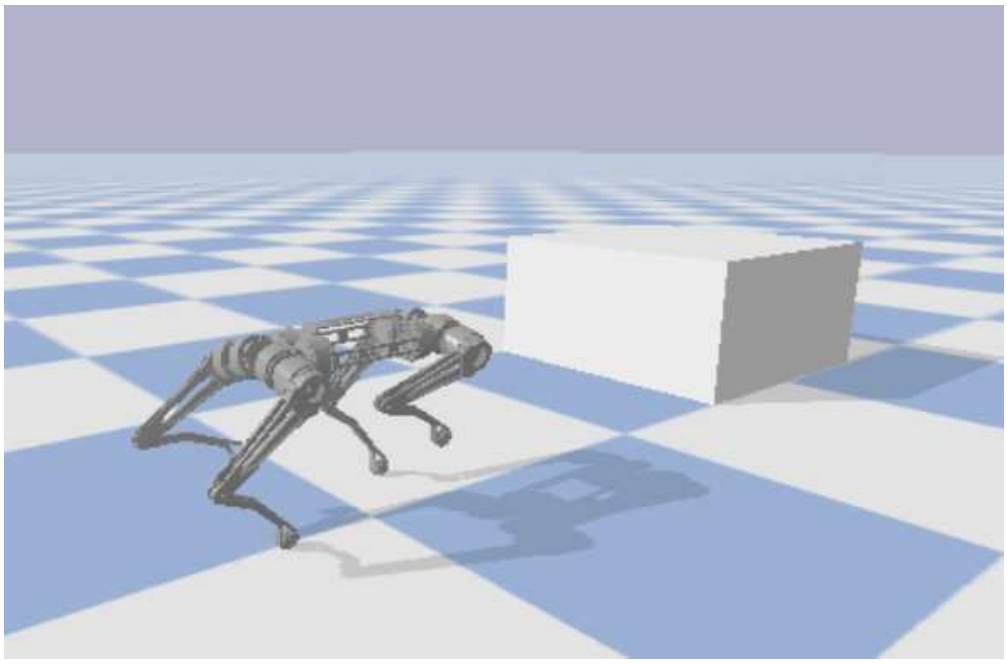
CHALMERS
UNIVERSITY OF TECHNOLOGY



The quadruped robot K3lso

Path planning of a quadruped robot and finding obstacles with depth cameras

Bachelor's degree project report in EENX16
EENX16-23-20

William Haag
August Holm
David Johansson
David Karlsson
Eva Le
Ludvig Tvingby

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023
www.chalmers.se

Bachelor's degree project report 2023

The quadruped robot K3lso

Path planning of a quadruped robot and finding obstacles with depth cameras

William Haag
August Holm
David Johansson
David Karlsson
Eva Le
Ludvig Tvingby

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2023

The quadruped robot K3lso
Path planning of a quadruped robot and finding obstacles with depth cameras
WILLIAM HAAG, AUGUST HOLM, DAVID JOHANSSON, DAVID KARLSSON, EVA LE, LUDVIG TVINGBY

© WILLIAM HAAG, AUGUST HOLM, DAVID JOHANSSON, DAVID KARLSSON, EVA LE, LUDVIG TVINGBY, 2023.

Supervisor: Jonas Fredriksson, Department of Electrical Engineering
Examiner: Karinne Ramirez-Amaro, Department of Electrical Engineering

Bachelor's degree project report 2023
Department of Electrical Engineeering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Cover: Digital twin of the quadruped robot K3lso simulated in PyBullet.

Typeset in LaTeX
Gothenburg, Sweden 2023

The quadruped robot K3lso
Path planning of a quadruped robot and finding obstacles with depth cameras
WILLIAM HAAG, AUGUST HOLM, DAVID JOHANSSON, DAVID KARLSSON, EVA LE, LUDVIG TVINGBY
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Robots have long been sought after to assist humans, and quadruped robots have an advantage in movement over other types. K3lso is a quadruped robot that has been lent to Chalmers University of Technology. This project is a continuation of previous bachelor projects working on K3lso. The main goal of this project is to develop an image processing system for one of K3lso's cameras and a path planning system for a simulated version of K3lso.

The obstacle detection was able to successfully detect objects as long as their heights were not below a decided margin of error. Some problems also arose with shadows cast by the obstacles, creating artifacts. Optimal conditions were found to be a uniformly illuminated room, with matte walls and floors, and no transparent surfaces.

Path planning was done using a potential field by creating a vector field with the goal of exerting an attractive force while the detected obstacles exert a repelling force. This allows K3lso to traverse the field to the goal while being repelled from obstacles. From the start to the goal, this calculates a path that the path follower instructs K3lso to follow. Preliminary tests showed a working path that avoided the registered obstacles, but when integrated with the path follower the simulated camera algorithm detects an obstacle where there should not be one.

Two controllers were constructed for path following that can walk to points in a curved or straight path. After comparing the two different path-following controllers the path controller was faster at longer distances and had higher accuracy than the curve controller.

In conclusion, obstacle detection can detect obstacles that affect the determination of the path, that K3lso then follows.

Keywords: K3lso, robot, quadruped, depth camera, simulation, path planning, obstacle detection, potential field.

Acknowledgements

The basis for this project is the robot dog by the name K3lso which has been charitably donated to Chalmers. The dog was a hobby project by Robin Fröjd, but he has since a few years back lent the robot to Chalmers to be used for educational purposes. We would also like to extend our thanks to our supervisor Jonas Fredriksson and our examiner Karinne Ramirez-Amaro for offering us their full support with indelible feedback and supportive conversations when we needed it the most throughout this project.

William Haag, August Holm, David Johansson, David Karlsson, Eva Le, Ludvig Tvingby, Gothenburg, May 2023

Contents

List of Figures

List of Tables

1
Introduction

Robots have long been sought after to assist humans and perform simple tasks, such as robot arms for assembly or sensors for surveillance. Mobile robots are found useful in environments not suitable for humans, such as remote locations or dangerous environments [5]. One potential improvement of these robots is to move on their own and take in impressions from their surroundings. There are different methods to increase mobility such as wheels, tracks, flying ability, or legs. Robots with legs have a mobility capability that cannot be achieved by the other mentioned solutions, allowing for mobility over unstructured terrains such as volcanoes, ocean floors, or other planets. This makes the quadruped robot particularly useful for use in radioactive areas, disaster areas, mining prospecting, or clearing mines, where mobility is not optimal. These robots also have the ability to handle dangerous materials such as compressed gas or oil, in areas where other robots would have difficulty traversing.

1.1 Background

K3lso is a four-legged robot that was developed by Robin Fröjd as a hobby project and later was lent out to Chalmers University of Technology [2]. It is based on the MIT Cheetah 3 created by Massachusetts Institute of Technology, which is one of today's leading four-legged robots [6]. There are many other notable quadrupedal robots including Spot and BigDog by Boston Dynamics, among others [5],[7],[8].

This project is a continuation of the previous year's two bachelor projects that worked on developing the functionality of the physical and digital robot K3lso [2],[9]. All essential components of K3lso are visualized in Figure 1.1. K3lso is equipped with an image processing system that has five depth-sensing cameras of Intel RealSense models D455 and D435i, and a tracking camera of Intel RealSense model T265. The cameras are currently detached from K3lso.

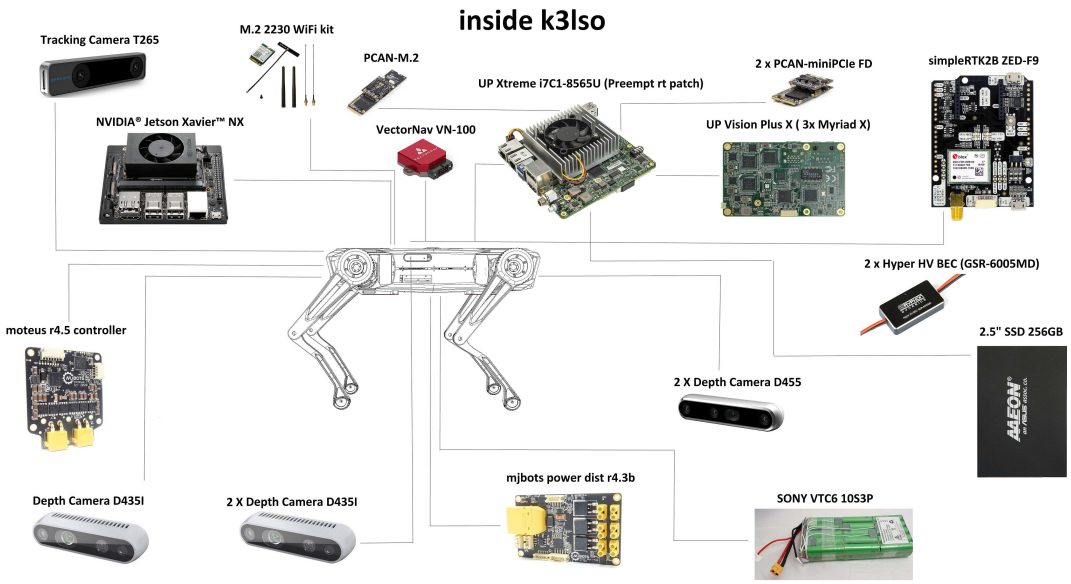


Figure 1.1: Essential components of K3lso [1].

The robot also has a digital twin implemented in several simulation programs. Simulations are useful when testing new code and functions to ensure functionality, and performance, and to not risk destroying robot hardware. There is a digital twin in PyBullet of K3lso from its developer that was further developed by one of the previous year's projects [1],[2]. Additionally, there is a model of K3lso available in Simulink which one of the previous year's groups worked on [10]. For a working robot, ROS2, which is a robot operating system with software libraries, drivers, and development tools can be used for connecting applications to the robot [11]. As there is a good amount of code and information available that needs to be reviewed to further develop the robot, this report will be a mixture of a literature study and an experimental study.

In Figure 1.2, a simple block diagram is presented that illustrates how K3lso's control system is built up. This project will be focusing on developing a solution for the path planner, blue block, and image processing, green blocks. While the orange blocks, the controller, and the physical hardware, K3lso, will not be modified from last year's contributions.

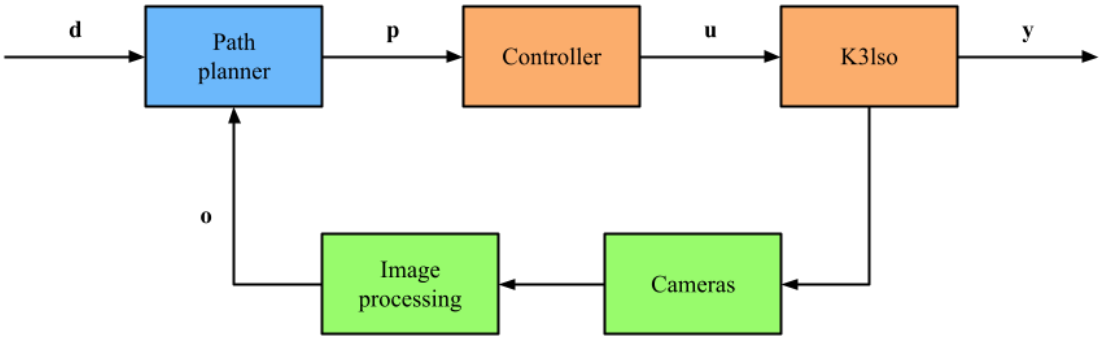


Figure 1.2: Block diagram for control of K3lso.
d = Destination (x,y), o = vector of obstacles (x,y), p = vector of points in path (x,y), u = vector of motor angles, y = K3lso's position (x,y).

1.1.1 Ethical aspects and concerns

With the development of robotics, especially quadruped robots, they can take a person's place in dangerous situations. Whether that be in the workplace or in a more hostile environment such as the rubble after an earthquake or possible areas with a limited or non-existent oxygen supply that would not be suitable for a human. In the workplace, the robot can supplement the worker in transporting heavier objects in a factory. In some cases, a robot could simply perform the work instead of a human replacing them for the sake of efficiency. Some people may lose their jobs, which could lead to a higher unemployment rate. However, in a report by Leslie Willcocks [12] he argues that with an aging population and a skill shortage in G20 countries, automation will keep productivity up instead of widening the production gap as he claims might otherwise happen. He concludes with a finishing thought that the way automation might become a problem is when the population is unable to adapt and learn new skills as we transition into an age of ever-increased automation.

As much of available cutting-edge technology, robotics has found its way to the military as it looks to capitalize on emerging advanced scientific fields. Some might see this development as concerning as it automates and desensitizes lethal warfare. But as Elinor Sloan [13] argues, it could take away the intense human emotions that may lead to war crimes and severe panic from the heat of battle. But using robots and specifically AI complicates the situation e.g. when something goes wrong and there is a need to find the person responsible for the action. If the robot acted autonomously, then it acted by itself and thereby not making it feasible to place the blame on the humans creating and giving the robots their orders.

1.2 Purpose and goal

The purpose of the project is to gain a deeper understanding of the technology behind the movements of quadruped robots and how the camera technology helps robots to navigate. The goal is to develop the basis for a camera system to help the robot K3lso navigate to avoid obstacles and walk according to predetermined instructions in a simulated environment.

1.2.1 Research questions

To achieve the purpose regarding the camera system, navigation, and simulation, two broad research questions were asked, followed by a few sub-questions.

1. How can K3lso determine where it can walk?
   (a) How do the cameras work?
   (b) How can the cameras be used to detect obstacles?
   (c) How can the camera's image be used to assess which directions K3lso can move in?
2. How can K3lso be simulated to walk and follow a path?
   (a) How can simulation be used to control a digital twin of K3lso?
   (b) How can the simulated model walk to points?
   (c) How can path planning be used to navigate around obstacles?

1.3 Limitations and scope

Quadruped robotics is an active research area with significant development over recent years, therefore a realistic overview of what is possible to achieve within the project's time frame is needed. It will therefore be necessary to curb the project's ambitions.

The robot is currently unable to move due to malfunctioning motors [2],[9]. It is speculated that the control cards are damaged. A troubleshooting process on the hardware of the robot was carried out by running tests in hopes of identifying the true reason for the error, see Appendix A. The conclusion from the tests was that an attempt to repair the hardware would be too time-consuming with no guarantee of success. However, the physical cameras are still functional and will be used for image processing development. The first priority for image processing will be to focus on one of the two front cameras since they provide the greatest functionality for navigation. The two side cameras and two rear cameras will not be integrated into the project. To ensure a stable testing environment, tests of the cameras will be conducted separately from the robot on a stand in a well-lit room.

For the software, the focus will be to further develop K3lso's functionality in simulated environments, specifically in PyBullet, with the help of the previous project's conclusions and code. The developed code will be constructed for well-lit rooms with flat surfaces such as indoor environments. Uneven surfaces and active balance regulation would complicate the task disproportionately and will therefore not be considered.

2

Theoretical background

To realize a path planner with the help of image processing from cameras, some underlying theoretical background is needed. The mathematical model for K3lso's controller is presented and how its mechanics can be used in an algorithm for path planning with potential fields. PyBullet's functions and K3lso's existing environments and control panel are explained, as well as the cameras available in the project.

2.1 Modelling

Mathematical models are often used in robotics as a way to realize a prediction of a real-world scenario and to get insight [14]. In a mathematical model, kinematics and inverse kinematics are used to describe the relations between objects, in this case, the links of the robot. Such a model is based on trigonometry. Figure 2.1 describes K3lso's torso and one of its legs, which is what is being considered in the mathematical model.

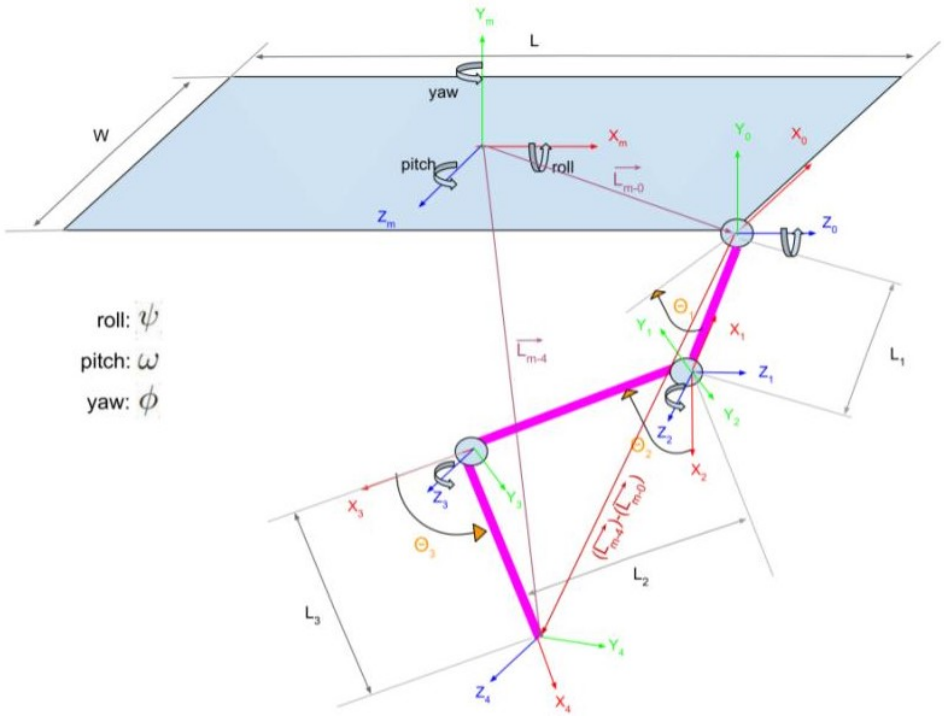


Figure 2.1: Torso and one leg of K3lso described by mathematical modelling [2].

2.1.1 Kinematics

As seen in Figure 2.1, a leg is constructed by three links. Each link is connected by a joint, and the lower link's end, or the "foot", is what is called an end-effector [15]. The end-effector is designated to interact with the environment. Every joint's position depends on the links' angles. If the first link is rotated all the joints leading up to the end-effector changes pose, including the end-effector itself. The new poses can be calculated using the following rotational matrices:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(w) & -sin(w) & 0 \\ 0 & sin(w) & cos(w) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

$$R_y = \begin{bmatrix} cos(\phi) & 0 & sin(\phi) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\phi) & 0 & cos(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

$$R_z = \begin{bmatrix} cos(\psi) & -sin(\psi) & 0 & 0 \\ sin(\psi) & cos(\psi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

where w is the pitch angle, ϕ is the yaw angle and ψ is the roll angle.

With the rotational matrices, every point can be computed for the end-effector affected when moving the first joint. To reach every point in the system, transformation matrices are needed, which can combine rotation and movement.

$$T_m = \begin{bmatrix} 1 & 0 & 0 & X_m \\ 1 & 0 & 0 & Y_m \\ 0 & 0 & 1 & Z_m \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

Matrix T_m describes the translation in the center of mass with the body's center of mass (X_m, Y_m, Z_m) as the reference point.

With these matrices introduced the foot's, or end-effector's position can be calculated. This is done by first computing the leg's starting position (X_0, Y_0, Z_0) with reference to the center of mass as the origin: $(X_m, Y_m, Z_m) = 0$. To do this, translation matrix T_m^0 is used:

$$T_m^0 = R_x \cdot R_y \cdot R_z \cdot \begin{bmatrix} 1 & 0 & 0 & X_m \\ 1 & 0 & 0 & Y_m \\ 0 & 0 & 1 & Z_m \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos(\frac{\pi}{2}) & 0 & \sin(\frac{\pi}{2}) & \frac{L}{2} \\ 0 & 1 & 0 & 0 \\ -\sin(\frac{\pi}{2}) & 0 & \cos(\frac{\pi}{2}) & \frac{W}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.5}$$

Given the starting position (X_0, Y_0, Z_0) as the origin, the end-effector's position can be calculated. This is done by computing the transformations representing the leg's single movement from the previous position. Four different transformation matrices are constructed for every link's endpoint.

Translation matrix T_0^1 for (X_1, Y_1, Z_1) with reference to (X_0, Y_0, Z_0):

$$T_0^1 = \begin{bmatrix} \cos(\theta_1) & -\sin(\theta_1) & 0 & -L_1 \cdot \cos(\theta_1) \\ \sin(\theta_1) & -\cos(\theta_1) & 0 & -L_1 \cdot \sin(\theta_1) \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.6}$$

Translation matrix T_1^2 for (X_2, Y_2, Z_2) with reference to (X_1, Y_1, Z_1):

$$T_1^2 = \begin{bmatrix} 0 & 0 & -1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.7}$$

Translation matrix T_2^3 for (X_3, Y_3, Z_3) with reference to (X_2, Y_2, Z_2):

$$T_2^3 = \begin{bmatrix} \cos(\theta_2) & -\sin(\theta_2) & 0 & L_2 \cdot \cos(\theta_2) \\ \sin(\theta_2) & \cos(\theta_2) & 0 & L_2 \cdot \sin(\theta_2) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.8}$$

Translation matrix T_3^4 for (X_4, Y_4, Z_4) with reference to (X_3, Y_3, Z_3):

$$T_3^4 = \begin{bmatrix} \cos(\theta_3) & -\sin(\theta_3) & 0 & L_3 \cdot \cos(\theta_3) \\ \sin(\theta_3) & \cos(\theta_3) & 0 & L_3 \cdot \sin(\theta_3) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.9}$$

Translation matrix T_0^4 for (X_4, Y_4, Z_4) with reference to (X_0, Y_0, Z_0) is then given by multiplying the four matrices:

$$T_0^4 = T_0^1 \cdot T_1^2 \cdot T_2^3 \cdot T_3^4 \tag{2.10}$$

Lastly, translation matrix T_m^4 for (X_4, Y_4, Z_4) with reference to (X_m, Y_m, Z_m) is given by multiplying matrix T_0^4 with T_m^0:

$$T_m^4 = T_0^4 \cdot T_m^0 \tag{2.11}$$

To calculate every end point's new position's coordinate, its transformation matrix is multiplied by the reference point.

2.1.2 Inverse Kinematics

Inverse kinematics is used to compute the angles of the links given the joints' positions. The interplay between kinematics and inverse kinematics is displayed in Figure 2.2.

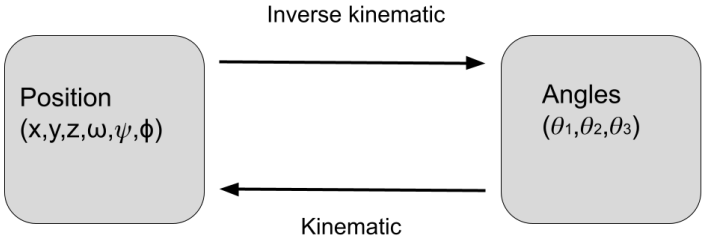


Figure 2.2: The interplay between kinematics and inverse kinematics

This is especially useful for the pose control of K3lso. The angles are computed for a wanted position, which is thereafter fed to the robot that assumes the new position. In inverse kinematics, the point (X_0, Y_0, Z_0) is set to the origin, and the rest of the coordinates are computed in reference to this. The angles θ_1, θ_2 och θ_3 are then calculated as:

$$\theta_1 = \arctan\frac{-Y_4}{X_4} - \arctan\frac{\sqrt{X_4^2 + Y_4^2 - L_1^2}}{-L_1} \tag{2.12}$$

$$\theta_2 = \arctan\frac{Z_4}{\sqrt{X_4^2 + Y_4^2 - L_1^2}} - \arctan\frac{L_3 \cdot \sin(\theta_3)}{L_2 + L_3 \cdot \cos(\theta_3)} \tag{2.13}$$

$$\theta_3 = \arctan\frac{-\sqrt{-1 - D^2}}{D} \tag{2.14}$$

where

$$D = \frac{X_4^2 + Y_4^2 + Z_4^2 - L_1^2 - L_2^2 - L_3^2}{2 \cdot L_2 \cdot L_3} \tag{2.15}$$

2.2 Cameras

K3lso uses multiple cameras to observe its surroundings. There are two larger depth cameras on the front part of K3lso, two smaller depth cameras, one on each side of the robot, one depth camera, and one tracking camera on the back. All cameras are made by Intel and thus use the Intel RealSense SDK. The front depth cameras are of model D455 and the side and back are of model D435i. The main difference between these models is their range [16]. The D455 works optimally between 60 centimeters and 6 meters while the D435i works optimally between 30 centimeters and 3 meters. They also have an inertial measurement unit, IMU.

2.2.1 RealSense SDK

Intel RealSense is the software used to interact with the depth cameras [17]. It works with multiple languages and frameworks, Python, Matlab and ROS 2 being the most relevant. With the SDK the data from the different sensors can be accessed. Standard RGB pictures can be created with the values from the RGB sensor, but the main focus is the data generated by the depth sensors. The camera delivers frames containing a matrix with the measured depth data. Each data point in the given matrix represents a pixel and the corresponding distance an object in that pixel is from the camera. The 4x4 matrix in 2.16 demonstrates how the depth data might look. The cameras give the distance in millimeters, so the matrix shows a flat surface 1 meter away, with a square in the middle that is just 50 centimeters away. This could for example be a image of a wall with a cabinet on it.

$$\begin{bmatrix} 100 & 100 & 100 & 100 \\ 100 & 50 & 50 & 100 \\ 100 & 50 & 50 & 100 \\ 100 & 100 & 100 & 100 \end{bmatrix} \tag{2.16}$$

2.3 Path planning with potential field

Path planning with a potential field is a simple idea - to have K3lso "attracted" to the goal and "repelled" by the obstacles [18]. The robot acts as if it is a positively charged particle moving towards a negatively charged goal. A simple potential field with one object and one goal is presented in figure 2.3.

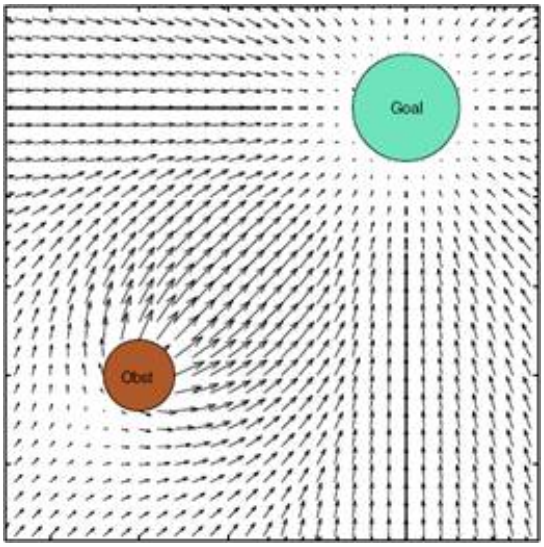


Figure 2.3: Potential field with one object and one goal [3].

In the potential field, the attractive potential generated by the goal can be computed as:

$$P_{x,y}^{goal} = K_P \cdot L \cdot \sqrt{|x - x_{goal}|^2 + |y - y_{goal}|^2} \tag{2.17}$$

where (x_{goal}, y_{goal}) is the goal position, K_P is the attractive potential gain and L is the distance in meters between points in the grid.

The force of repulsion for the obstacle is computed by first calculating the distance $D(x, y)$ to the obstacle with coordinates (x_{object}, y_{object}).

$$D_{x,y} = \sqrt{|x - x_{object}|^2 + |y - y_{object}|^2} \tag{2.18}$$

If the distance D is greater than the robots radius for detection rr, the algorithm chooses to ignore the object. If the distance is less, the repulsive potential can be computed as:

$$P_{x,y}^{object} = ETA \cdot L \cdot (\frac{1}{D(x,y)} - \frac{1}{rr})^2 \tag{2.19}$$

Here, ETA is the repulsive potential gain and L is the length between points in the grid.

The total potential for every point can thereafter be computed.

$$P_{x,y} = P_{x,y}^{goal} + P_{x,y}^{object} \tag{2.20}$$

And every point's affecting force becomes:

$$F_{x,y} = -\nabla P_{x,y} \tag{2.21}$$

The equations above were constructed with the condition that there is only one object present in the potential field. If there were more than one, every point's closest object have to be derived and calculations done thereafter.

To then compute the force affecting K3lso, the robot's coordinates are calculated with the kinematics equations (2.6) **-** (2.11) and then placed in the force field.

2.4 PyBullet simulation

The developed simulation environment in PyBullet is an open space to test the robot's movements, see Figure 2.4. The available tools in the environment consist of a fixed virtual camera generating three different 240 x 320 images, RGB, depth, and a segmentation mask image. There is also a control panel with different robot control options such as `MPC`, `Pose`, and `Bezier` controller.

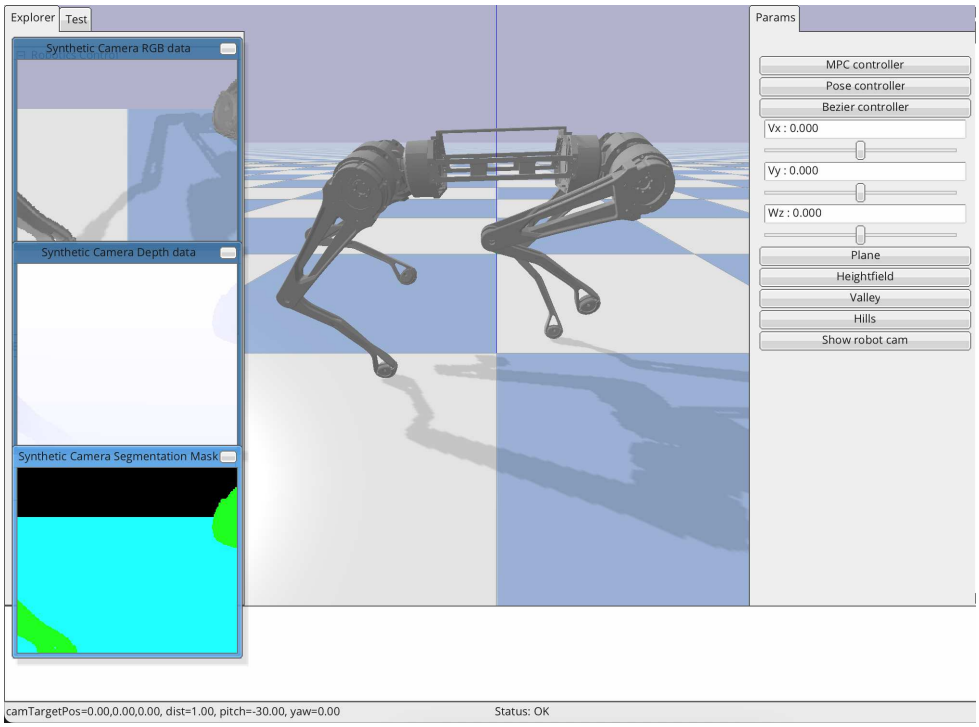


Figure 2.4: PyBullet simulation starting point.

These control options allow the robot K3lso to move in different ways. MPC, or model-based predictive control, is a type of controller used in various fields all over. As the name suggests, the controller tries to optimize it's output by creating a mathematical model of the process outcomes to try and predict the controller's future output [4]. This can be seen in Figure 2.5. The controller allows for different inputs to translate into translational movements to allow for K3lso to move around.

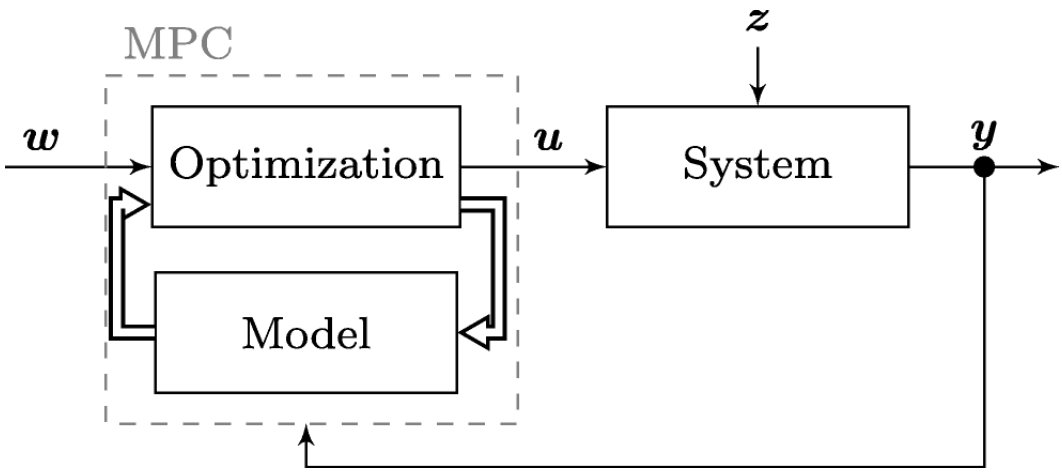


Figure 2.5: Block diagram showcasing an essential version of the MPC [4].

The pose controller allows K3lso to move its upper body around while having all four feet planted in the ground, allowing it to contort its body and assume poses. The robot can rotate in three dimensions, roll is rotation around the x-axis, pitch is rotation around the y-axis and yaw is rotation around the y-axis. The last controller was supposed to create a smooth gait trajectory for K3lso's leg and feet [19]. But the Bezier controller does not currently work and no obvious solution could be found, therefore it has no use to the project. It is also possible to change the ground surface to a plane, highfield, valley, and hill. The environment also has a developed path planner based on potential fields and path-following algorithms, but it is not integrated with the simulation and the camera.

3
Method

The project started with an underlying literature study to gain a base knowledge of the subject matter. After the initial literature study, experimental studies started in parallel. The first step is image processing using physical and virtual cameras to generate necessary variables. The second step is to send the data to the path planning algorithm, that is using potential fields, shown in the block diagram 1.2 earlier. During development, additional literature studies were done as the need arose.

3.1 Image processing

The image processing part seen in Figure 1.2 has been divided into two parts, the first part is about obstacle detection which is needed for the second step, obstacle position estimation.

3.1.1 Obstacle detection

Any obstacle on a flat surface will be closer to the camera on K3lso than the floor behind the obstacle would be otherwise. Thus, a method has been developed to detect obstacles based on this idea. A reference vector was created with the distance to the floor at all the rows in the depth matrix. Anything measured closer than what the floor should be, outside some added margin of error, was deemed to be an obstacle. A simple proof of concept was developed in MATLAB to test whether the concept works. When the calibration is done, meaning when the reference vector is created, it is important that it is done in an empty space. Since the reference vector should be the distance to the floor, there cannot be anything on the floor, otherwise, anything on the floor will be included in the calibration and thus not be detected as an obstacle when the program is run. To avoid the need for an absolutely empty room, the calibration is done only using one or a few columns in the center of the image. This means that the floor only needs to be obstacle-free right in front of the camera.

To more easily communicate with the simulation of K3lso, the image processes were coded in Python instead of MATLAB [20]. MATLAB has its workspace functionality which was used to save the reference vector between tests of obstacle detection. In Python the vector was instead saved as a text file which will allow it to be kept when the computer is shut off, reducing the need for calibration. This is preferable since clearing the floor might not always be possible.

To test the camera's functionality, measurements were made to assess optimal and sub-optimal conditions. After noticing that matte and opaque surfaces along with uniformly illuminated rooms gave clear results, those were determined to be optimal conditions for detecting obstacles. Tests were then made to instead show how the depth camera deals with transparent and reflective surfaces. The camera was placed in front of a window at the same distance that an object should be detected. Furthermore, the camera was placed on a table with direct sunlight reflecting off of the surface. Several measurements were made for both cases until conclusions could be made. Later, in optimal conditions, multiple objects of differing sizes were placed on the floor in front of the camera to test their ability to detect objects of different sizes.

The simulation model of K3lso in PyBullet also includes a simulated camera. To test the obstacle detection together with the simulated path planning, the obstacle detection program was reworked to be able to use the output of the simulated camera.

3.1.2 Obstacle position estimation

Once an obstacle has been detected using the camera's generated depth images, the next step was to calculate the world coordinate of the obstacle. To easier test the performance of the obstacle position estimation, the code was implemented in PyBullet for simulation. In PyBullet, there are built-in virtual cameras generating three different images seen in 2.4, RGB, depth, and segmentation mask images. The depth image in the simulation is similar to the depth image from the physical camera except for the image size which was easy to change. The segmentation mask image in the simulation is also similar to the obstacle detection code presented in section 3.1 and is interchangeable in case of obstacle detection.

The segmentation mask image in PyBullet generates values between -1, 0, and 1 for the sky, floor respective objects. Since obstacle pixels are the needed information, only indices for pixels with value 1 were stored in a list and processed using depth information in the next step. To estimate the obstacle's position in world coordinates, the depth image was used with a combination of some trigonometry, K3lso's yaw rotation, camera position, correction constants, and with consideration of the physical camera's view, range from 0.6 units to 6 units. The depth image generated values from 0-1, where 0 was the closest position to the camera and 1 was the furthest. In the calculation, the depth image used was the one with the smallest value within the pixels where the obstacle was found.

The algorithm did not take the object's position in the horizontal line into account, which meant that any object detected was estimated to be positioned directly in front of K3lso. Therefore, some thresholds were set to check whether the center of the object was at the pixel columns in the middle of the image. Anything outside the threshold needed to be checked if the object was more to the left or right side of the image. To test the accuracy of the estimation of the object position in world coordinates, the object has been placed in different x- and y-coordinates as illustrated in Figure 3.1. The different colors of the boxes indicate different y-values, but the same x-values.

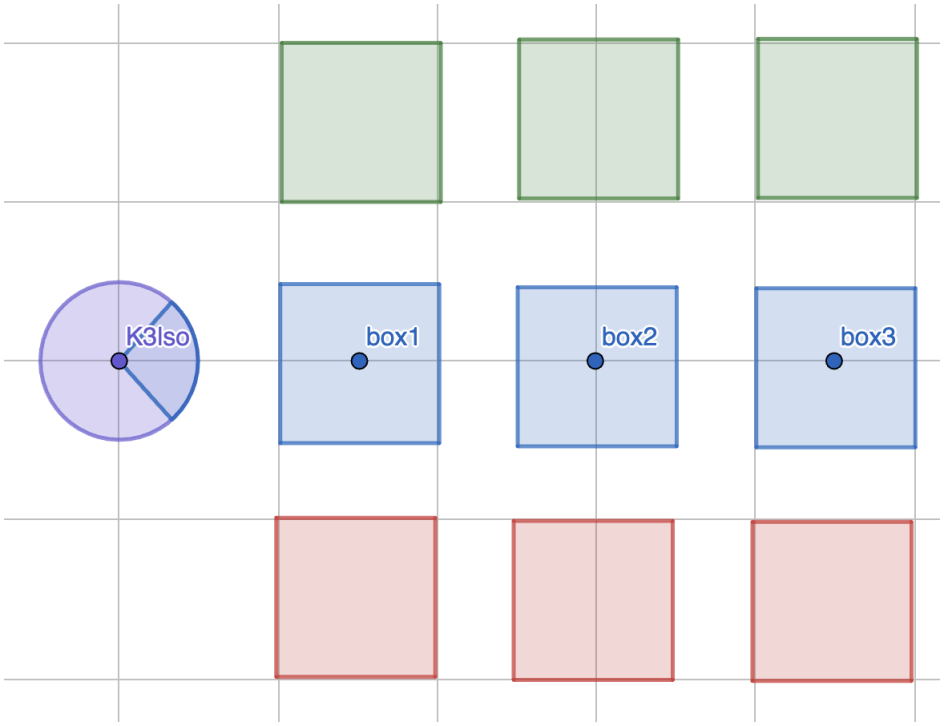


Figure 3.1: Testing the accuracy of estimating the object position to world coordinates by placing objects in different positions.

3.2 Path planning using potential fields

Here, the path planning algorithm was used and tested in PyBullet, given that the image processing part is completed, see 1.2. First, the simulation of an object in the environment and attaching the camera to K3lso were needed. Then, to successfully integrate the simulation with the existing path planning algorithm using potential fields, 2.17 - 2.21, some required input variables needed to be determined including the target position, obstacle position, and K3lso's position. The target position could be set to a desired position and K3lso's position was easily obtained with kinematic equations, 2.6 - 2.11, but the obstacle position needed to be calculated using the provided virtual camera.

Figure 3.2 shows an example of how K3lso would navigate through the potential field presented in Figure 2.3. The green cross marks the starting point and the red arrow represents the path it would take.

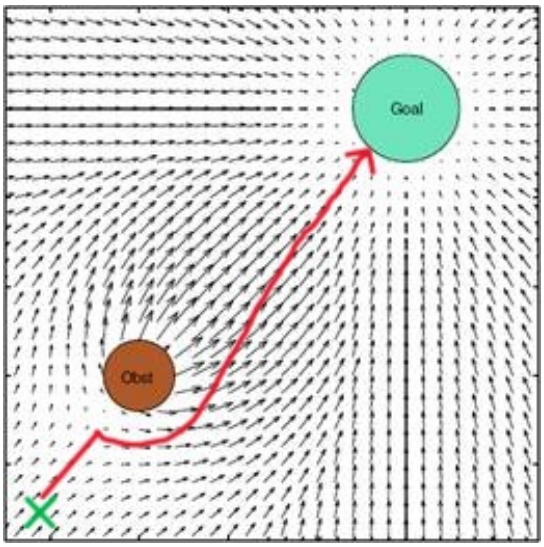


Figure 3.2: Path planning using potential field [3].

In this implementation of path planning, K3lso is asked to navigate a vector field that consists of the entirety of the simulated world. The final destination given emits an attractive force pulling K3lso towards it. By using the implemented simulated camera and extracting pictures of the environment segmented into different categories thereby identifying obstacles. The path planning algorithm used the potential field algorithm to plot the path using points separated by a distance of 0.5 meters.

3.3 Path following controller

Using PyBullet's methods and pre-existing code, a controller to follow a path has been developed. Given an obstacle-free path, the robot follows it and gets to the determined destination. K3lso's MPC controller was used for setting different speeds for the robot to develop the path and curve controller. New functions were created to see if the robot arrived or if it was angled directly at the next destination. The function that checks if the robot arrived checks if the robot's position is equal to the destination with a small margin. Similarly, the function to see if the robot is angled towards the next destination checks the robot's orientation within a margin.

Using these two methods, the robot could then be rotated in the correct orientation and walk to the destination. This path controller could then take in two points with x- and y-coordinates and the speeds between the points. Walking the path was tested in two ways, standing in one spot and rotating to the direction of the next point and then walking straight with the path controller. The curved controller instead walks in an arc between the points, in this case, K3lso rotates as it walks.

4
Results

This chapter will present the results of different tests and what has been developed in the same order as what was described in the method chapter. First, how the image processing for the physical and virtual camera worked, then how the path planning and path following for the simulations went.

4.1 Image processing

Focus has been on one front-facing depth camera, with the goal of K3lso avoiding obstacles. To achieve this, the depth data generated by the camera is used. The RGB sensor can be paired with an AI model for object detection, but this was deemed outside of the scope since the focus is to avoid obstacles and not to classify them.

4.1.1 Obstacle detection

Figure 4.1 shows the result of a proof of concept developed in MATLAB. This program focuses only on the floor and ignores what is above it. In other words, to avoid the problems that arise with walls and such when calibrating the floor, only the lower part of the image was used. Figure 4.1 has a jet ski in the background so to not make this part of the calibration, data was only collected up to the green part of the gradient. This is why the detected box shown in white is smaller than the real box. Not collecting data for the upper part of the image means that no obstacles can be detected there. This will generally not be a problem since most obstacles will be standing on the floor and thus will be on the lower part of the image when they get closer. For the cases when they are not on the floor, the part of the calibration vector will get the value of the floor where the cut-off is, see Figure 4.2.

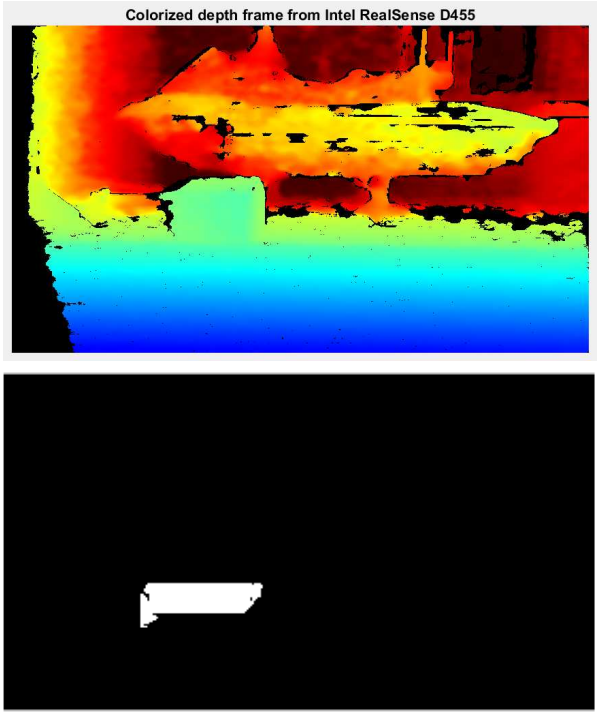


Figure 4.1: Box placed on floor detected as an obstacle.

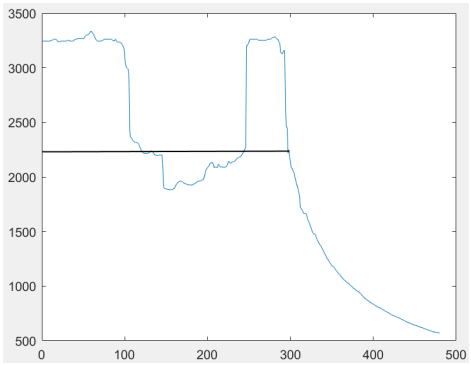


Figure 4.2: Calibration vector with jet ski and what it would look like if ignored it.

The graphs in Figures 4.2, 4.8, and 4.9 show the distance to objects on the y-axis in millimeters, while the x-axis represents the vertical rows of pixels received from the camera. The pixels start from the top of the vertical viewing range at row 0 and continue downward to row 480. The rightmost values from the graph are therefore the lowest vertical row of pixels measured with the camera. Using Figure 4.2 as a reference, the graphs shown in figures 4.9 and 4.8 ignore the upper half of the received picture, only showing received data from rows $240 - 480$.

Since there is no need to run the image processing continuously the code was developed to fetch a frame from the cameras whenever a detection should be made. This revealed that the quality of the depth measurements where inferior for a single frame compared to a continuous collection of frames. To solve this, since only one frame is needed and continuously collecting frames is unnecessary, a few frames are collected right before the frame that will be processed is collected, this resulted in

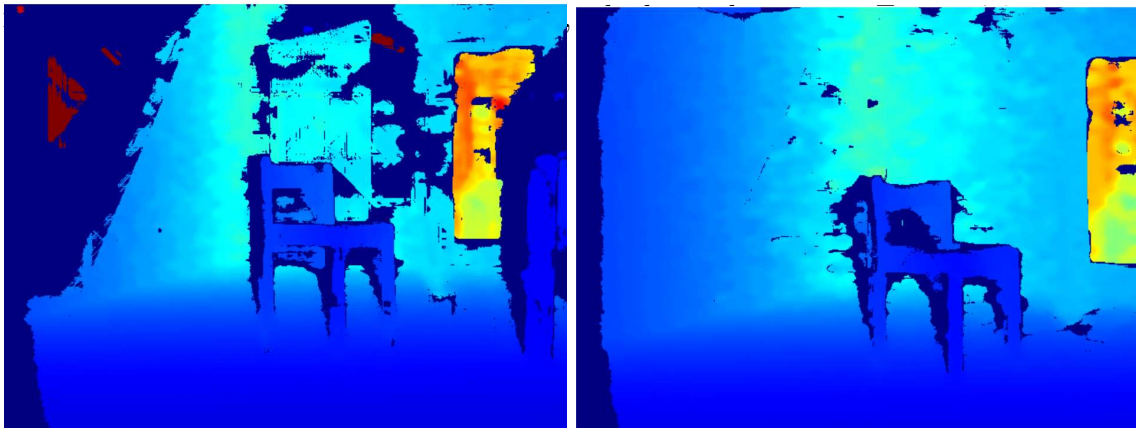


Figure 4.3: Left - Single frame collected. Right - Single frame collected after the collection of a few frames.

The object detection was run with a few different objects in the frame to see which ones would be detected and how well. The test was done in a room with a matte floor and matte walls. The room also had uniform lighting, apart from some light coming from windows to the right, creating some slight shadows. Figure 4.4 shows how the camera was set up along with the generated depth frame and reference vector. The figure also includes a two-colored detection frame which was generated with the reference vector immediately after calibration. This demonstrates how the reference is only made with the middle part of the image since the jacket on the wall, the person and the chair, and the side wall have all been detected as obstacles.

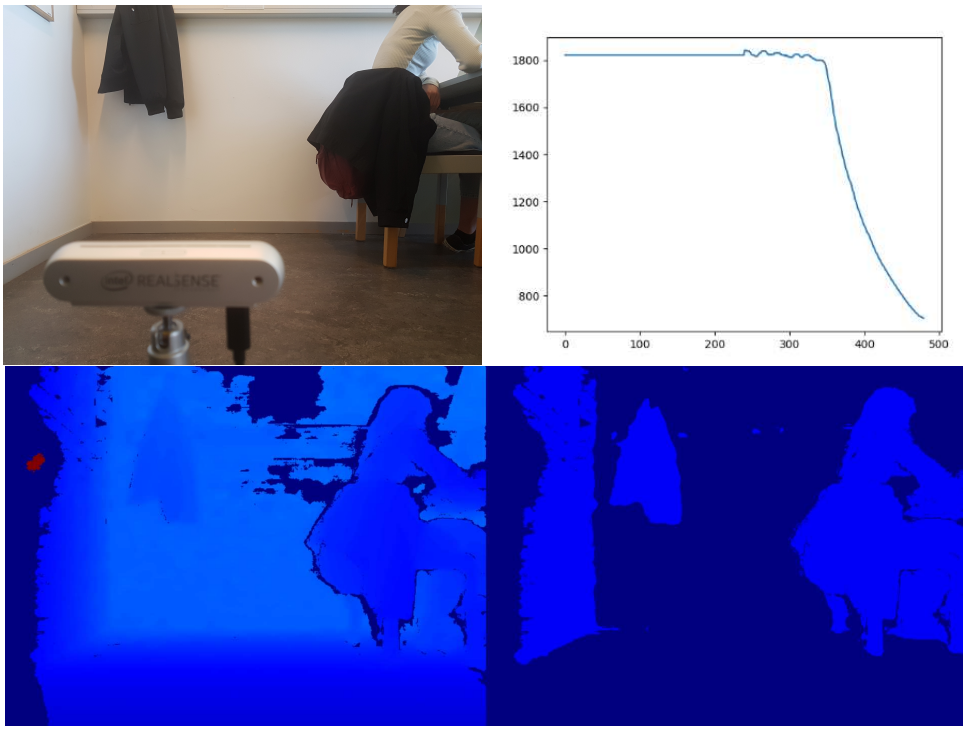


Figure 4.4: Top left - Picture of camera setup. Top right - Reference vector. Bottom left - Depth frame. Bottom right - Detection frame

Figure 4.5 shows the results of three different tests, an empty backpack laying on the floor, a chair, and a water bottle. All three obstacles have been detected. It can be noted some clear artifacts can be seen on the leftmost leg of the chair and the bottom left of the water bottle. These artifacts may be present on the backpack as well, but the less distinct shape makes it hard to determine.

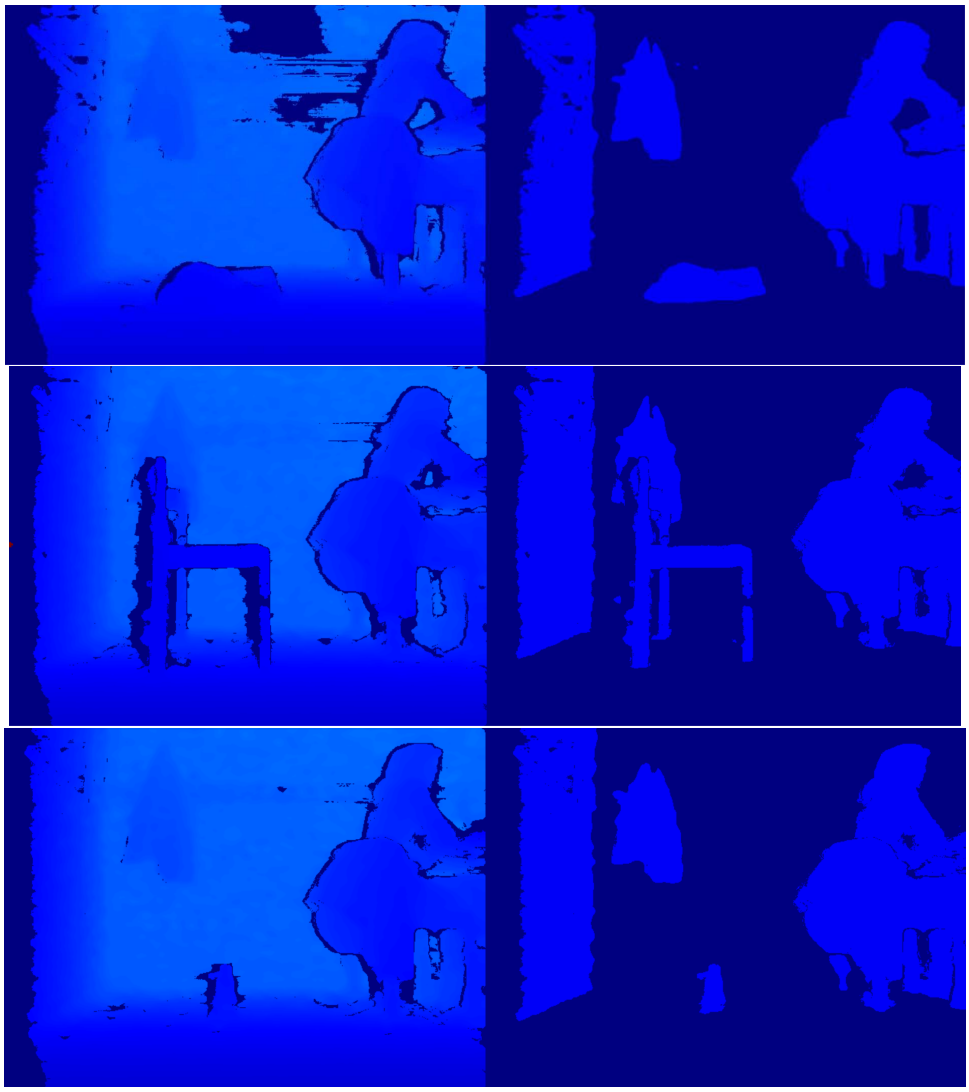


Figure 4.5: All images are pairs of a depth frame and a detection frame. Top - Backpack laying on the floor. Middle - Chair standing on the floor. Bottom - Water bottle standing on the floor.

Lastly, the detection was tested on a whiteboard eraser, the results of this can be seen in figure 4.6. The program failed to detect the eraser. The reason for this is the margin that is applied on top of the reference vector to ensure that the floor itself is not detected as an obstacle. The eraser is not thick enough to extend above the margin and is thus not detected. The objects in figure 4.5 are also affected by the margin, but since they all extend way above it, it is harder to notice.

Figure 4.6: Top - Depth frame to the left and detection frame to the right. Bottom - Image showing an eraser on the floor in front of the camera

The obstacle detection was also tested to work with the data from the simulated cameras, which can be seen in the upper two pictures of Figure 4.7. The two lower pictures show PyBullet's built-in depth data and how PyBullet segments the image into background, floor, and an object. In this case, the developed object detection identifies the cube just as well as the built-in segmentation.

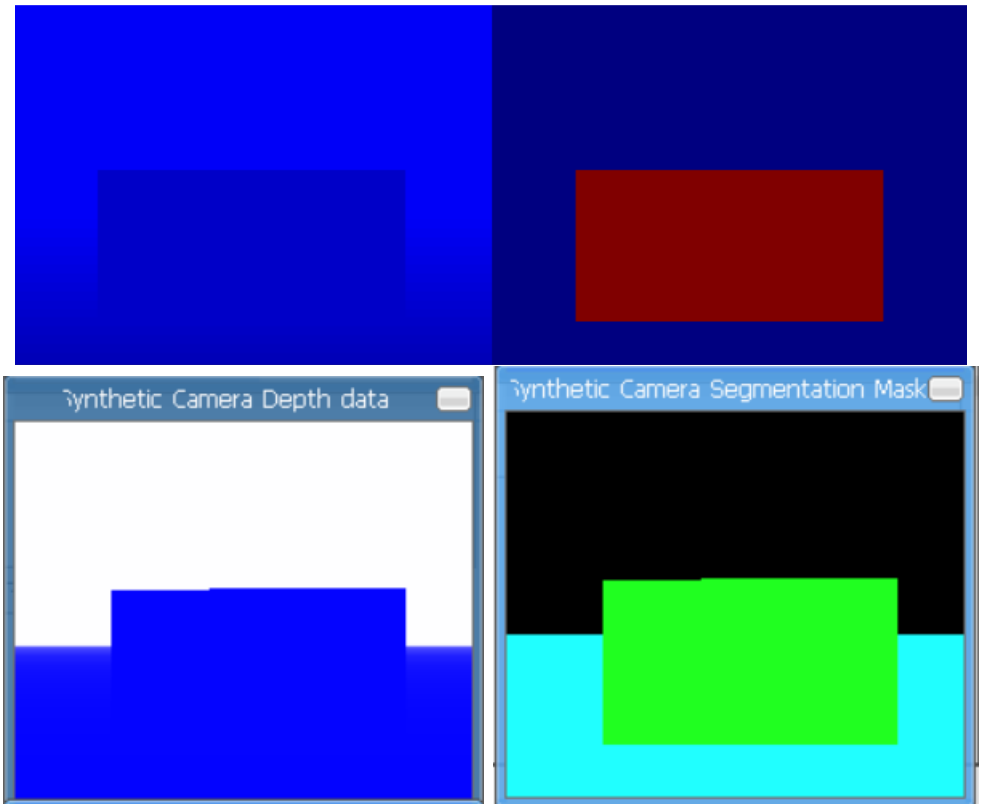


Figure 4.7: Top left - Depth frame from the simulated camera in Python. Top right - Obstacle detected by Python. Bottom left - Depth frame created by PyBullet. Bottom right - Segmented image generated by PyBullet.

4.1.2 Camera functionality

The depth camera used has difficulties detecting objects with certain reflective properties, especially when angled to not directly face the camera. This was particularly noticed when the robot was to be moved across a reflective surface, creating gaps in the received data. While distances shorter than expected are interpreted as objects, longer distances are instead interpreted as holes in the surface, e.g. the robot facing the edge of a staircase. If holes are to be accounted for in the detection algorithm, reflective surfaces can cause large problems for path planning, but if not accounted for, the robot will instead be fully free to fall down from the edges. Similarly, the camera has difficulty recognizing transparent surfaces like windows, which can result in the robot not understanding that it faces such an obstacle.

Figure 4.8 shows the camera directly facing a window at a distance of 0.6 meters. The received data instead shows that the only objects to be found are more than 60 meters away, while other data points in the camera's y-axis are set at a distance of 0 meters, which means that no data is recorded at all for those pixels rows.

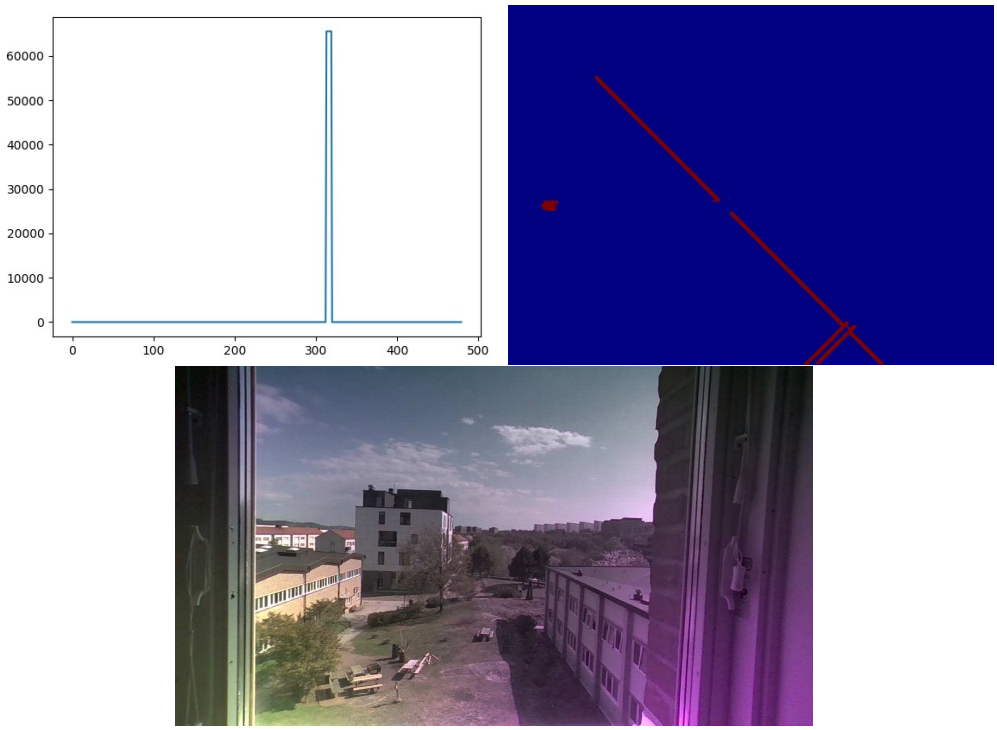


Figure 4.8: Camera facing a window.

Figure 4.9 shows what the camera observes when facing a Table at a certain angle. The upper left graph shows an object at a distance of 13 meters at pixel 280 and a closer object at pixel 420. As stated earlier, the graph shows vertical rows of pixels, ignoring the first 240 rows. The upper right picture shows the color map of the received data. The dark blue color that makes up the majority of the lower half of the picture means that no data was received. The lower picture shows a standard RGB interpretation of what the camera observes.

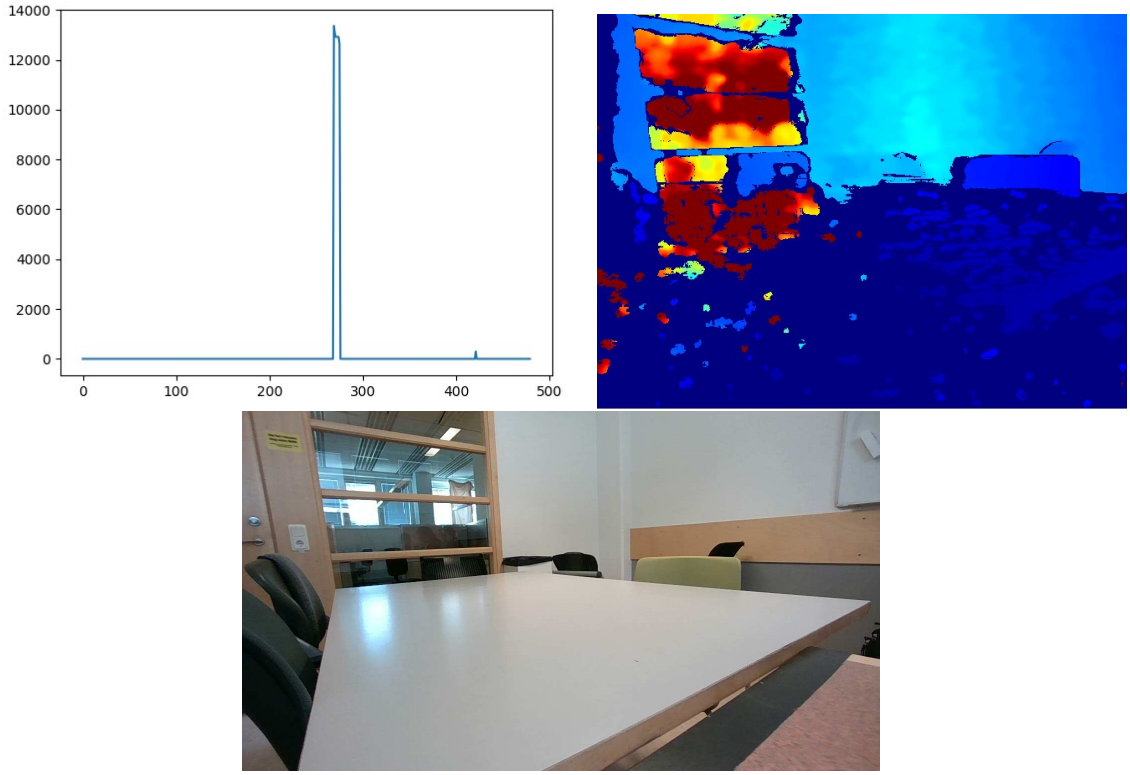


Figure 4.9: Camera angled at a table.

4.1.3 Obstacle position estimation

The results of the test for estimating x- and y-positions from the virtual camera to get world position for varied box positions can be seen in Table 4.1. Table 4.2 shows the supposedly true closest box position relative to K3lso. When comparing the tables, it is seen that the positions [3, -1], [3, 0], and [3, 1] are placed in about the center of the virtual image and have the highest accuracy for object estimation. Estimations with the lowest accuracy are objects placed far to the edges of the frame, too far from the camera, or even too close to the camera where it was not detectable.

Table 4.1: Estimation of the closest detected box position relative to K3lso in the world coordinates. The first row and first column are the y-value respective x-value of the box's center point.

box[x, y]	-2	-1	0	1	2
1	[- , -]	[1.1, -0.2]	[- , -]	[1.1, 0.2]	[- , -]
2	[- , -]	[1.9, -0.4]	[1.9, 0.0]	[1.9, 0.4]	[- , -]
3	[2.7, -0.6]	[2.6, -0.6]	[2.7, 0.0]	[2.6, 0.6]	[2.7, 0.6]
4	[2.9, -0.7]	[2.9, -0.7]	[2.9, 0.0]	[2.8, 0.6]	[2.7, 0.7]
5	[3.0, -0.7]	[3.0, -0.7]	[3.1, 0.0]	[3.0, 0.7]	[3.0, 0.7]

It is seen in Table 4.1 that box positions [1, -2], [2, -2], [1, 0], [1, 2], and [2, 2] were unable to be detected and no estimations were made. This can also be seen in

Table 4.2: The true closest box position relative to K3lso in the world coordinates. The first row and first column are the y-value respective x-value of the box's center point.

box[x, y]	-2	-1	0	1	2
1	[0.5, -1.5]	[0.5, -0.5]	[0.5, 0]	[0.5, 0.5]	[0.5, 1.5]
2	[1.5, -1.5]	[1.5, -0.5]	[1.5, 0]	[1.5, 0.5]	[1.5, 1.5]
3	[2.5, -1.5]	[2.5, -0.5]	[2.5, 0]	[2.5, 0.5]	[2.5, 1.5]
4	[3.5, -1.5]	[3.5, -0.5]	[3.5, 0]	[3.5, 0.5]	[3.5, 1.5]
5	[4.5, -1.5]	[4.5, -0.5]	[4.5, 0]	[4.5, 0.5]	[4.5, 1.5]

Figure 4.10 where the box is almost off the frame when placed in positions [1, 1] and [1, -1]. When moving the box further to the side it is completely out of sight and an object can not be detected. This includes object positions [1, -2], [2, -2], [1, 2], and [2, 2]. In the case of object position [1, 0], the camera and the box were placed too close to each other, less than 0.6 units, making the camera unable to find the box.

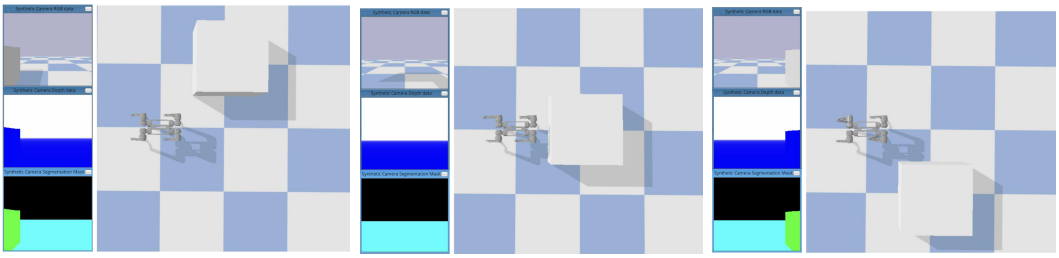


Figure 4.10: Box object placed at position [1, 1], [1, 0] and [1, -1] seen from image left to right.

4.2 Path planning

Figure 4.11 shows that a 1x1 box object is successfully simulated and placed in front of K3lso by clicking on the `Objects` button in the control panel. From the camera images, it is also seen that attaching the camera to K3lso was successful as well. It is seen that the object is also detected and colored green in the segmented image. The yellow line shows the path K3lso has been instructed to walk by specifying the positions of two points, $[x_1, y_1]$ and $[x_2, y_2]$, in the GUI. The red, green, and blue arrows pointing in different directions show the positive x-, y-, and z-directions.

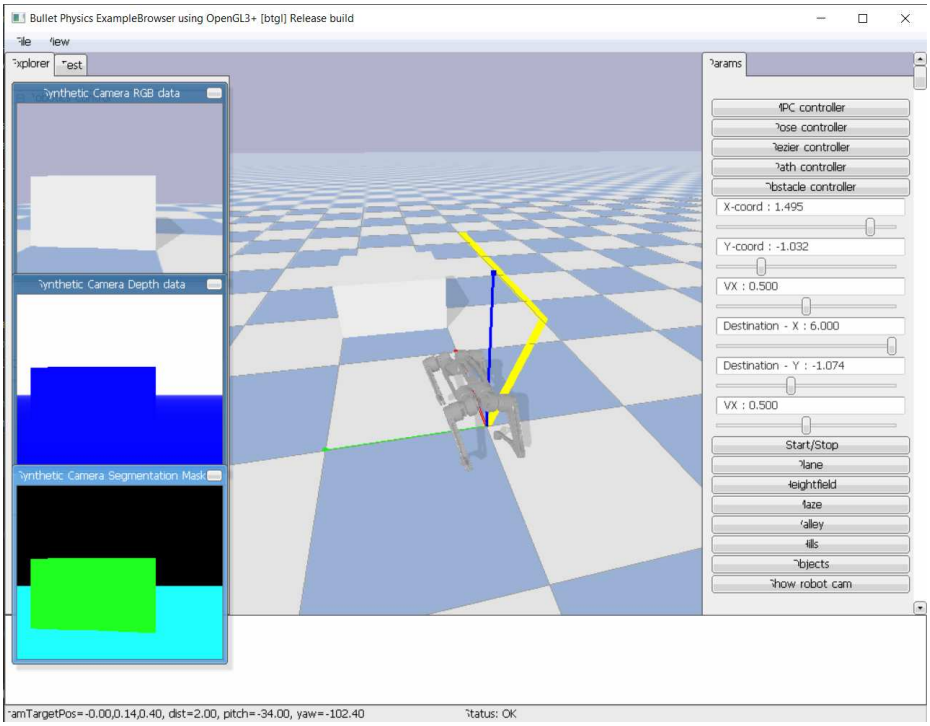


Figure 4.11: The developed PyBullet simulation of K3lso.

Moving on to the path planning algorithm using potential fields. Figure 4.12 below shows 2-D plots of how the path planner plans the robot path for three different scenarios with obstacle position [2, 0], [2, 1] respective [2, 2]. The top and middle image shows how the path planner suggests a path to avoid the obstacles by turning to the right and then back to a straight path. The bottom image shows how the robot does not find an obstacle but generates the default value. Without obstacles, the algorithm generates a straight path to the target position.

The robot's current position is marked as a black star, the detected obstacle is a blue box, the suggested path is the red dots and its target position is marked as a magenta-colored star. The x- and y-axis of the plot are displaced by 5 units, where position [5, 5] represents the coordinate [0, 0] in the simulation. Furthermore, two units in the plot equal one coordinate unit in the simulation.

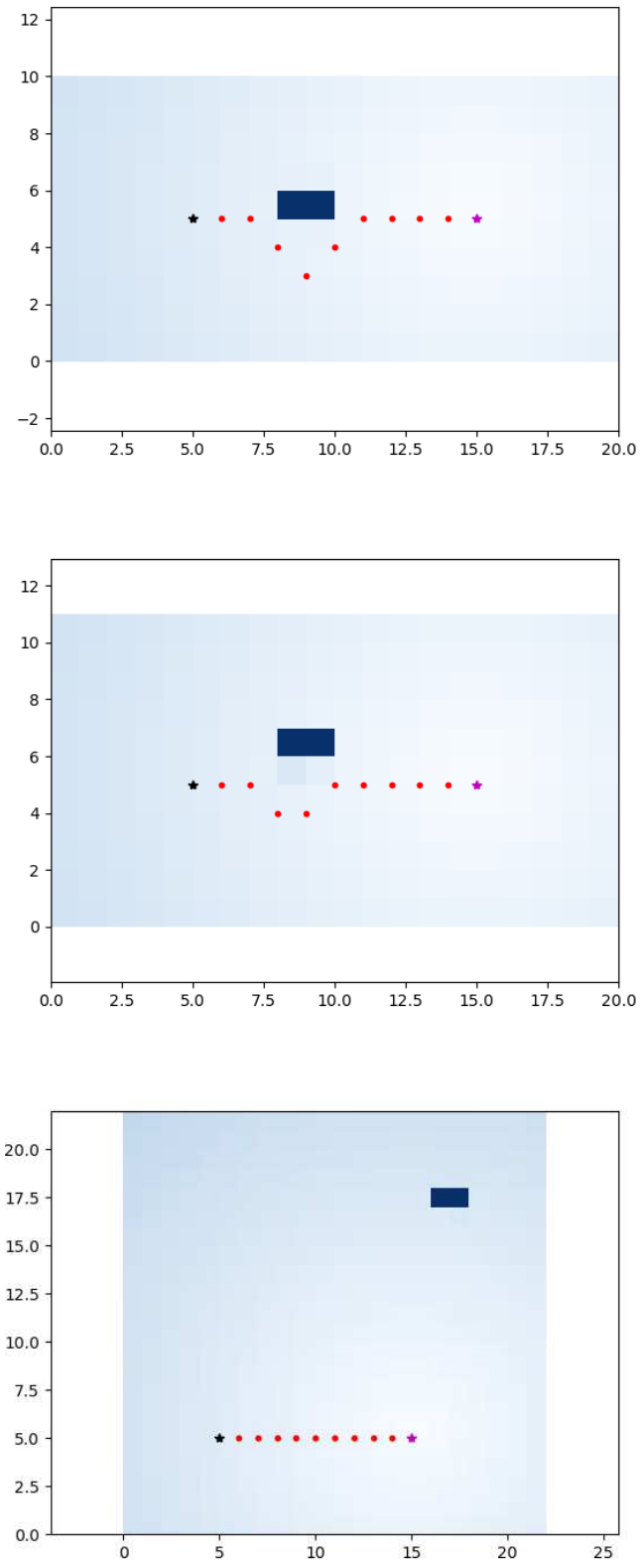


Figure 4.12: Three different scenarios on how the path planner algorithm using potential fields generates a path to walk from [0, 0] to [5, 0]. Top - Obstacle at position [2, 0]. Middle - Obstacle at [2, 1]. Bottom - Obstacle at [2, 2].

Figure 4.13 shows in the third line of code that it has detected an obstacle at roughly $[0.5, 0.07]$ in the left side of the camera picture imported by the path following algorithm. The simulation rendering as can be seen in Figure 4.14 shows that this can not be the case as the obstacle is placed at $[2, 0.5]$.

```
Found obstacle!
pixel_min_y 80
3. obstacles_list LEFT side [0.5127805690739669] [0.06918095023052592]
[[ 2.29006646e-01 -4.96356627e-03]
 [-2.70993354e-01  4.95036434e-01]
 [ 2.29006646e-01  9.95036434e-01]
 [ 7.29006646e-01  9.95036434e-01]
 [ 1.22900665e+00  9.95036434e-01]
 [ 1.72900665e+00  9.95036434e-01]
 [ 2.22900665e+00  9.95036434e-01]
 [ 2.72900665e+00  9.95036434e-01]
 [ 3.22900665e+00  9.95036434e-01]
 [ 3.72900665e+00  9.95036434e-01]
 [ 4.22900665e+00  9.95036434e-01]
 [ 4.72900665e+00  9.95036434e-01]
 [ 5.05263138e+00  1.01052618e+00]]
```

Figure 4.13: Output showcasing the planned path.

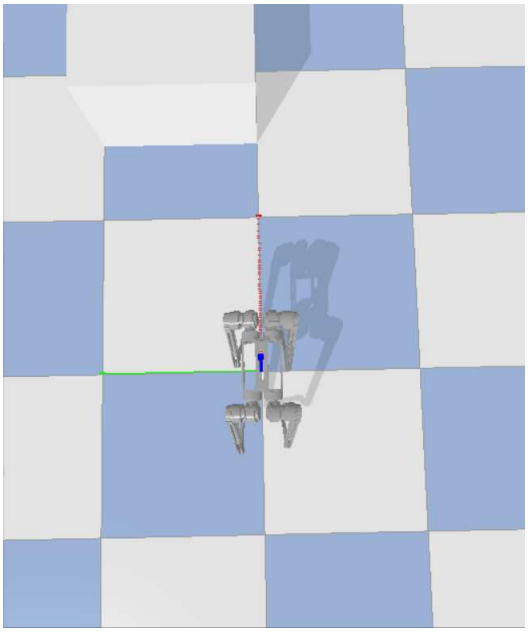


Figure 4.14: The actual position of the obstacle in the simulated world as at [2, 0.5].

4.3 Path following performance

The simulated model can walk to points in 2 different ways. It can walk to a point by walking a straight path to the point after rotating in place, or it can start walking instantly and turn as it is walking to get to the point. Tables 4.3 and 4.4 show how long time it takes for the robot to walk between the same two points, starting at $[0, 0]$ and ending at the goal, in a straight, respectively curved, path.

Table 4.3: Values obtained using the path controller.

Real translation	Time	Goal
0,7071 m	7,9982 s	[0.5, 0.5]
1,4142 m	11,3858 s	[1, 1]
2,8284 m	19,7168 s	[2, 2]

Table 4.4: Values obtained using the curve controller.

Real translation	Time	Goal
0,7071 m	7,8135 s	[0.5, 0.5]
1,4142 m	15,9368 s	[1, 1]
2,8284 m	29,0435 s	[2, 2]

The data in Figure 4.15 is obtained from walking between 2 points with different distances and comparing the times of walking a straight or curved path starting from the origin. Looking at the plots of time over distance for the different path styles it is seen that the path controller instead of the curve controller becomes faster the longer the distance is between two points. If the points are more than 0.71 meters apart the straight walking method will then be the faster option.

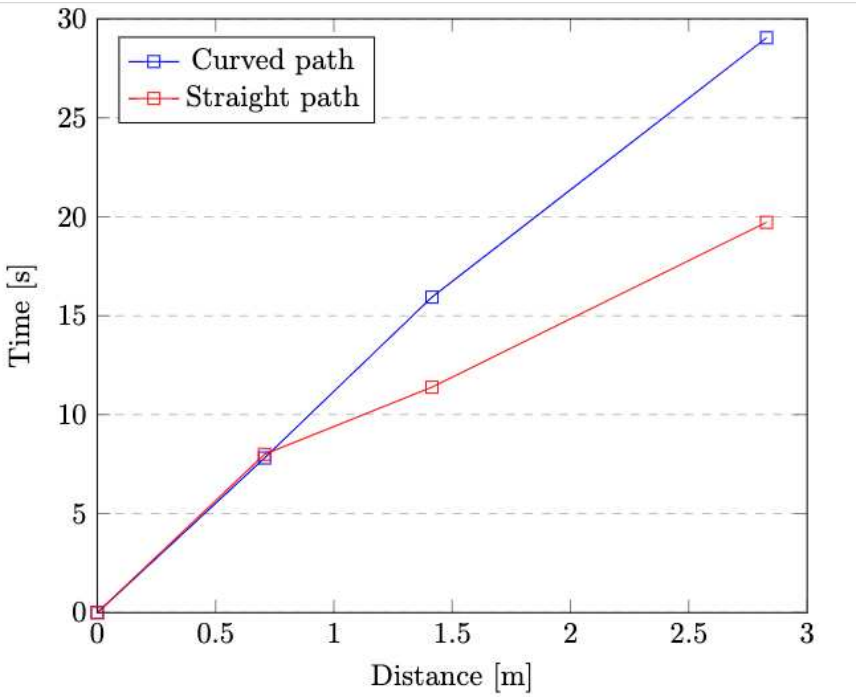


Figure 4.15: A graph of the time it takes to travel a distance in a curved or straight path.

In Figure 4.16 differences in the accuracy of the two different controllers can be seen. The path controller has higher accuracy. The accuracy does not seem to be dependent on how far the robot walks. Both the path and curve controller arrive with a distance shorter than 5 centimeter on the x- and y-axis.

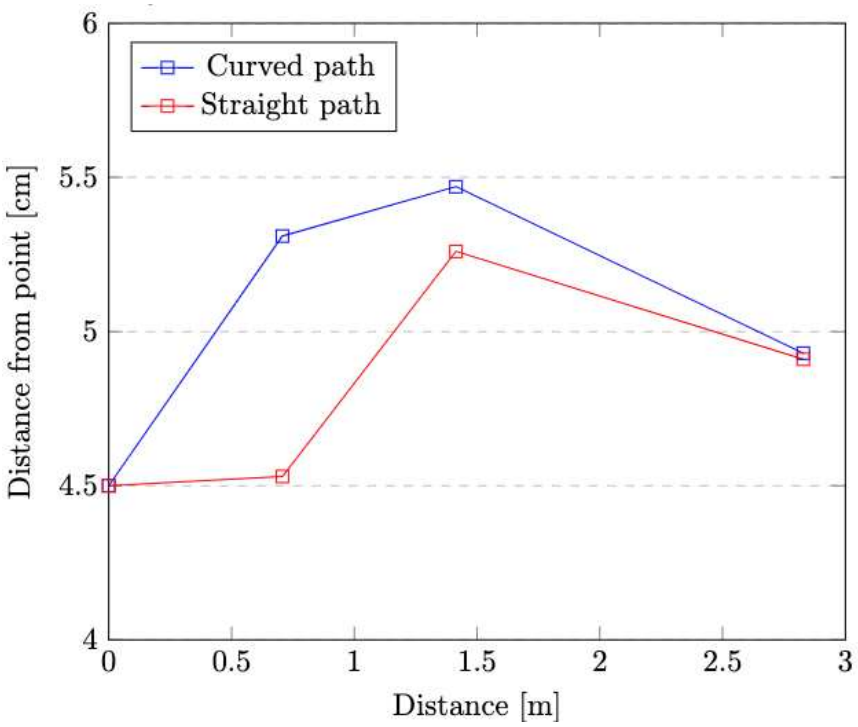


Figure 4.16: A graph of the distance from the determined point to show accuracy.

The path following algorithm is not finished as it can not update its next point of arrival after the path vector is updated.

5
Discussion

In this chapter, the main research questions about image processing, path planning and path following presented in section 1.2.1 will be discussed and answered using the results from the previous chapter.

5.1 Image processing

The image processing functions as intended. In the defined optimal conditions for the camera, it is able to mostly accurately identify obstacles, with the exception of thin objects laying on the ground.

5.1.1 Obstacle detection

The camera detection works relatively well and in the right environment could probably be enough for real-world use on K3lso. The limitation to only work on flat surfaces is a big part of why, but the margin added on top of the reference vector means that this solution might not work even if the area is flat. The cameras failed to detect the eraser on the floor which means that this solution would not work in an environment where there might be smaller objects on the floor. For the tests, the margin was set relatively high and could possibly be fine-tuned to be able to detect smaller objects. Time was not spent on fine-tuning, but potential problems that could arise are that the camera might move vertically or tilt when the robot moves. This is the reason the margin was put higher than what might be necessary to demonstrate that this is a potential problem.

The detection tests also show some difficulty with depth detection around the objects, especially if the object casts a shadow as the chair does in the middle picture of figure 4.5. On this side, the camera has a small area where it detects zero depth and has also created some artifacts, extending the object into this area. This should not be a problem since K3lso would want to keep some distance from any obstacle that it is avoiding anyway. This could become a problem in tighter spaces if these artifacts make it seem that the distance between two objects that K3lso is able to walk between is too narrow. It could also be a bigger problem if the artifacts become significantly larger.

A more complex route that the image processing could have gone in, is the AI route. Machine learning can be used to create a model for object detection. The RGB sensor on the camera can then be used to detect objects and determine if they are obstacles, and the depth sensor can then be used to determine the distance to the object. This would be far more complicated to develop and would without a doubt introduce problems that the method developed for this project does not have. A robot might also have limited computing power and a program that is simple to run could be preferable. Time might be an important factor, so to be able to quickly tell the path planning where obstacles are could make for smoother and faster navigation. Thus, a simpler model could be beneficial for certain applications or robots.

5.1.2 Camera functionality

As shown in Figures 4.8 and 4.9 the camera has difficulty recognizing objects or surfaces that are transparent or reflective. Figure 4.8 shows this clearly by having the camera face a window. A diagonal red line is also shown in the color map, which indicates that an object is detected at a distance far away from the camera. Since the lower picture in the figure shows no such object, it can be concluded that the transparent window itself, or the light passing through it, causes problems for the camera's depth measurements. This means that K3lso might not be able to detect transparent objects in its path.

The graph in figure 4.9 only shows data from the lower half of the recorded image, meaning that the table is the only object that is measured. This should be shown as a slope like the right half of the graph in 4.2. Instead, at pixel 280, an object at a distance of 13 m is shown to be present, which is further away than the walls of the room. Another object is detected at pixel 420 at a far shorter distance which can be a correct measurement, but since the majority of the data points are left blank, no conclusions can be made other than that the measurement as a whole is unusable. As no other data is received the camera can not notice that there is a table below or in front of it. Since the floor needs to be accounted for when surveying an area for obstacles, a gap in the data will be interpreted as a gap in the surface, like a hole or an edge.

5.1.3 Obstacle position estimation

From Table 4.1 it is seen that the estimation of object positions for $y = -1$ and $y = -2$, and respectively $y = 1$ and $y = 2$ are quite similar or identical. This is reasonable since the algorithm for obstacle position estimation only checks if the obstacle pixel with the lowest depth value was found is either on the left side of the image, the right side of the image, or within columns in the middle of the image. Thus, the algorithm does not take into account how far to the left or right the obstacles detected are positioned.

The are many shortcomings with these estimations as only objects placed around the center of the image are estimated with a margin of error of less than 0.2. This low accuracy could be a factor in a defective path planner algorithm. Possible improvements could be to instead estimate the angle to an obstacle as described in Appendix B. With this, where an obstacle is detected in the horizontal line is taken into account and a more accurate estimation would be possible.

5.2 Path planning

Given the incomplete state of the path planning with the potential field algorithm, there are not many concrete results to measure. From what could be seen from the produced results in Figure 4.12 it seemed to follow the points given to it by the path planner. The path avoided the obstacle when in the way and ignored it if it was out of reach. But since the path planner and by extension the camera implementation in our Python environment seemed to feed the path follower the wrong directions when used in conjunction with one another. A clear result was never achieved since the simulation would run into problems before it was able to follow the path planner through to the end. In a real-world scenario, there could eventually be more obstacles interrupting the planned path that has yet to be discovered after calculating the new path. This should not be a difficult task to implement, but time constraints have left it for further development.

5.3 Path following

Path following was implemented with the ability to set the speed of K3lso between different points. This approach was better than being able to choose a time since K3lso's max speed could be set manually to a speed guaranteeing success. If instead a time to complete the path could be chosen it would be easy to give the robot a too short time where the speed would become too fast and make K3lso fall over, effectively stopping the simulation.

When turning while walking, K3lso could fall over. This problem arose when setting K3lso's velocity higher than the pre-set one (0.5) for the path controller and higher than (0.25) for the curve controller. This is most likely because of an error in the mathematical modeling done on the gait style and is likely fixable with more investigation. Nevertheless, as explained in the limitations of the report, the controller was to be used but not changed for this project.

Between the curve and path controllers, the path controller arrived at its target position faster at longer distances, while there was no real-time difference when walking 0.7 meters or shorter. Considering how the paths are constructed from the potential fields and that the points are placed in a 0.5m grid, most points will be below the threshold where the straight controller is faster. When measuring the distance from the exact destination where the robot arrived it was found that the path controller had higher accuracy. Both controllers are constructed so you can set the speed of K3lso between points, at higher speeds the robot becomes unstable and can fall over. The curve controller becomes more unstable than the path controller and falls over at slower speeds. Additionally, the curve controller may run into issues because the potential field path planner assumes the robot walks straight between points. The robot could hit obstacles when walking a curved path between the points.

6
Conclusion

The obstacle detection works as intended. Large enough objects are detected as obstacles, but thinner objects might be below the floor margin. Shadows may also create artifacts. To improve the object detection, fine tuning of the floor margin would allow for smaller objects to be detected. Further experimentation with less optimal conditions and development to improve the program's ability to handle them. Lastly, a different model could be created using AI that utilizes object detection with the RGB sensor in combination with the depth data from the depth sensor.

The results from path planning with potential fields all pointed at the algorithm being successful. But since the path following of the planned path did not work, this is yet to be proven. For further implementation, the path planning could be run in an iterative way and be updated at each point.

Given specific velocities, K3lso could travel a pre-decided path without hinder. However, when the velocity was too high it could easily fall over when turning. To prevent this a closer look for defects could be taken at the mathematical modeling done on K3lso.

Given the results from Appendix A, it is clear that the physical version of K3lso is in need of some further diagnosis and reparation. If this is a fitting case for a bachelor's thesis or not could not be concluded, as it might be outside of both budget and the limit of a student's knowledge.

Bibliography

[1] R. Fröjd, "K3lso-overwiev," GitHub.com," Accessed: Feb. 2, 2023. [Online]. Available: https://github.com/raess1/K3lso-overview

[2] M. Husseini, M. Melander, P. Pettersson, A. Laveno Ling, A. U. Wilhelmsson, and L. Wincent, "Benroboten K3lso Stabil styrning av fyrbent robot för gång framåt- och i sidled," 2022. [Online]. Available: https://hdl.handle.net/20.500.12380/304739

[3] H. Safadi, "Local path planning using virtual potential field," 2007. [Online]. Available: https://www.cs.mcgill.ca/~hsafad/robotics/

[4] M. Schwenzer, M. Ay, T. Bergs, and D. Abel, "Review on model predictive control: an engineering perspective," *The International Journal of Advanced Manufacturing Technology*, vol. 117, no. 5-6, pp. 1327–1349, Nov. 2021. [Online]. Available: https://link.springer.com/10.1007/s00170-021-07682-3

[5] "Spot® - The Agile Mobile Robot," BostonDynamics.com," Accessed: 2023-02-20. [Online]. Available: https://www.bostondynamics.com/products/spot

[6] G. Bledt, M. J. Powell, B. Katz, J. Di Carlo, P. M. Wensing, and S. Kim, "MIT Cheetah 3: Design and Control of a Robust, Dynamic Quadruped Robot," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct. 2018, pp. 2245–2252, iSSN: 2153-0866.

[7] "Legacy robots," BostonDynamics.com," Accessed: 2023-02-20. [Online]. Available: https://www.bostondynamics.com/legacy

[8] P. Biswal and P. K. Mohanty, "Development of quadruped walking robots: A review," *Ain Shams Engineering Journal*, vol. 12, no. 2, pp. 2017–2031, Jun. 2021. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2090447920302501

[9] A. Tengborg, V. Nassif, J. Runeby, L. Wisell, B. V. J. Johansson Leskinen, and F. Fjelddahl, "Styrning av en robothund," 2022. [Online]. Available: https://hdl.handle.net/20.500.12380/304735

[10] L. Wincent, "EENX15-22-11," GitHub.com," Accessed: Feb. 2, 2023. [Online]. Available: https://github.com/linawincent/EENX15-22-11

[11] "ROS 2 Documentation — ROS 2 Documentation: Foxy documentation," ROS.org," Accessed: Feb. 20, 2023. [Online]. Available: https://docs.ros.org/en/foxy/

[12] L. Willcocks, "Robo-Apocalypse cancelled? Reframing the automation and future of work debate," *Journal of Information Technology*, vol. 35, no. 4, pp. 286–302, Dec. 2020. [Online]. Available: http://journals.sagepub.com/doi/10.1177/0268396220925830

[13] E. Sloan, "Robotics at War," *Survival*, vol. 57, no. 5, pp. 107–120, Sep. 2015. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/00396338.2015.1090133

[14] B. G. Katie Kavanagh, "What is math modeling?" 2021. [Online]. Available: https://m3challenge.siam.org/resources/whatismathmodeling

[15] R. Explained, "Transformation," 2023. [Online]. Available: https://robotics-explained.com/transformation

[16] "Stereo Depth Solutions from Intel RealSense," IntelRealsense.com," Accessed: Mar. 31, 2023. [Online]. Available: https://www.intelrealsense.com/stereo-depth/

[17] "Developing depth sensing applications - collision avoidance, object detection, volumetric capture and more," IntelRealsense.com," Accessed: Mar. 31, 2023. [Online]. Available: https://www.intelrealsense.com/sdk-2/

[18] R. Siddiqui, "Path planning using potential field algorithm," 2018. [Online]. Available: https://medium.com/@rymshasiddiqui/path-planning-using-potential-field-algorithm-a30ad12bdb08

[19] S. Huang and X. Zhang, "Biologically Inspired Planning and Optimization of Foot Trajectory of a Quadruped Robot," in *Intelligent Robotics and Applications*, X.-J. Liu, Z. Nie, J. Yu, F. Xie, and R. Song, Eds. Cham: Springer International Publishing, 2021, vol. 13016, pp. 192–203. [Online]. Available: https://link.springer.com/10.1007/978-3-030-89092-6_18

[20] A. Holm, "K3lso20," GitHub.com," Accessed: 2023-05-11. [Online]. Available: https://github.com/augustholm/k3lso20

A

Troubleshooting of hardware

A.1 Control of K3lso

To figure out how K3lso will able to walk, an approach to control K3lso is constructed, see figure A.1. The first step is to successfully control a separate, detached motor to later control the motors implemented in K3lso as well. The second step is to control a joint to a desired angle which is essential for the third step, to control an extremity that consists of three links and three joints. Now, for the robot to move as one body and not each extremity separately, synchronization of all extremities is needed. Synchronization is when multiple legs move simultaneously and coordinated which enables robot control to a specific position and walking sequence.

The debugging of K3lso, more specifically examining the wiring and control boards, will be the first step to then work in parallel with image analysis, controlling a motor and developing an interface to give K3lso commands. With the latter two completed, control of a joint, and then a leg, can be achieved. Using this, a system for pose-control can begin to be constructed and eventually be able to walk with K3lso.

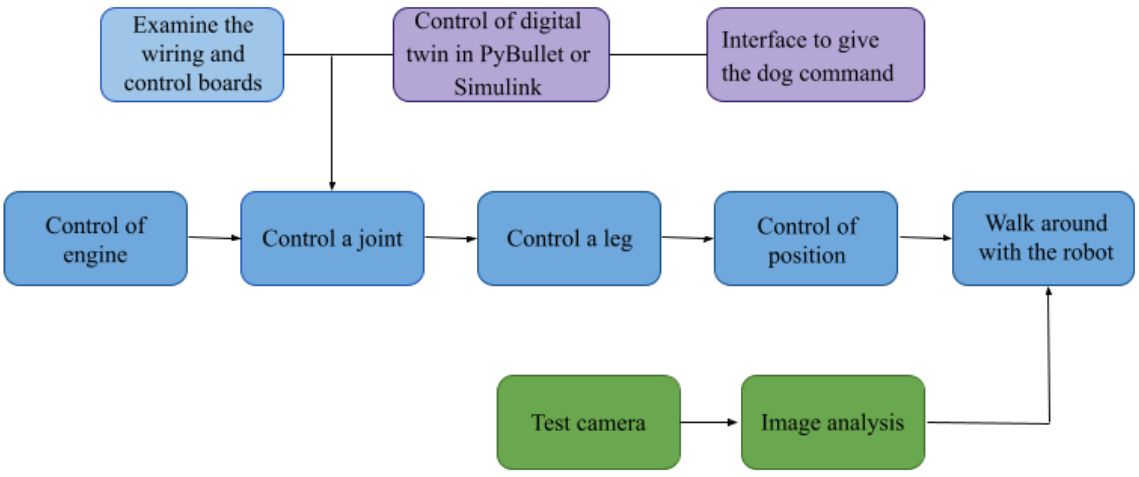


Figure A.1: Overview of the troubleshooting process for control of K3lso.

A.2 Examination of hardware

K3lso has many different components that must work together to make the robot work as intended. It is therefore relevant to investigate the hardware and how it should be used to get a more practical overview. This comes in the form of individual tests for the motors and then controlling multiple joints simultaneously. Once the joints can be controlled, it will be investigated how their movement overloads the cables connected to the joints as these have previously been broken or disconnected.

Via a screen connected to K3lso, visualization software can be used to clearly see what kind of signals the motors of the different joints send out. Currently, two out of six joints on the front legs send signals to programs. To fix this, it will investigate whether the data buses or controller cards are causing the problems, and then repair or replace broken components.

A.2.1 Measurements

Resistance measurements were performed in the CAN pins on the front leg motors to investigate what might be wrong. A multimeter was used to determine the resistance of the pin compared to the ground. The H and L pins were measured against each other, and also against the ground pin. Measurements done on the right front thigh were inconclusive.

Motor(position)	Ground to L[Ω]	Ground to H[Ω]
3 (Left front shin)	1250	350
6 (Left front shin)	90	60
5 (Left front thigh)	100	150
2 (Right front thigh)	?	?

Table A.1: Measurements from K3lso's problematic control cards

A.2.2 Disconnected motor

Code on GitHub to control the moteus board in the decoupled motor was used but gave no results. The program is supposed to perform a calibration, and when the program runs, a message appears referring to the motor moving. This does not happen.

A.3 Results of troubleshooting the hardware

Currently, the robot's movement causes cables to be pulled out of their contacts, and they can also be broken by overstretching or pinching. Tests also show that several of the control cards in the front legs of the robot do not work. This can be confirmed by a visualization program that can be run on K3lso. The program shows a model of the robot that in real-time shows how its joints move. When moving the front legs, only two of the six joints moved, meaning that the other four do not give any feedback to the robot's main computer. The measurements presented earlier also showcase these problems. The internal resistances should be a lot higher than measured, which implies that the cards themselves might be short-circuited, or that cables connecting the cards to the main computer cause a short-circuit. The connecting hub on the main computer is tightly sealed away behind several parts of K3lso's body, which means that it can not be fixed as is. If the cards themselves are short-circuited, it would mean that they would have to be replaced. There are three unused cards available to the group, so if four and not three joints are broken, more will have to be ordered. Since the measurements on one of the cards were inconclusive, it could not be determined if it was broken or if something else had happened. Replacing the cards would mean that the front legs would first be taken apart, and later the motors themselves. The cards are connected to the motors with soldered databuses and power cables. If the main computer was to be examined it would instead mean that large parts of the robot would be disassembled in hopes of finding where the issue lies.

It was concluded that trying to repair K3lso would take too much time with no guarantee of any results, which lead to the group not following through with any reparations.

B

Obstacle positioning

After the program has found an obstacle it needs to tell the path planning algorithm its location. This means that the camera data has to be converted to a form that the algorithm can use. When an obstacle is found the data that the image processing has is the pixel where the obstacle is found and the distance to it, but for K3lso to avoid it, a point in 3D space would be preferred. To be able to do this conversion the position and angle of the camera are needed. Then the measured distance to the obstacle and how far to the side the pixel is, from the vertical center line in the image, is used to calculate at what angle the obstacle is located at in correlation to the camera. The angle and the distance along with the camera position can then be used to calculate where the obstacle is.

Equation B.1 shows how the angle to the object could be calculated. This calculation is done in one-half of the camera's field of view, FOV, meaning the calculations are done as if the camera sees from the center line either to the left edge or to the right.

$$\theta_a = \frac{\frac{x_d}{x_h} * r\theta_{FOV}}{r} \tag{B.1}$$

x_d is the number of pixels from the center line to the object and x_h is the number of pixels from the center line to the edge, which would be half the pixels how the horizontal image size. r is the distance to the object and θ_{FOV} is half the camera's FOV in radians, so the FOV for the half that is calculated. This gives the angle to the object from the center line on the given half. This equation has been made, but not tested and assumes that the camera does no flattening to the depth image, something that cameras normally do, but is instead curved with the circle of the FOV. Thus to work properly compensating for any image processing that the camera does might be necessary.

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY