose to be relative to map
    pose = tf2_geometry_msgs.do_transform_pose(pose, robot_pose
)
    pose = tf2_geometry_msgs.do_transform_pose(pose, map_pose)
    pose.header.frame_id = "map"

    #copy orientation from dooredge1, in order to work with the
 control system the orientation has to be like below
    original_quaternion = [
        dooredge1.transform.rotation.w,
        dooredge1.transform.rotation.x,
        dooredge1.transform.rotation.y,
        dooredge1.transform.rotation.z
    ]

    # Define the individual rotations
    # 90 degrees counter-clockwise rotation along the red axis
    rot1 = quat.axangle2quat([0, 1, 0], np.pi / 2)  # Rotate 90
 degrees around X-axis
    # 90 degrees clockwise rotation along the green axis
    rot2 = quat.axangle2quat([1, 0, 0], np.pi / 2)  # Rotate
-90 degrees around Y-axis

    # Combine the rotations
    combined_rot = quat.qmult(rot2, rot1)

    # Apply the combined rotation to the original quaternion
    transformed_quaternion = quat.qmult(combined_rot,
original_quaternion)
    # Extract the transformed components
    pose.pose.orientation.w = transformed_quaternion[0]
    pose.pose.orientation.x = transformed_quaternion[1]
    pose.pose.orientation.y = transformed_quaternion[2]
    pose.pose.orientation.z = transformed_quaternion[3]
    tf_broadcast(pose, "door_handle")

    distance_to_dooredge1 = euclidean_distance(pose.pose.
position, dooredge1.transform.translation)
    distance_to_dooredge2 = euclidean_distance(pose.pose.
position, dooredge2.transform.translation)

    doorhinge = PoseStamped()
    doorhinge.header.stamp = rospy.Time.now()

    doorhinge.header.frame_id = "map"
    if distance_to_dooredge1 > distance_to_dooredge2:
        rospy.loginfo("edge 2 closer, hinge on edge 1")
        doorhinge.pose.position.x = dooredge1.transform.
```

```
translation.x
        doorhinge.pose.position.y = dooredge1.transform.
translation.y
        doorhinge.pose.position.z = pose.pose.position.z
        doorhinge.pose.orientation = pose.pose.orientation
    else:
        rospy.loginfo("edge 1 closer, hinge on edge 2")
        doorhinge.pose.position.x = dooredge2.transform.
translation.x
        doorhinge.pose.position.y = dooredge2.transform.
translation.y
        doorhinge.pose.position.z = pose.pose.position.z
        doorhinge.pose.orientation = pose.pose.orientation
    tf_broadcast(doorhinge, "door_hinge")
    rospy.loginfo("Sucessfully detected door hinge and door
handle. Program succesful")
    rospy.sleep(0.1)
    moveRobothandle()
    while(1):
        rospy.sleep(1)
        doorhinge.header.stamp = rospy.Time.now()
        pose.header.stamp = rospy.Time.now()
        tf_broadcast(doorhinge, "door_hinge")
        tf_broadcast(pose, "door_handle")
    #rospy.loginfo(coords)


"""
This function is used to handle detected door objects, when a
door is found by yolo it will add its corners to an average
When it has collect [average_needed] amount of them it computes
 an average and then sends it to the getpose() function
"""
def handleDoorObject(img, class_id, confidence, x, y, x_plus_w,
 y_plus_h, depth):
    global average_counter, average_needed, coordinates
    label = str(classes[class_id])

    #Append new values to coordinates
    coordinates[0].append(cameraValscalc(x,y, depth))
    coordinates[1].append(cameraValscalc(x_plus_w, y, depth))
    coordinates[2].append(cameraValscalc(x, y_plus_h, depth))
    coordinates[3].append(cameraValscalc(x_plus_w, y_plus_h,
depth))

    #When we have collected enough values we take the averages
    if len(coordinates[0]) == average_needed:

        #take the average of every coordinate for every corner
        meanCoord = []
        for i in coordinates:
            xtot = 0
            ytot = 0
            ztot = 0
            for n in i:
                xtot = xtot + n[0]
                ytot = ytot + n[1]
                ztot = ztot + n[2]
            xmean = np.round(xtot / len(i),2)
            ymean = np.round(ytot / len(i),2)
            zmean = np.round(ztot / len(i),2)
```

```
                meanCoord.append([xmean,ymean,zmean])



        """ If you want to display the bounding box of the door
        #following is for producing the bounding box with the
coordinates for the corners and displaying the image
        coords1 = str(meanCoord[0][0]) + " " + str(meanCoord
[0][1]) + " " + str(meanCoord[0][2])
        coords2 = str(meanCoord[1][0]) + " " + str(meanCoord
[1][1]) + " " + str(meanCoord[1][2])
        coords3 = str(meanCoord[2][0]) + " " + str(meanCoord
[2][1]) + " " + str(meanCoord[2][2])
        coords4 = str(meanCoord[3][0]) + " " + str(meanCoord
[3][1]) + " " + str(meanCoord[3][2])

        cv2.rectangle(img, (x, y), (x_plus_w, y_plus_h), (255,
255, 0), 2)

        #cv2.putText(img, coords1, (x - 10, y - 10), cv2.
FONT_HERSHEY_SIMPLEX,0.5,(0,255,0),2)
        #cv2.putText(img, coords2, (x_plus_w - 10, y - 10), cv2
.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)
        #cv2.putText(img, coords3, (x - 10, y_plus_h + 10), cv2
.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)
        #cv2.putText(img, coords4, (x_plus_w - 10, y_plus_h +
10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

        #display output image
        cv2.imshow("coordinates", img)

        # wait until any key is pressed
        cv2.waitKey(delay)

        # reset global variables
        cv2.destroyAllWindows()
        """
        getdoorpose(meanCoord)

        coordinates = [[], [], [], []]



def yolo(img):
    weight = 'yolostuff/yolo-obj.weights'
    config = 'yolostuff/yolo-obj.cfg'
    class1 = 'yolostuff/obj.names'
    image = cv2.imdecode(img, cv2.IMREAD_COLOR)

    Width = image.shape[1]
    Height = image.shape[0]
    scale = 0.00392


    global classes, COLORS, delay, depth_data
    depthimg = depth_data
    with open(class1, 'r') as f:
        classes = [line.strip() for line in f.readlines()]

    # generate different colors for different classes
    COLORS = np.random.uniform(0, 255, size=(len(classes), 3))
```

```
    # read pre-trained model and config file
    net = cv2.dnn.readNet(weight, config)

    # create input blob
    blob = cv2.dnn.blobFromImage(image, scale, (416, 416), (0,
0, 0), True, crop=False)

    # set input blob for the network
    net.setInput(blob)
    # run inference through the network
    # and gather predictions from output layers
    outs = net.forward(get_output_layers(net))

    # initialization
    class_ids = []
    confidences = []    #Set the camera info, this will only
run once as these variables are not expected to change

    boxes = []
    conf_threshold = 0.5
    nms_threshold = 0.4

    # for each detetion from each output layer
    # get the confidence, class id, bounding box params
    # and ignore weak detections (confidence < 0.5)
    for out in outs:
        for detection in out:
            scores = detection[5:]
            class_id = np.argmax(scores)
            confidence = scores[class_id]
            if confidence > 0.5:
                center_x = int(detection[0] * Width)
                center_y = int(detection[1] * Height)
                w = int(detection[2] * Width)
                h = int(detection[3] * Height)
                x = center_x - w / 2
                y = center_y - h / 2
                class_ids.append(class_id)
                confidences.append(float(confidence))
                boxes.append([x, y, w, h])

    # apply non-max suppression
    indices = cv2.dnn.NMSBoxes(boxes, confidences,
conf_threshold, nms_threshold)

    # go through the detections remaining
    # after nms and draw bounding box
    for k in indices:
        i = k[0]
        box = boxes[i]
        x = box[0]
        y = box[1]
        w = box[2]
        h = box[3]
        rospy.loginfo(str(classes[class_ids[i]]))
        if str(classes[class_ids[i]]) == "handle" and doordone
== True:
            handleHandleObject(image, class_ids[i], confidences
[i], round(x), round(y), round(w), round(h),depthimg)
        if str(classes[class_ids[i]]) == "door" and doordone ==
```

```
False:
            handleDoorObject(image, class_ids[i], confidences[i
], round(x), round(y), round(x + w), round(y + h),depthimg)
            return 0


    # display output image
    #cv2.imshow("object detection", image)

    # wait until any key is pressed
    #cv2.waitKey(delay)

    # save output image to disk
    #cv2.imwrite("object-detection.jpg", image)

    # release resources
    #cv2.destroyAllWindows()


def rawimagehandle(data,method):   #function to transform the
raw_image data from ROS into a .jpg
    bridge = CvBridge()
    raw_image = bridge.imgmsg_to_cv2(data, desired_encoding=
method)

    # Encode undistorted image as JPEG
    _, image_data = cv2.imencode('.jpg', raw_image)

    return image_data

def depth_handle(data):
    global depth_data
    bridge = CvBridge()

    # Convert ROS Image message to OpenCV image
    raw_depth_image = bridge.imgmsg_to_cv2(data,
desired_encoding="passthrough")

    # Convert OpenCV image to NumPy array
    depth_image_array = np.array(raw_depth_image)

    # Convert the depth image to the desired data type (e.g.,
uint16)
    depth_image = depth_image_array.astype(np.float32)  #
Assuming 16-bit unsigned integers

    depth_data = np.fliplr(depth_image)  # Save the depth image
 data for later use and flip it


def imgproc(data):
    global latest_image
    img = rawimagehandle(data,"bgr8")
    latest_image = img
    #yolo(img)

def info_handle(data):
    global K
    K = data.K
    #rospy.loginfo(K)
```

```
def find_catkin_ws(start_path):     #used to find catkin_ws
    for root, dirs, files in os.walk(start_path):
        if 'catkin_ws' in dirs:
            return os.path.join(root, 'catkin_ws')
    return None


def noder():
    global movePub, tf_puber, tf_listener, tf_buffer,
latest_image, doordone
    rospy.init_node('camera_processing', anonymous=True)
    tf_buffer = tf2_ros.Buffer()
    tf_puber = tf2_ros.TransformBroadcaster()
    tf_listener = tf2_ros.TransformListener(tf_buffer)
    rospy.Subscriber("/camera/color/image_raw", Image, imgproc,
 queue_size = 1)
    rospy.Subscriber("/camera/aligned_depth_to_color/image_raw"
, Image, depth_handle, queue_size = 1)
    rospy.Subscriber("/camera/aligned_depth_to_color/
camera_info", CameraInfo, info_handle)
    movePub = rospy.Publisher('/move_base_simple/goal',
PoseStamped, queue_size=10)

    while not rospy.is_shutdown():
        if latest_image is not None:
            yolo(latest_image)
            latest_image = None


if __name__ == '__main__':
    home_path = os.path.expanduser('~')
    catkin_ws_path = find_catkin_ws(home_path)
    if catkin_ws_path is None:
        print("Error: catkin_ws directory not found.")
        sys.exit(1)
    #set the correct directory
    scripts_path = os.path.join(catkin_ws_path, "src/
camera_scripts/src")
    os.chdir(scripts_path)
    noder()
```

Listing 5: doordetect.py the object detection and pose estimation program