



**Faculty of Electrical Engineering**  
**Department of Control Engineering**

**Bachelor's thesis**


# **Collaborative Robot for Control of Small Touch Screen Devices**

**Hynek Zamazal**

**May 2022**

**Supervisor: Ing. Petr Čížek**





*"A robot may not injure a human being or, through  
inaction, allow a human being to come to harm."*

THE FIRST LAW OF ROBOTICS, ISAAC ASIMOV

## I. Personal and study details

Student's name: **Zamazal Hynek** Personal ID number: **491965**  
Faculty / Institute: **Faculty of Electrical Engineering**  
Department / Institute: **Department of Control Engineering**  
Study program: **Cybernetics and Robotics**

## II. Bachelor's thesis details

Bachelor's thesis title in English:

**Collaborative Robot for Control of Small Touch Screen Devices**

Bachelor's thesis title in Czech:

**Kolaborativní robot pro ovládání dotykových zařízení**

Guidelines:

- 1) Get familiar with a robotic arm, its control using forward and inverse kinematics [1], and methods of collision detection and isolation [2,3].
- 2) Get familiar with camera calibration of intrinsic and extrinsic parameters and detection of the small touch screen device on a scene [4].
- 3) Adapt the method [3] for use with a 3-DoF robotic arm to detect tactile-based collisions and estimate the device height.
- 4) Develop a library of arm motion primitives to perform a basic set of touch screen actions such as short click, long click, swipe, and double click.
- 5) Investigate the possibility of using the arm feedback in teach and repeat control of small touch screen devices.

Bibliography / sources:

- [1] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo. "Robotics: Modelling, Planning and Control", Advanced Textbooks in Control and Signal Processing. Springer London, 2009.
- [2] S. Haddadin, A. De Luca, and A. Albu-Schaffer. "Robot Collisions: A Survey on Detection, Isolation, and Identification," IEEE Transactions on Robotics, 33(6), 1292-1312, 2017.
- [3] J. Faigl and P. Čížek. "Adaptive Locomotion Control of Hexapod Walking Robot for Traversing Rough Terrains with Position Feedback Only", Robotics and Autonomous Systems, 116, 136-147, 2019.
- [4] E. Roy Davies. "Computer and machine vision: theory, algorithms, practicalities," Academic Press, 2012.

Name and workplace of bachelor's thesis supervisor:

**Ing. Petr Čížek Department of Computer Science FEE**

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **03.02.2022** Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

Ing. Petr Čížek  
Supervisor's signature

prof. Ing. Michael Šebek, DrSc.  
Head of department's signature

prof. Mgr. Petr Páta, Ph.D.  
Dean's signature

## III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

\_\_\_\_\_  
Date of assignment receipt

\_\_\_\_\_  
Student's signature



## Declaration

I declare that the presented work was developed independently and that I have listed all sources of the information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, May 20, 2022

.....  
Hynek Zamazal



## Acknowledgement

I would like to thank my supervisor, Ing. Čížek, for always being helpful, insightful and patient with my work, and to my colleagues from the Computational Robotics Laboratory for lending me their phones for testing without showing fear.

I would also like to extend my gratitude to my friends and family for supporting me throughout my studies.

## Abstract

Mobile phones and other small touch screen devices play a significant role in our everyday life. Although most of the tasks that can be performed on the touch screen device can be automated via software and emulation of the physical control, the configuration of the device to enable this software remains the sole operation that has to be done manually.

In this work we present a collaborative robot for setting up and control of small touch screen devices. The robot consists of an off-the-shelf robotic arm and a camera that observes the arm's workspace. We have developed a library of functions that allows for camera calibration, touch screen device detection, and determining the device's real world position and orientation in the workspace from the camera, as well as a set of motion primitives for the robotic arm to allow the basic touch screen interaction operations like click, long click, double click, and swipe.

On top of that, we have investigated and successfully deployed software enabled collaborativity of the arm based solely on the position feedback. This allows a teach and repeat functionality of the configuration scenarios as well as collision detection of the arm.

**Keywords:** collaborative robot, touchscreen interaction, automated phone configuration

## Abstrakt

Mobilní telefony hrají významnou roli v našem každodenním životě. Přestože většina úkonů, které se dají na těchto zařízeních provést se dá automatizovat pomocí softwaru a emulace fyzických interakcí, konfiguraci nového zařízení pro tento software je stále potřeba dělat manuálně.

V této práci představujeme kolaborativní robot pro nastavení a ovládání malých zařízení s dotykovými obrazovkami. Zařízení se skládá z robotického ramene a kamery, která sleduje jeho pracovní prostor. Vyvinuli jsme knihovnu funkcí, které umožňují kalibraci vnitřních i vnějších parametrů kamery, detekci dotykových zařízení a získání jejich pozice a orientace v pracovním prostoru robotu. Také jsme připravili sadu pohybových primitiv pro robotické rameno, které umožňují základní interakce s dotykovou obrazovkou jako je klik, dlouhý klik, dvojitý klik a tažení.

V rámci práce jsme také úspěšně testovali možnost softwarové detekce kolizí robotu s překážkami, čímž se nám podařilo z robota nekolaborativního vytvořit robota kolaborativního, a tedy i bezpečnějšího pro své okolí.

**Klíčová slova:** kolaborativní robot, ovládání dotykové obrazovky, automatizované nastavení telefonů

# Contents



<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem statement . . . . .	1
1.2	Robot description . . . . .	2
1.2.1	Hardware . . . . .	2
1.2.2	Software . . . . .	3
<b>2</b>	<b>Vision</b>	<b>5</b>
2.1	Camera calibration . . . . .	5
2.1.1	Intrinsic parameters . . . . .	5
2.1.2	Extrinsic parameters . . . . .	7
2.2	Position reconstruction . . . . .	8
2.3	Image undistortion . . . . .	9
2.4	Touch screen device detection . . . . .	10
<b>3</b>	<b>Arm control</b>	<b>13</b>
3.1	DexArm firmware . . . . .	13
3.2	Movement modes . . . . .	13
3.3	Kinematics . . . . .	14
3.3.1	Robot parameters . . . . .	14
3.3.2	Tick remapping . . . . .	15
3.3.3	Forward kinematics . . . . .	16
3.3.4	Inverse kinematics . . . . .	17
3.3.5	Marlin coordinate system . . . . .	19
3.4	Workspace . . . . .	19
3.4.1	Important positions . . . . .	20
<b>4</b>	<b>Collision detection</b>	<b>21</b>
4.1	Theoretical background . . . . .	21
4.2	Implementation . . . . .	22
4.2.1	Interpolation . . . . .	22
4.2.2	Models . . . . .	23



<b>5</b>	<b>Touch screen device control</b>	<b>27</b>
5.1	Phone coordinates . . . . .	27
5.2	Control actions . . . . .	28
5.2.1	Arm motion primitives . . . . .	28
<b>6</b>	<b>Teach and repeat</b>	<b>31</b>
6.1	Recording movement . . . . .	31
6.2	Action recognition . . . . .	31
6.3	Repetition . . . . .	32
<b>7</b>	<b>Results</b>	<b>33</b>
7.1	Camera calibration . . . . .	33
7.2	Touch screen device detection . . . . .	34
7.3	Arm control . . . . .	35
7.4	Collision detection . . . . .	36
7.5	Touch screen device control . . . . .	37
7.6	Teach and repeat . . . . .	38
<b>8</b>	<b>Conclusion and possible future work</b>	<b>39</b>
	<b>Bibliography</b>	<b>41</b>

# List of Figures

1.1	The robot. . . . .	2
2.1	Pinhole camera projection schematics. . . . .	6
2.2	Positions and orientations of the camera, Aruco marker and robot coordinate systems. . . . .	7
2.3	Comparison of a distorted and an undistorted image. . . . .	9
2.4	Camera image phone detection example. . . . .	10
3.1	Robot described using constants and variables with the joint coordinate systems according to the DH convention. . . . .	15
3.2	The arm's reach limitations. . . . .	20
4.1	The precision of $\theta_{est}$ when obtained through interpolation. . . . .	23
4.2	The precision of $\theta_{est}$ when obtained through the average speed method. . . . .	24
4.3	The precision of $\theta_{est}$ when obtained through the polynomial regression method. . . . .	25
5.1	Visualization of the workspace and phone coordinate frames. . . . .	27
6.1	Visualization of the teach functionality. . . . .	32
7.1	The touch screen device detection test setup. . . . .	34
7.2	An example of erroneous detections. . . . .	35
7.3	Collision detection using the interpolation method. . . . .	36
7.4	Collision detection using the polynomial regression method. . . . .	37



# List of Tables

3.1	Constants and variables of the robot. . . . .	14
3.2	DH parameters of forward kinematics. . . . .	17
7.1	Measured marker position and orientation. . . . .	33
7.2	Results of the device detection test. . . . .	34
7.3	Arm positioning repeatability. . . . .	35

# Introduction

In this day and age most people prefer to use their mobile phones and other small touch screen devices<sup>1</sup> to interact with friends, businesses and even their household appliances. This is a big incentive for companies to develop mobile applications which are quick and bug free.

Testing these programs on real devices is usually done by a remote access software and software emulation of physical interaction. However it is still necessary to set up the device and install the remote access software manually. This is a short 10 minute operation for one phone, but it adds up quickly. Therefore in cooperation with Škoda Auto we decided to design an easy to operate robot capable of automating the small touch screen device setup.

The simplest way to give a robot instructions is to show it what needs to be done. This robot will support the teach and repeat functionality not only as a simple repetition of movements, but also for phone control actions. Meaning the operator can show the robot how to setup one phone and the robot will be able to repeat these actions on another phone.

It will still be necessary to change the phones in the work area by hand. As the operator will be performing this task while the robot is running, the robot will need to detect collisions to prevent injury.

This thesis serves as a proof of concept for a collaborative robot using only position feedback. The replacement of additional sensors with a smart piece of software may make this technology cheaper and therefore more accessible.

## 1.1 Problem statement

The goal of this bachelor's thesis is to create a collaborative robot capable of identifying and controlling small touch screen devices.

The robot will detect all touch screen devices in an image from a camera capturing the robot's workspace, then translate the information about their position, size and orientation from the image to the real world using the camera's calibration parameters.

To move the robot arm in real world coordinates, it's forward and inverse kinematic models are going to be calculated. For collaborativity and phone thickness measurement, the method described by Faigl and Čížek (2019) will be adapted.

---

<sup>1</sup>In this thesis, the terms "touch screen devices", "mobile phones" and their synonyms will be used interchangeably, as their differences are irrelevant in this context.

For the device control itself, we will develop a library of motion primitives like *click*, *double click*, and *swipe*.

Lastly the robot will have a *teach and repeat* functionality. The operator should be able to guide the robot arm while performing a series of interactions with a small touch screen device. The robot will remember the location, type and order of the above mentioned motion primitives present in the guided interaction. This sequence of actions will then be repeated on another device in the workspace.

## 1.2 Robot description

### 1.2.1 Hardware

The base of the robot is a 80 cm by 60 cm rectangle made out of 40 mm aluminium extrusion with a white acrylic top. On one of its long sides is mounted a Rotrics DexArm.

The DexArm is an inexpensive consumer robot arm. It is suitable to our purpose as it is precise enough to accurately and repeatably interact with a touch screen device, but also simple enough to demonstrate the benefits of software enabled collaborativity. The DexArm is controlled by gcode commands send through a serial line. In the arm's gripper is mounted the pen holder attachment with an upside down pen. This pen has at its top (that is why it is upside down) a soft conductive ball from a rubber-like material, which allows the robot to interact with the touch screen devices.

Behind the DexArm is another aluminium extrusion holding a 3D printed camera mount 55 cm above the workplane. The camera chosen is a Basler Ace 2 industrial camera with a Basler C125-0418-5M-P f4mm lens. The camera is connected to the computer by an ethernet cable and communicates with a specialized library called *Basler Pylon*.

The whole setup is shown in Figure 1.1.



Figure 1.1: The robot.

### 1.2.2 Software

The interesting part of this work is the software which will be in detail described in the rest of the thesis.

It is a C++ library containing all the above mentioned functionalities as blocks to be used by a frontend developer. The code heavily relies on principles of object oriented programming. Excluding the standard libraries and libraries necessary for hardware interaction (*Basler Pylon*), we used *Eigen* (Guennebaud et al., 2010) for vector and matrix operations and *OpenCV* (Bradski, 2000) for image processing.



# Vision

The eyes of the robot is a Basler Ace 2 camera. This chapter describes how it is used to detect small touch screen devices in an image and then translate this information into the coordinate frame of the robot arm.

## 2.1 Camera calibration

To find the real world position of a point in an image plane we need to go backwards and first figure out, how the point is projected onto said plane. This is done using the *pinhole camera model*, which simplifies the problem by disregarding the camera lens distortion and using ray optics to calculate the projection.

This projection can be described as a transformation of point  $\vec{x}$  in 3D space to point  $\vec{p}$  in the image plane represented by a  $3 \times 3$  *camera matrix*  $K$  - the 9 values of matrix  $K$  are known as *intrinsic parameters*.

However, this transformation works for a point in camera coordinates, which are not the same as the robot arm coordinates. Therefore it is necessary to also find a  $4 \times 4$  matrix  $T$  representing the homogenous transformation between these two coordinate frames. The elements of  $T$  are called *extrinsic parameters*.

There are many different sources of information on camera calibration, this thesis uses the same notation as Davies (2012) Chapters 15 and 18.

### 2.1.1 Intrinsic parameters

The image plane is a plane with the  $z$  coordinate equal to the focal length of the camera lens  $f$ . The pinhole camera model works by projecting a ray from the origin (also known as the *camera center*) to the point being projected as shown in Figure 2.1. The intersect of this ray with the image plane is the image point.

This type of projection is called *perspective projection*. If the real point is described as a vector  $\vec{x} = (x \ y \ z)^T$  and the corresponding image point is  $\vec{x}_i = (x_i \ y_i \ z_i)^T$ , the relationship between these two can be described as

$$x_i = f \frac{x}{z}, \quad y_i = f \frac{y}{z}, \quad z_i = f. \quad (2.1)$$



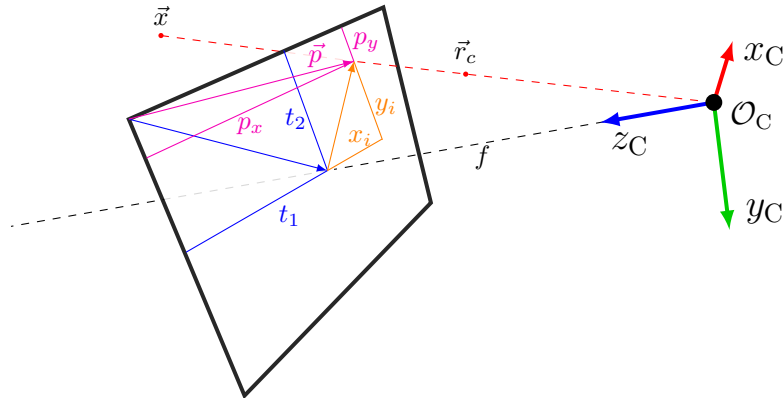


Figure 2.1: Pinhole camera projection schematics.

$x_i$  and  $y_i$  are distances from the  $z$  axis to the image point. But in an image the distances from the top left corner ( $p_x$  and  $p_y$ ) are important, so an offset  $\vec{o} = (t_1 \ t_2 \ 0)^T$  has to be added to  $\vec{x}_i$ .

Perspective projection is performed by two mathematical operations:

$$\begin{pmatrix} 1 & 0 & t_1 \\ 0 & 1 & t_2 \\ 0 & 0 & 1/f \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x + t_1 z \\ y + t_2 z \\ z/f \end{pmatrix} = \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix}, \quad (2.2)$$

and

$$\vec{p} = \begin{pmatrix} p_x \\ p_y \end{pmatrix} = \frac{1}{z_i} \begin{pmatrix} x_i \\ y_i \end{pmatrix} = f \begin{pmatrix} x/z + t_1 \\ y/z + t_2 \end{pmatrix}. \quad (2.3)$$

The  $3 \times 3$  matrix in Equation 2.2 prescribes the camera matrix  $K$  containing the intrinsic parameters. There is only one last thing unaccounted for: units. The image coordinates are in pixels, but the camera coordinates are in meters. This can be easily remedied by adding two scaling factors  $s_1$  and  $s_2$  so that

$$K = \begin{pmatrix} s_1 & 0 & t_1 \\ 0 & s_2 & t_2 \\ 0 & 0 & 1/f \end{pmatrix}. \quad (2.4)$$

Because of how the calibration method works, it is easier to search for less parameters. The format of  $K$  can be changed by multiplying it by  $f$ :

$$K = \begin{pmatrix} s_1 f & 0 & t_1 f \\ 0 & s_2 f & t_2 f \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} k_{11} & 0 & k_{13} \\ 0 & k_{22} & k_{23} \\ 0 & 0 & 1 \end{pmatrix}. \quad (2.5)$$

This change does not affect the results as can be seen in Equations 2.2 and 2.3.

Even though all of these parameters could be determined from the camera's datasheet, that would not account for focus adjustment and imperfections during manufacturing. That is why a ROS `camera_calibration` package was used. This tool is based around the `cv::calibrateCamera` OpenCV function which implements an algorithm described by Zhang (2000).

Given at least 2 different images of 3 coplanar points with known relative positions (we used an 8-by-11 checkerboard pattern), a matrix  $H$  is a maximum likelihood estimate of the

transformation between the plane defined by these three points and the image plane (the camera *homography*).  $H$  is a combination of the internal and external camera parameters and can be expressed as

$$H = \begin{bmatrix} \vec{h}_1 & \vec{h}_2 & \vec{h}_3 \end{bmatrix} = K \begin{bmatrix} \vec{r}_1 & \vec{r}_2 & \vec{t} \end{bmatrix}. \quad (2.6)$$

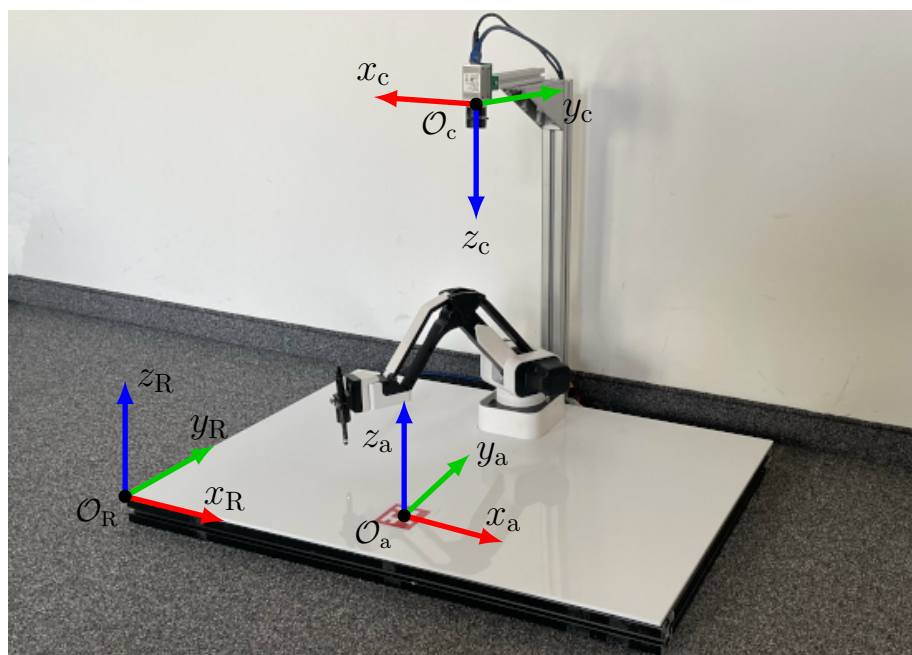
Equation 2.6 represents the multiplication of camera matrix  $K$  with the transformation matrix of extrinsic parameters between points in the  $XY$  plane of the world coordinate system and the camera coordinate system<sup>1</sup>. Therefore  $\vec{r}_1, \vec{r}_2, \vec{t} \in \mathbb{R}^3$ ,  $\vec{r}_1$  and  $\vec{r}_2$  are orthonormal.

$H$  is not just an ordinary matrix, but a homography. This fact in combination with orthonormality of  $\vec{r}_1$  and  $\vec{r}_2$  can be used to put two constraints on the intrinsic parameters and calculate them from the known parameters of the homography.

The resultant camera matrix for our camera is <sup>2</sup>

$$K = \begin{pmatrix} 1179 & 0 & 971 \\ 0 & 1179 & 567 \\ 0 & 0 & 1 \end{pmatrix}. \quad (2.7)$$

### 2.1.2 Extrinsic parameters



**Figure 2.2:** Positions and orientations of the camera, Aruco marker and robot coordinate systems.

Unlike with the intrinsic parameters that are always the same, the extrinsic parameters change based on the position and orientation of the camera. The camera mount is the least sturdy part of the robot and therefore it is possible for it to slightly move, for example during

<sup>1</sup>Because the extrinsic parameters are calculated separately, the assumption of coplanar calibration points lying in the global  $XY$  plane can be made without the loss of generality.

<sup>2</sup>The values have been rounded to whole integers here, but are used with precision of 11 decimal places in the code.

transportation. Because even small deviations could make the robot useless, it is beneficial to calibrate the extrinsic parameters on every startup.

That is why an Aruco marker was glued onto the center of the workspace. Each Aruco tag has its own coordinate system. Because the marker is on a flat plain of a known height ( $z = 0$  cm), it is easy to measure its position and orientation within the robot's coordinate system and find a transformation matrix  $T_R^a$  between those two. It can also be easily detected in an image and because we know the camera matrix  $K$  and the exact size of the Aruco marker, we can precisely determine its position and orientation in the camera's frame of reference, finding transformation matrix  $T_a^c$  between these two coordinate systems. Finding the full transformation matrix  $T$  is then a question of simple matrix multiplication. The coordinate systems referenced can be seen in Figure 2.2.

These transformation matrices are expressed in *homogenous coordinates*. To convert a vector from  $\mathbb{R}^3$  to homogenous coordinates one only has to append a 1 as the fourth element to the vector. This additional unitary fourth element allows us to express rotations and translations as matrix multiplication. A homogenous vector can be converted back to  $\mathbb{R}^3$  by dividing its first three elements by the fourth one.

The Aruco marker's coordinate system is located at  $\vec{t}_a = (39.56 \ 8.38 \ 0)^T$  and is rotated  $2^\circ$  (0.0349 rad) around the  $z$  axis. This transformation can be described using a matrix

$$T_R^a = \begin{pmatrix} R_z(0.0349) & \vec{t}_a \\ \vec{0}^T & 1 \end{pmatrix} = \begin{pmatrix} \cos(0.0349) & -\sin(0.0349) & 0 & 39.56 \\ \sin(0.0349) & \cos(0.0349) & 0 & 8.38 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \quad (2.8)$$

To find the position of the marker in the camera's coordinate system, the openCV function `cv::estimatePoseSingleMarkers` is used. It returns the position as a rotation vector and translation vector  $\vec{t}_c$ . The rotation vector is then transformed into a rotation matrix  $R$  using the Rodrigues formula,  $\vec{t}_c$  is multiplied by 100 to change its scale to centimeters<sup>3</sup> and the transformation matrix  $T_a^c$  is assembled:

$$T_a^c = \begin{pmatrix} R & 100 \cdot \vec{t}_c \\ \vec{0}^T & 1 \end{pmatrix} \quad (2.9)$$

It makes little sense to write the exact values of  $T_a^c$  here, as they are always slightly different. (See Section 7.1.)

The final matrix of extrinsic parameters  $T$  is

$$T = T_a^c T_R^a. \quad (2.10)$$

## 2.2 Position reconstruction

While the robot operates, it performs the inverse of perspective projection - an object is detected in an image and its position needs to be determined. In mathematical terms, 2D point  $\vec{p} = (p_x \ p_y)^T$  representing a picture of a real world object  $\vec{x} = (x \ y \ z)^T$  is known. The task of position reconstruction is to find the coordinates of the point  $\vec{x}$ .

If  $\vec{p}$  is changed to homogeneous coordinates by adding a 1 as the third element ( $\vec{p}_H = (p_x \ p_y \ 1)^T$ ), it can be multiplied by the inverse of the camera matrix  $K$  to obtain point

---

<sup>3</sup>The robot operates in centimeters as they are a nice scale to use for small touch screen devices.

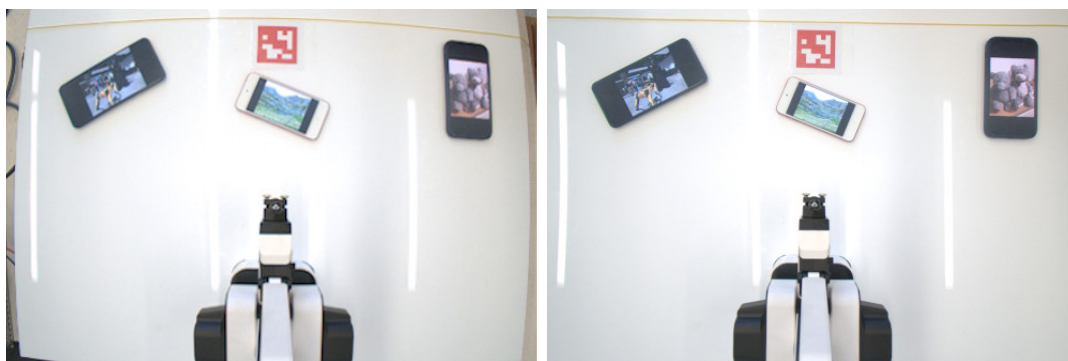
$$\vec{r}_c = K^{-1}\vec{p}_H. \quad (2.11)$$

As can be seen from Figure 2.1 a whole ray defined by the camera center and point  $\vec{r}_c$  is projected into point  $\vec{p}$ . To find the precise location of a small touch screen device, the collision detection function of the robot arm is used. The arm moves downwards along the path of the ray starting at a height of 5 cm above the work surface as it can be assumed no phone is thicker than that. Once the robot detects a collision, it has hit a device. The point of collision is the desired point  $\vec{x}$ .<sup>4</sup>

Equation 2.11 only gives the ray direction in terms of camera coordinates. For the arm to be able to follow the ray, it is converted to the arm frame of reference as a line defined by two points

$$\vec{r}_s = T^{-1} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}, \quad \vec{r}_e = T^{-1} \begin{pmatrix} \vec{r}_c \\ 1 \end{pmatrix}. \quad (2.12)$$

## 2.3 Image undistortion



(a) The distorted image.

(b) The undistorted image.

**Figure 2.3:** Comparison of a distorted and an undistorted image.

The used camera lens heavily distorts the image as can be seen in Figure 2.3a. This distortion causes straight lines to appear bulged out and is called *radial distortion* because the distortion depends on the radial distance from the center of the image.

The pinhole camera model used to find the real location of points from an image does not expect this distortion, so the image is undistorted before the detection of small touch screen devices.

The real position of a pixel should be  $(p_x \ p_y)^T$ , the center of the image is  $(p_{xc} \ p_{yc})^T$  and the pixel position in the distorted image is  $(p'_x \ p'_y)^T$ . The distortion function is

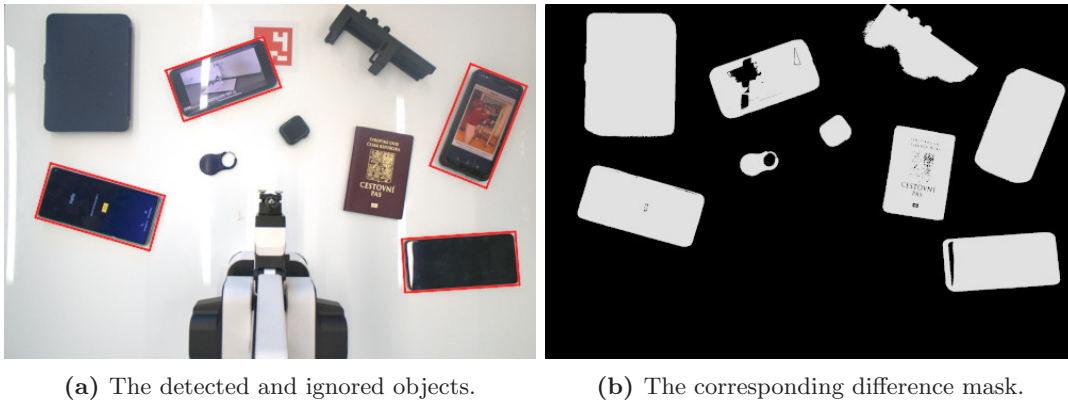
<sup>4</sup>As discussed in Section 7.4, the collision detection function was not utilized in the final deployment, hence the height of the point from the workplane has been entered manually during the setup.

$$\begin{pmatrix} p'_x - p_{xc} \\ p'_y - p_{yc} \end{pmatrix} = \begin{pmatrix} p_x - p_{xc} \\ p_y - p_{yc} \end{pmatrix} \left( a_0 + a_2 \left( \frac{(p_x - p_{xc})^2}{(p_y - p_{yc})^2} \right) + a_4 \left( \frac{(p_x - p_{xc})^4}{(p_y - p_{yc})^4} \right) + a_6 \left( \frac{(p_x - p_{xc})^6}{(p_y - p_{yc})^6} \right) \right). \quad (2.13)$$

The four distortion parameters  $a_0$ ,  $a_2$ ,  $a_4$ , and  $a_6$  are found using the same ROS `camera_calibration` tool as the intrinsic calibration parameters using the method described by Zhang (2000). After finding the intrinsic parameters of the camera, this tool finds a maximum likelihood estimate of the distortion parameters to minimize the error between where the calibration points are expected after multiplication by  $K$  and where in the image they truly are.

To undistort the image the OpenCV `cv::undistort` function, which implements the inverse of Equation 2.13, is used.

## 2.4 Touch screen device detection



**Figure 2.4:** Camera image phone detection example. The phones are detected and the other objects are ignored.

Part of the initialization process of the object detection class is moving the robot arm out of the view of the camera and taking a picture of the empty workspace. When object detection is requested a new image of the workspace is taken and a difference of the two images is calculated. The pixels that stayed the same are black, and the others are lighter and colored depending on the change. This difference image is then converted to grayscale<sup>5</sup> and thresholded so that all pixels lighter than 50 are white and all others black. Each of the white areas corresponds to an object in the workspace as is shown in Figure 2.4.

This threshold value was found experimentally. It was necessary to detect small changes, because the contents of the phone's screen may closely match the background, however if one detected every change, shadows started to be a problem.

The two parameters used for identifying a phone are its size (area) and its shape.

Every detected object with an area smaller than  $35 \cdot 10^3$  pixels or larger than  $150 \cdot 10^3$  pixels is not considered to be a phone. These values were found experimentally to reject small object like credit cards and too large ones like the operator's hands.

<sup>5</sup>Each pixel of the grayscale image can have a value from 0 (black) to 255 (white).

Minimum area bounding boxes for the remaining objects are found using the OpenCV function `cv::minAreaRect`. The ratio of their longer to shorter side is calculated and all objects which are too narrow (ratio larger than  $\frac{22}{9}$ ) or too square (ratio smaller than  $\frac{15}{9}$ ) are eliminated. These values were found by a short research into the aspect ratios of popular mobile phones. The narrowest ones have aspect ratios around  $\frac{21}{9}$ ; e.g., Sony Xperia 10, older phones tend to have the standard full HD aspect ratio  $\frac{16}{9}$ ; e.g., Samsung Galaxy Note 1-7. The ratios selected for filtering allow slightly broader selection to account for inaccuracies caused by; e.g., the camera not being exactly above the device, etc.

The remaining objects are considered to be phones and their real world positions, orientations and sizes are computed. The position of the phone is the position of its corner closest to the origin and orientation is the angle between the workspace x axis  $X_R$  and the shorter side of the device. These two parameters are important for coordinate transformations between the small touch screen device and the robot. More about that in Section 5.1.

Note, when searching for phones, the robot arm is docked, so that it does not obstruct the view of the workspace.



# Arm control

## 3.1 DexArm firmware

The Rotrics DexArm came with preinstalled open source firmware for 3D printers called Marlin created by Zalm (2011) modified to support the DexArm. Unfortunately this firmware was unsuitable to our needs, mainly because it was impossible to read the arm's position or stop the arm while the robot was moving.

To overcome these issues the DexArm firmware was adjusted by adding a custom program loop that continuously streams the motor positions to the serial line and also listens for custom commands: *initialize*, *move to* and *stop moving*. The *initialize* and *move to* commands are connected to their standard Marlin equivalents, but they bypass the blocking logic of Marlin command and movement sequencing.

The new DexArm firmware reads bytes from the serial line in a non-blocking mode and saves them into a buffer. While the received byte is not an end of the line symbol, the position of the robot's motors in encoder ticks is sent out. After a whole line has been read into the buffer, it is converted to a string and the specified command is executed. If the received text is not a known command, **ERR** is sent out. When factoring in the time needed for command execution, the arm reports its position with an average period of 2.2 ms.

The robot arm's motion is handled by an interrupt. The motion commands only write motion requests into a queue. Periodic interrupt checks the queue and if there is a motion to be executed, the motors are started. Then the interrupt periodically checks, if the movement has been finished and if so, it stops the motors. This logic allows the robot to still write out its position while moving and listen for the **stop** command, which immediately stops the robot if a collision is detected.

To execute a straight line move, the motion needs to be interpolated. This is done in a separate loop from the main one, causing the command reading and position reporting logic to be unavailable. To remedy this a copy of the main loop was implemented in this interpolation loop, but only listening for the **stop** command and writing out the robot's current position.

## 3.2 Movement modes

The arm can move from point to point in two ways: *fast mode* and *straight line mode*.



Fast mode sets the target motor positions and the motors move there at such speeds that all 3 of them finish the movement at the same time.

Moving in fast mode is useful for most robot movements because a curved trajectory of the robot's tip is not a problem. However for some operations, like the swipe gesture on a touchscreen, it is important to move in a straight line. That is done by interpolating the path between the current robot position and the target position. Because the motors have to be synchronized, this type of movement is slower.

The DexArm joints are moved by stepper motors. To prevent them from losing a step, an acceleration profile is implemented by the Marlin firmware. That makes movements in fast mode very smooth. However when the same approach was attempted for straight line mode - telling the arm to move in fast mode from one of the interpolated points to another - the robot stopped after each interpolation step making the movement choppy. Fortunately Marlin has a straight line movement function already implemented so the custom layer of firmware was modified to use it. This behaviour has also shown to be the main source of problems in the implementation of the collision detections further detailed in Chapter 4.

### 3.3 Kinematics

To accurately locate the tip of the robot arm, a way of translating the angular positions of the motors into a cartesian coordinate frame was found. Mathematical operation doing this is called the *forward kinematics*. To position the arm in said coordinate frame, the motor positions will be calculated from a cartesian vector. Because it is the opposite of forward kinematics, this task is called the *inverse kinematics*.

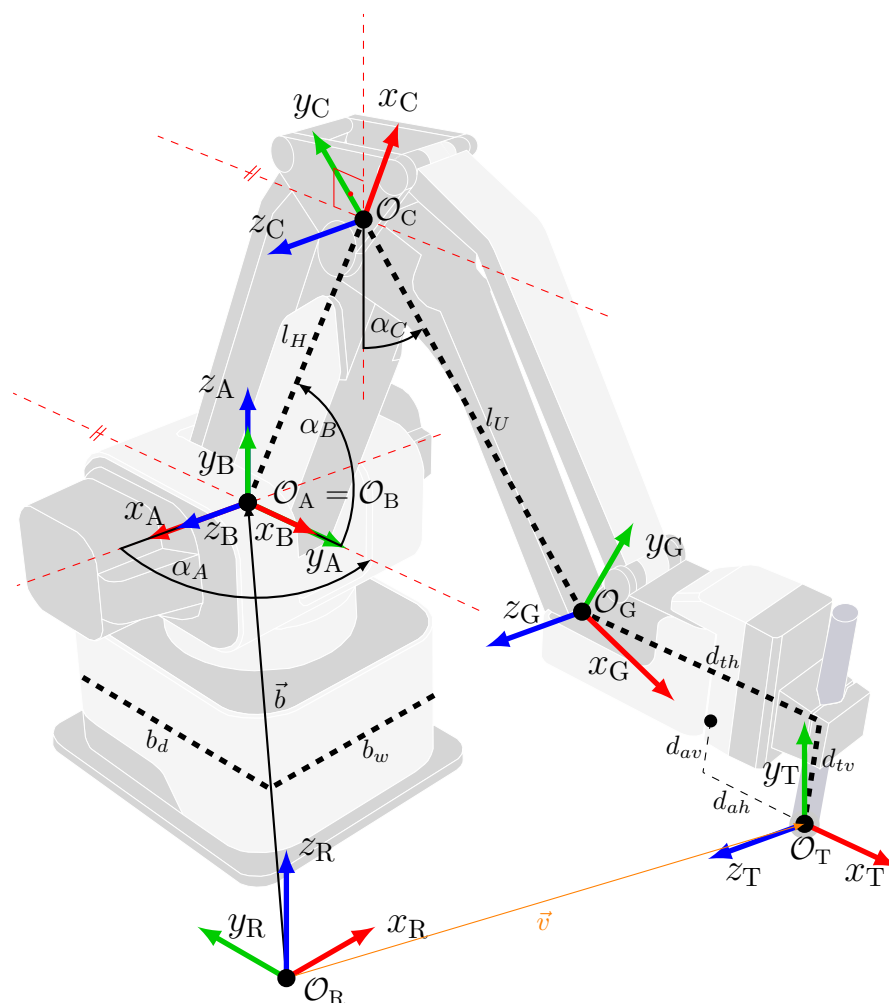
#### 3.3.1 Robot parameters

When calculating the kinematic models, constants representing the robot's structure and variables representing its motor positions are used. For clarity, their meaning and values are in Table 3.1 and Figure 3.1. In the calculations, only the symbols will be used.

Values of  $l_H$  and  $l_U$  were obtained from the Dex Arm's datasheet (Rotrics, 2022).

**Table 3.1:** Constants and variables of the robot.

Symbol	Meaning	Value
$l_H$	length of the robot's humerus	15 cm
$l_U$	length of the robot's ulna	15 cm
$d_{th}$	horizontal distance between the gripper base and the tip	10.5 cm
$d_{tv}$	vertical distance between the gripper base and the tip	6.5 cm
$d_{ah}$	horizontal distance between the gripper attachment point and the tip	4 cm
$d_{av}$	vertical distance between the gripper attachment point and the tip	6.2 cm
$b_w$	the width of the base of the robot arm	11.35 cm
$b_d$	the depth of the base of the robot arm	13.2 cm
$\vec{b}$	position of the robot base (origin of the $O_A$ coordinate system)	$(40 \ 46.15 \ 9)^T$
$\theta$	rotation of the $O_A$ coordinate system about the $z$ axis compared to $O_R$	$\pi$ rad
$\alpha_A$	motor A angle	variable
$\alpha_B$	motor B angle	variable
$\alpha_C$	motor C angle	variable



**Figure 3.1:** Robot described using constants and variables with the joint coordinate systems according to the DH convention.

### 3.3.2 Tick remapping

The DexArm has a 12 bit encoder on each motor giving it 4096 ticks for the whole 360 degree rotation. To convert an angle from ticks to radians, the encoder value is multiplied by  $\frac{2\pi}{4096}$  and vice versa. The motors cannot move freely and are restricted to these ranges:<sup>1</sup>

$$t_A \in \langle 0, 1341 \rangle \cup \langle 3389, 4095 \rangle, \quad (3.1a)$$

$$t_B \in \langle 0, 925 \rangle \cup \langle 3848, 4095 \rangle, \quad (3.1b)$$

$$t_C \in \langle 0, 705 \rangle \cup \langle 3536, 4095 \rangle. \quad (3.1c)$$

Only multiplying values in these intervals by the conversion constant would not yield the motor angles as described in Figure 3.1 because of conflicting positive directions and

<sup>1</sup>The encoder positions here are referred to as  $t_A$ ,  $t_B$  and  $t_C$ .  $\alpha_A$ ,  $\alpha_B$  and  $\alpha_C$  are the corresponding angle values in radians used for kinematic calculations.

misaligned zeros. For identifying the dynamic parameters of the robot arm in Chapter 4 about collisions, it is necessary for the encoder values to be continuous. That is why the tick values are remapped before unit conversion. It is also very important to convert calculated tick positions back into the original sets before they are send to the arm in a motion command.

The mapping function and its inverse for  $t_A$  are

$$f_A(t_A) : \langle 0, 1341 \rangle \cup \langle 3389, 4095 \rangle \rightarrow \langle 0, 2048 \rangle, \quad (3.2)$$

$$f_A(t_A) = \begin{cases} 1341 - t_A & t_A \in \langle 0, 1341 \rangle, \\ 5437 - t_A & t_A \in \langle 3389, 4095 \rangle, \end{cases} \quad f_A^{-1}(t_A) = \begin{cases} 1341 - t_A & t_A \in \langle 0, 1341 \rangle, \\ 5437 - t_A & t_A \in \langle 1341, 2048 \rangle. \end{cases}$$

For  $t_B$  are

$$f_B(t_B) : \langle 0, 925 \rangle \cup \langle 3848, 4095 \rangle \rightarrow \langle 0, 1173 \rangle, \quad (3.3)$$

$$f_B(t_B) = \begin{cases} 925 - t_B & t_B \in \langle 0, 925 \rangle, \\ 5021 - t_B & t_B \in \langle 3848, 4095 \rangle, \end{cases} \quad f_B^{-1}(t_B) = \begin{cases} 925 - t_B & t_B \in \langle 0, 925 \rangle, \\ 5021 - t_B & t_B \in \langle 925, 1173 \rangle. \end{cases}$$

And for  $t_C$

$$f_C(t_C) : \langle 0, 705 \rangle \cup \langle 3536, 4095 \rangle \rightarrow \langle -137, 1128 \rangle, \quad (3.4)$$

$$f_C(t_C) = \begin{cases} t_C + 423 & t_C \in \langle 0, 705 \rangle, \\ t_C - 3673 & t_C \in \langle 3536, 4095 \rangle, \end{cases} \quad f_C^{-1}(t_C) = \begin{cases} t_C + 3673 & t_C \in \langle -137, 423 \rangle, \\ t_C - 423 & t_C \in \langle 423, 1128 \rangle. \end{cases}$$

### 3.3.3 Forward kinematics

The forward kinematics were mostly calculated using the *Denavit-Hartenberg* (DH) convention described by Siciliano et al. (2009). Even though this method can be used only on open kinematic chains and the DexArm has a parallel mechanism, this mechanism is used just to keep the tip perpendicular to the ground which does not interfere with the DH approach.

Each joint of the robot arm is assigned a coordinate system of which the  $z$  axis is the axis of rotation. The forward kinematics are calculated by transforming the origin of the tip coordinate system  $\vec{o}_T$  from one joint reference frame to another, until it is expressed in the global robot coordinate system.

The DH convention describes each coordinate transformation in 4 steps: rotation about the  $z$  axis by an angle  $\vartheta$ , translation along the  $z$  axis by distance  $d$ , translation along the new  $x$  axis by distance  $a$  and lastly rotation about the new  $x$  axis by an angle  $\alpha$ . This means each coordinate transformation can be described by 4 DH parameters. The transformation matrix in homogenous coordinates <sup>2</sup> from  $O_a$  to  $O_b$  is

$$T_a^b = \begin{pmatrix} \cos(\vartheta) & -\sin(\vartheta) & 0 & 0 \\ \sin(\vartheta) & \cos(\vartheta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \quad (3.5)$$

<sup>2</sup>How homogenous coordintes work is described in Section 2.1.2.

The DexArm has 3 joint coordinate systems  $O_A$ ,  $O_B$  and  $O_C$ , a gripper and tip coordinate systems  $O_G$  and  $O_T$ , and the main robot coordinate system  $O_R$ . Their positions are shown in Figure 3.1.

Transformation between the  $O_R$  and  $O_A$  frames of reference is not done using the DH parameters. Instead a generic transformation matrix was used

$$T_A^R = \begin{pmatrix} R & \vec{b} \\ \vec{0}^T & 1 \end{pmatrix}, \quad \text{where } R = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}. \quad (3.6)$$

DH parameters for the next 3 transformations are in Table 3.2.

**Table 3.2:** DH parameters of forward kinematics.

Transformation	$\vartheta$	$d$	$a$	$\alpha$
$T_B^A$	$\alpha_A$	0	0	$\pi/2$
$T_C^B$	$\alpha_B$	0	$l_H$	0
$T_G^C$	$\alpha_C - \alpha_B - \pi/2$	0	$l_U$	0

The last transformation from  $O_G$  to  $O_T$  also deviates from the DH method. The coordinate system  $O_G$  has to rotate  $\frac{\pi}{2} - \alpha_C$  radians about the  $z$  axis to have the axes properly oriented relative to the workplane and then translate by vector  $\vec{t}_T$ . The matrix representing this transformation is assembled from these components

$$R = \begin{pmatrix} \cos(\pi/2 - \alpha_C) & -\sin(\pi/2 - \alpha_C) & 0 \\ \sin(\pi/2 - \alpha_C) & \cos(\pi/2 - \alpha_C) & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \vec{t}_T = R \begin{pmatrix} d_{th} \\ -d_{tv} \\ 0 \end{pmatrix} \quad (3.7)$$

and is

$$T_T^G = \begin{pmatrix} R & \vec{t}_T \\ \vec{0}^T & 1 \end{pmatrix}. \quad (3.8)$$

The matrix of forward kinematics is obtained by multiplying all of these matrices together

$$T_T^R = T_A^R T_B^A T_C^B T_G^C T_T^G. \quad (3.9)$$

The position of the robot arm's tip  $\vec{v}$  is the origin of the  $O_T$  coordinate system in the robot's global frame of reference

$$\vec{v} = T_T^R \vec{o}_T. \quad (3.10)$$

### 3.3.4 Inverse kinematics

The tip of the arm needs to be placed into position  $\vec{v}$ . First  $\vec{v}$  is transformed from the robot coordinate frame into the frame of reference of the base joint by multiplying it by the inverse of the matrix from Equation 3.6. Angle  $\alpha_A$  is calculated as

$$\vec{v}^A = \begin{pmatrix} v_X^A \\ v_Y^A \\ v_Z^A \\ 1 \end{pmatrix} = (T_A^R)^{-1} \vec{v}, \quad \alpha_A = \arctan \left( \frac{v_Y^A}{v_X^A} \right). \quad (3.11)$$

The end of the arm is not in the tip, but at the base of the gripper - the origin of the  $O_G$  coordinate system (see Figure 3.1). To move from the tip there, vector  $\vec{v}^A$  is translated to

$$\vec{v}^G = \begin{pmatrix} v_X^G \\ v_Y^G \\ v_Z^G \\ 1 \end{pmatrix} = \vec{v}^A + \begin{pmatrix} -d_{th} \cdot \cos(\alpha_A) \\ -d_{th} \cdot \sin(\alpha_A) \\ -d_{tv} \\ 1 \end{pmatrix}. \quad (3.12)$$

Because joints  $\alpha_B$  and  $\alpha_C$  give the arm only two degrees of freedom,  $\vec{v}^G$  can be expressed in just two dimensions as

$$\vec{u} = \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \sqrt{(v_X^G)^2 + (v_Y^G)^2} \\ v_Z^G \end{pmatrix}. \quad (3.13)$$

The elbow joint ( $O_C$ ) lies in the intersection of two circles with radii  $l_H$  and  $l_U$  and centers in the origin and  $\vec{u}$ . If the elbow joint is expressed as a vector  $(x \ y)^T$  it can be obtained by solving a set of two equations with two unknowns

$$x^2 + y^2 = l_H^2, \quad (3.14a)$$

$$(x - a)^2 + (y - b)^2 = l_U^2. \quad (3.14b)$$

Subtracting 3.14a from 3.14b, expanding and rearranging gives  $y$  as a linear function of  $x$ :

$$(x - a)^2 - x^2 + (y - b)^2 - y^2 = l_U^2 - l_H^2, \quad (3.15a)$$

$$a^2 - 2xa + b^2 - 2yb - l_U^2 + l_H^2 = 0, \quad (3.15b)$$

$$y = -\frac{a}{b}x + \frac{a^2 + b^2 - l_U^2 + l_H^2}{2b} = cx + d. \quad (3.15c)$$

$x$  is obtained by substituting 3.15c into 3.14a.

$$x^2 + (cx + d)^2 = l_H^2, \quad (3.16a)$$

$$(1 + c^2)x^2 + 2cdx + (d^2 - l_H^2) = 0, \quad (3.16b)$$

$$ex^2 + fx + g = 0, \quad (3.16c)$$

$$x_{1,2} = \frac{-f \pm \sqrt{f^2 - 4eg}}{2eg}. \quad (3.16d)$$

Equation 3.16d also serves as a test of reachability - if the term under the square root is smaller than zero, the circles do not intersect and therefore this position is unreachable by the robot.

By substituting 3.16d back into 3.15c two solutions are obtained. Because of the robot's structure, it is known the correct solution is the one with higher  $y$  value. The rest of the joint angles are

$$\alpha_B = \arctan\left(\frac{y}{x}\right), \quad \alpha_C = \arctan\left(\frac{a - x}{y - b}\right). \quad (3.17)$$

### 3.3.5 Marlin coordinate system

For straight line movements, the target coordinates have to be sent to the robot in the original Marlin coordinate system.

The Marlin coordinate system is the same as the  $O_A$  reference frame, except its scale is in millimeters. Because the straight line function used is the one from the original DexArm firmware, it does not account for the tip position. Instead it positions the connection point between the gripper and the pen attachment.

If the target tip position is  $\vec{v}$ . Its position  $\vec{v}^A$  in  $O_A$  and the angle  $\alpha_A$  are obtained as in Equation 3.11. The target position vector  $\vec{m}$  is

$$\vec{m} = 10 \cdot \left( \vec{v}^A + \begin{pmatrix} d_{ah} \cos(\alpha_A) \\ d_{ah} \sin(\alpha_A) \\ -d_{av} \\ 0 \end{pmatrix} \right). \quad (3.18)$$

Only the first 3 elements of  $\vec{m}$  are sent to the DexArm, as the fourth one is only there for the homogenous transformations.

## 3.4 Workspace

For the purpose of touch screen device control, the workspace of the robot is a 180 degree section of an annulus with the center in the origin of the  $O_A$  coordinate system as shown in Figure 3.2. The arm cannot go lower than  $z = 0$  cm and it is not necessary to go above  $z = 5$  cm.

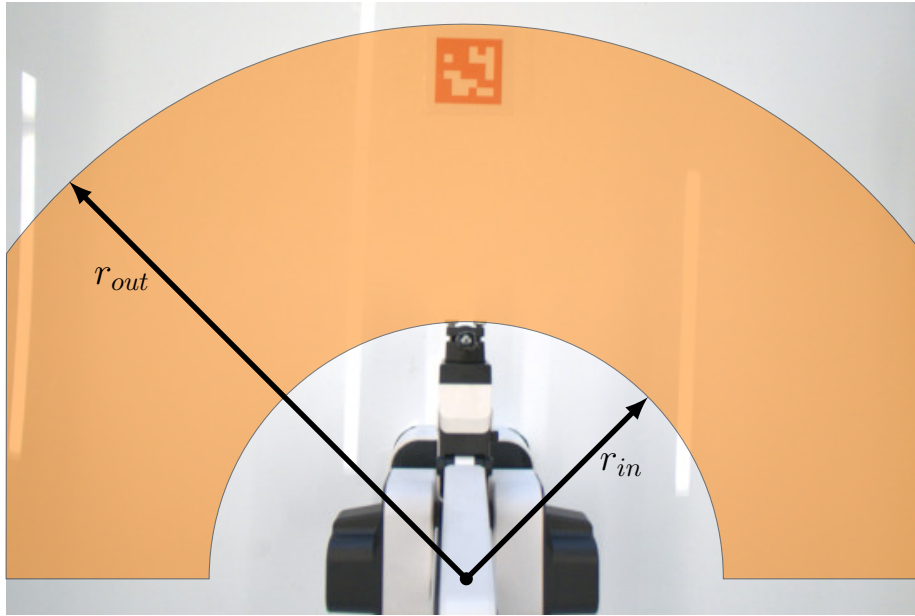
The inner radius  $r_{in}$  is the radius of a circle circumscribed to the robot base plus the horizontal distance between the gripper base and the robot tip. If it were smaller, the gripper would sometimes collide with the DexArm's base. It can be calculated as

$$r_{in} = d_{th} + \frac{\sqrt{b_w^2 + b_d^2}}{2} = 19.2 \text{ cm}. \quad (3.19)$$

The outer radius  $r_{out}$  is limited by the arm's reach. The farthest the arm can go is when both the robot's humerus and ulna are horizontal ( $\alpha_B = 0$  and  $\alpha_C = \frac{\pi}{2}$ ), in this position the tip is  $b_z - d_{tv} = 2.5$  cm above the workplane. However when the tip moves up or down, this distance decreases, because the arm has to bend. This change is the same in both directions and the height of maximum reach is in the middle of the chosen height interval, so the maximum reach is the same both at  $z = 0$  cm and  $z = 5$  cm. Because the boundary of reachability is close to elliptical in this section, we can guarantee nowhere within the height range will the reach be smaller than at the limits.

Therefore to obtain  $r_{out}$  it suffices to calculate the maximum reach at  $z = 0$  cm. In this position,  $\alpha_B = 0$  and the ulna is slightly bent downwards. The horizontal distance provided by the ulna can be easily calculated using the pythagorean theorem. The maximum reach of the arm is

$$r_{out} = l_H + d_{th} + \sqrt{l_U^2 - (b_z - d_{tv})^2} = 40.290 \text{ cm} \quad (3.20)$$



**Figure 3.2:** Schema of the workplane with the robot arm's reach limitations from the point of view of the robot's camera.

### 3.4.1 Important positions

There are two significant arm positions: *home* and *docked*.

#### Home

The *home* position is

$$\vec{h} = \begin{pmatrix} 39.41 \\ 13.1 \\ 3.7 \end{pmatrix} \text{ cm.} \quad (3.21)$$

This position is approximately in the middle of the workspace, making it a good ready position to access all the reachable devices. The arm can be easily grabbed from wherever the operator is sitting, making it a good starting position for the teach and repeat mode.

The *home* position coincides with the Marlin home position. The robot has to move there at startup to initialize the motor drivers and encoders.

#### Docked

The *docked* position is

$$\vec{d} = \begin{pmatrix} 39.41 \\ 32.46 \\ 5.48 \end{pmatrix} \text{ cm.} \quad (3.22)$$

To detect touch screen devices the arm should obscure as little of the workspace as possible. This position is technically outside of the robot's workspace as defined in section 3.4, therefore it cannot obscure any reachable devices.

# Collision detection

## 4.1 Theoretical background

The common methods used to detect collisions are described by Haddadin et al. (2017). Most of them rely on external torque estimation either by measuring the power consumed by (current passing through) the robot's motors, or by measuring the acceleration of different parts of the robot. However the DexArm allows only for the position measurements enabled by our custom firmware. This constraint rules out the power based methods completely and to convert position values into acceleration, double differentiation would have to be used. Differentiation introduces significant noise into the measurements and should be avoided.

This is where methods based on observers come in. Specific parameters like velocity can be simulated using a dynamic model or an observer, when the real measured value of said parameter is different from the prediction more then expected a collision has occurred (Haddadin et al., 2017).

Faigl and Čížek (2019) have implemented an algorithm based on this approach for a robotic leg with servo motors. One of the tasks of this thesis is to try to adapt their method for the robotic arm that uses stepper motors.

The core of this collision detection method is very simple. For each joint we have an estimate of where it should be  $\theta_{est}$  and we know where it really is  $\theta_{real}$ .  $e_{thr}$  is the maximum allowed error. Collision is detected when

$$|\theta_{est} - \theta_{real}| \leq e_{thr} \quad (4.1)$$

does not hold.

Complexities arise with acquiring the  $\theta_{est}$ . The simplest way is to estimate it using interpolation. When the robot is moving between positions  $\theta_i$  and  $\theta_{i+1}$ ,  $\theta_{est} = \theta_{i+1}$ . The precision of the estimate and therefore the reaction time to collisions depends on the size of the interpolation step as the maximum allowed error has to be at least the same.

To make the model more precise or when interpolation is impossible, a dynamic model of the robot arm is needed. Before a movement is executed, the positions of the robot's joints in time  $\theta(t)$  are predicted using this model. After a motion command is sent to the robot a timer is started and  $\theta_{est}(t) = \theta(t)$ . The more accurate the model, the quicker the reaction to a collision can be.



The main problem with the used robotic arm is that the particular motion of the stepper motors is governed not by our software, but by the Marlin firmware and therefore for the software defined collision checking module it behaves like a black-box which makes estimation of  $\theta_{est}$  particularly difficult, as described further.

## 4.2 Implementation

This section recounts our attempts at implementing the method of Faigl and Čížek (2019). They differ from each other in the way  $\theta_{est}$  is acquired. Because there are sometimes measurement errors in the motor positions, collision is detected only after equation 4.1 is broken two samples in a row.

### 4.2.1 Interpolation

The first attempted approach at obtaining  $\theta_{est}$  is just the simple interpolation. The desired movement is separated into as many sub-movements, such as there is no bigger difference between two position then 5 encoder ticks.

This is done by representing the joint angles of the start position as  $\vec{\theta}_s$  and of the end position as  $\vec{\theta}_e$ . The required direction  $\vec{\delta}$  and distance to cover  $l$  are then calculated to be

$$\vec{\delta} = \frac{\vec{\theta}_e - \vec{\theta}_s}{|\vec{\theta}_e - \vec{\theta}_s|}, \quad l = |\vec{\theta}_e - \vec{\theta}_s|.$$

The number of required steps  $N$  is calculated by the integer division of  $l$  by the interpolation step length  $s$  - in our case 5. The individual interpolation steps are then calculated as

$$\vec{\theta}_n = \vec{\theta}_s + n \cdot s \cdot \vec{\delta}, \quad n \in \langle 1, N \rangle, \quad \text{and} \quad \vec{\theta}_{N+1} = \vec{\theta}_e. \quad (4.2)$$

This interpolation is calculated so that the resultant trajectory is equivalent to the movement in fast mode, however it would work for any trajectory shape as long as the interpolation step is small enough.

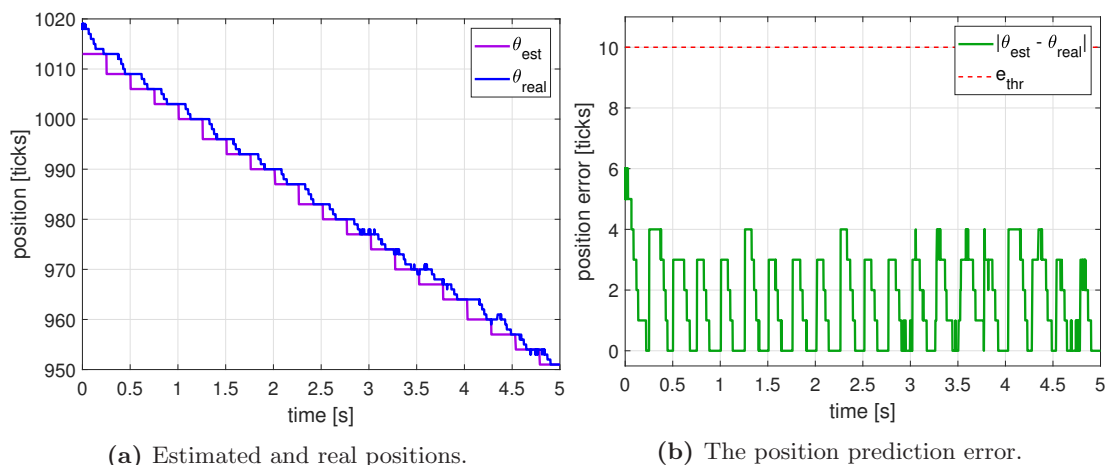
The command to move to  $\vec{\theta}_1$  is sent to the arm and  $\vec{\theta}_{est} = \vec{\theta}_1$ . Then the arm's position  $\vec{\theta}_{real}$  is read, subtracted from  $\vec{\theta}_{est}$  and each element of the resultant vector is compared with  $e_{thr}$  to check if Equation 4.1 holds. If a collision is detected, the **stop** command is sent to the arm and the rest of the trajectory is discarded, otherwise after 0.2 ms the target position and  $\vec{\theta}_{est}$  are updated to be  $\vec{\theta}_2$ . This is repeated until the desired final position is reached or a collision is detected.

The minimum  $e_{thr}$  for this to work would be 5 ticks, however we have to account for the acceleration profile and timing inconsistencies. That is why the smallest  $e_{thr}$  we found to be reliably working (with no false positives) is

$$e_{thr} = 10 \text{ ticks}. \quad (4.3)$$

That is 0.879°.

How the angle  $\alpha_A$  follows the interpolated trajectory and the  $\theta_{est}$  error are shown in Figure 4.1.



**Figure 4.1:** The precision of  $\theta_{est}$  when obtained through interpolation visualized for the angle  $\alpha_a$ .

## 4.2.2 Models

Because of the issues the DexArm has with interpolation, which are in detail described in Sections 3.1 and 7.4, it was decided to attempt the acquisition of  $\theta_{est}$  through a dynamic model. An attempt was made to match the position curves of the individual motors which have a shape similar to an arctangent, scaled and shifted based on the start and end positions of the motion.

After the motion command is sent, there is a nonconstant period of time in which nothing happens. This is caused by the timeout and queue structure of Marlin motion sequencing described in Section 3.1. On the other side of the motion, there is no reason why monitor the motor positions after the arm has reached its destination. When only slices of the data where the arm is actually moving are used, the curves are still not linear, however their interpolation with a straight line seems to be possible with tolerable errors.

Linear model of the first order has only one parameter - the speed of the movement. Because the knowledge of how long the movement will take is a prerequisite to other modeling methods and the motion speed can be calculated from a known movement time and distance it was a logical starting point. Also linear models are easy and fast to train and then use for online calculations.

We attempted two approaches to estimate the movement time. First by calculating the average movement speeds for each motor and dividing the movement distances by said average speeds. Second we used a polynomial regressor to estimate the time from the start and end motor positions.

These models were only prepared for fast mode as a proof of concept.

### 4.2.2.1 Data acquisition

To train the models and then verify them before their implementation into the C++ library, training and validation data are needed.

To gather the data the robot was left to pseudorandomly move in the workspace for 550 moves in fast mode. After each movement command was sent to the arm, the motor positions were recorded for 3 seconds, as no movements are longer than that, and saved into a text file with timestamps and the target arm position.

The recorded data was analyzed using MATLAB. All recorded movements are cropped between the start and end of the movement. The movement starts when all three motors move at least two ticks in the target direction and is considered finished when all three motors haven't moved for 5 samples and are closer than 5 ticks to the target position. The laxness in detecting the end of the movement has to be because of the steady state error of the DexArm's position controller. These values are also used in the final robot control library to detect the start and end of movements.

The first 500 movements are used as training data and the last 50 as validation data.

#### 4.2.2.2 Average speed

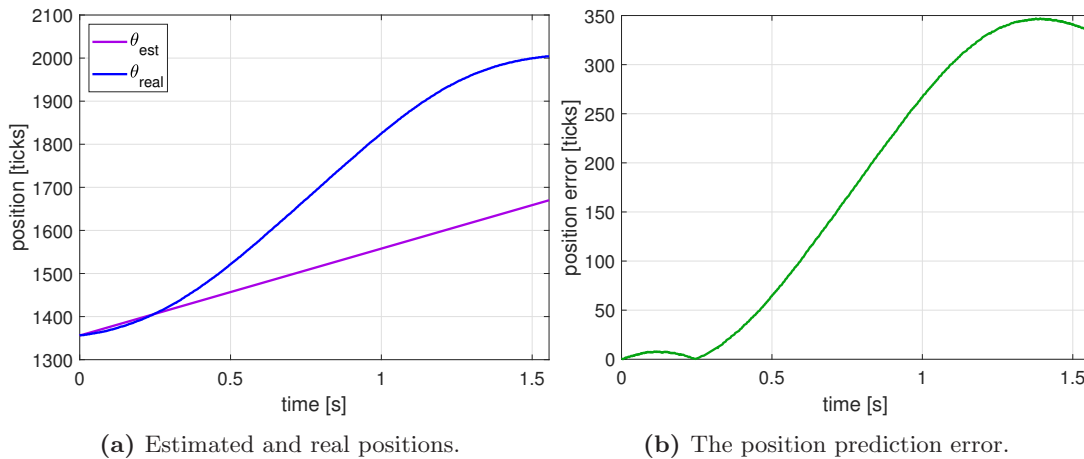
This method builds on the assumption that the average speed of each of the three motors can be estimated from the 500 training moves. Then it will be known where the movement starts and in what direction and speed is the joint moving. From this information the robot joint position can be estimated at any point in time after the start of the movement.

Because the duration of the movement is dependent on the joint with the largest angle difference, the time it will take is estimated from the average speed of the corresponding angle's motor and the speeds for the other joints are calculated so that all motors stop moving at the same time.

The average speeds were calculated to be

$$\dot{\alpha}_A = 201.802 \text{ ticks/s}, \quad \dot{\alpha}_B = 96.849 \text{ ticks/s} \quad \text{and} \quad \dot{\alpha}_C = 79.534 \text{ ticks/s}. \quad (4.4)$$

Figure 4.2a shows how the angle  $\alpha_A$  moves, and how it would move if it followed the trajectory predicted by the average speed method. The difference between the real trajectory and the estimate is shown in Figure 4.2b.



**Figure 4.2:** The precision of  $\theta_{est}$  when obtained through the average speed method visualized for the angle  $\alpha_a$ .

As can be seen from Figure 4.2 this method is not good enough to be worth implementing, as the error reaches 350 ticks -  $30.762^\circ$ .

### 4.2.2.3 Polynomial regression

The only difference between the average speed method and this one is the way in which the movement duration is calculated. Instead of estimating the time from the largest angle change and known speed, it is predicted using a polynomial regressor.

The regressor's input is a vector  $\vec{a} \in \mathbb{Z}^6$  of 6 values, the first three representing the starting position and the second three representing the target position. The desired output is the time the arm will take moving between these two positions in fast mode.

From the six input values the polynomial features are calculated. The polynomial features of degree  $N$  are all the terms in a polynomial  $\sum_{i=0}^N (x_1 + x_2 + \dots + x_6)^i$  without their constants. Meaning for vector  $\vec{v} = (a \ b)$  its polynomial features of the second degree are  $(1 \ a \ b \ a^2 \ ab \ b^2)$ .

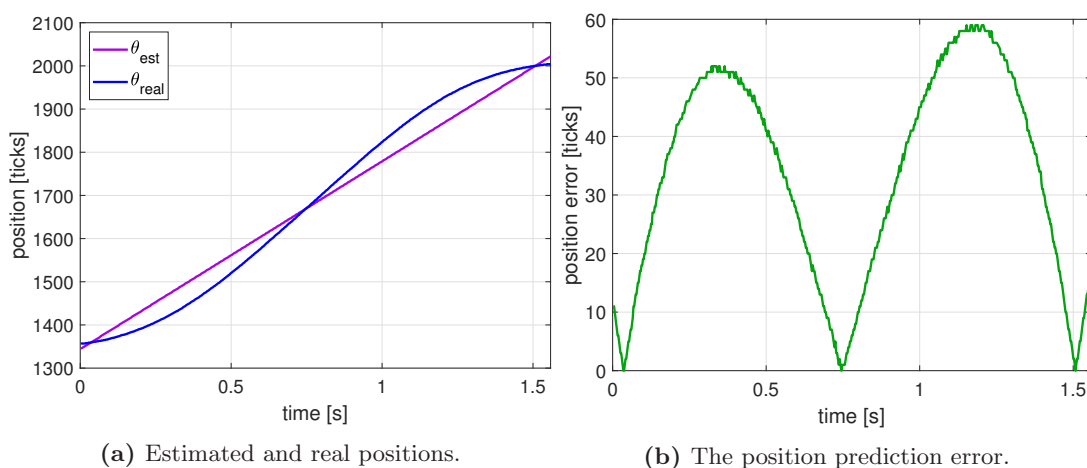
The predicted movement duration is the result of the sum of these features with different weights, which were obtained using the linear least squares method. Polynomials of degrees 1 to 9 were tried and best came out to be the polynomial of the third degree with mean error of 0.129s on the validation data.

There are 84 polynomial features of the 3 degree from a vector with 6 elements, if their vector is  $\vec{f}$  and the corresponding 84 weights are in the vector  $\vec{w}$ , the predicted time  $t_p$  can be calculated as

$$t_p = \vec{w}^T \vec{f}. \quad (4.5)$$

The time value and known movement angle differences are used to calculate the motor speeds which are with combination of the starting position and movement direction used to predict joint positions.

Figure 4.3 shows the same  $\alpha_A$  movement as Figure 4.2, but the estimated linear trajectory is based on the time estimate of the regressor.



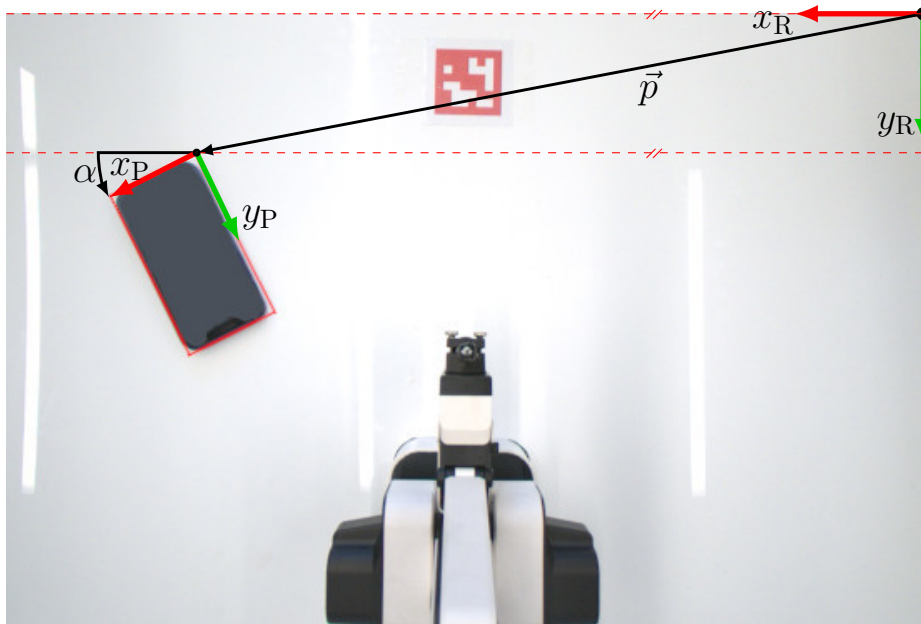
**Figure 4.3:** The precision of  $\theta_{est}$  when obtained through the polynomial regression method visualized for the angle  $\alpha_a$ .

Most of the validation data had the maximum error below 100, however 4 samples reached up to 140. To allow for these imprecisely predicted movements the chosen  $e_{thr} = 150$  ticks -  $13.184^\circ$ .



# Touch screen device control

## 5.1 Phone coordinates



**Figure 5.1:** Visualization of the workspace and phone coordinate frames.

Each touch screen device has its own 2D coordinate system used for the positioning of control actions. To execute these actions with the robot arm, a conversion between phone and world coordinates is necessary.

As described in Section 2.4 each phone's position can be described using two parameters: the position of its closest corner to the origin  $\vec{p}$  and the angle of the shorter side of the device and the global  $x$  axis  $\alpha$ . These parameters can be seen in Figure 5.1. It will be useful to know the components of the position vector represent the device's position in the XY plain  $\vec{q}$  and

its thickness  $t$ ,

$$\vec{p} = \begin{pmatrix} \vec{q} \\ t \end{pmatrix} = \begin{pmatrix} x_p \\ y_p \\ t \end{pmatrix}. \quad (5.1)$$

The origin of the phone coordinate system is at position  $\vec{p}$ , the  $x$  axis is along the shorter side of the device and the  $y$  is along the longer side ( as the devices are assumed to be rectangular). Note that the global coordinate system was chosen so that if the operator sits on the side opposing the robot arm and places a touch screen device in the workspace the right side up, the phone screen will be in the I. quadrant of the phone coordinate system.

If a 2 dimensional vector  $\vec{u}_P$  represents a position in phone coordinates of a device at position  $\vec{p}$  (with components according to Equation 5.1) and rotated  $\alpha$  radians, its 3 dimensional representation  $\vec{u}_R$  in global robot coordinates can be obtained as

$$\vec{u}_R = T_P^R \begin{pmatrix} \vec{u}_P \\ 1 \end{pmatrix}, \quad \text{where} \quad T_P^R = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & x_p \\ \sin(\alpha) & \cos(\alpha) & y_p \\ 0 & 0 & t \end{pmatrix} = \begin{pmatrix} R & \vec{q} \\ \vec{0}^T & t \end{pmatrix}. \quad (5.2)$$

The opposite transformation is done in a similar way using the notation from Equations 5.1 and 5.2

$$\vec{u}_P = T_R^P \vec{u}_R, \quad \text{where} \quad T_R^P = \begin{pmatrix} R^T & -\frac{1}{t} R^T \vec{q} \end{pmatrix}. \quad (5.3)$$

It is important to know this transformation works only for those  $\vec{u}_R$  lying in the  $z = t$  plane. Otherwise their representation in phone coordinates does not make sense.

## 5.2 Control actions

The robot controls small touch screen devices using a set of motion primitives. Each device detected in the workspace (How that is done is described in Section 2.4.) is represented as a **phone** object containing information about the device's position and orientation used to convert between its screen coordinates and global robot coordinates. Each motion primitive can be called as a function of said object.

For the teach and repeat functionality, **action** objects were created, representing the type and parameters of the action to be performed. If an action object is passed to a device object said action is performed on said device. How these actions are generated is described in Chapter 6.

The logic of touch screen device control like control element detection and action sequencing is beyond the scope of this thesis. The possibilities are discussed in Chapter 8.

### 5.2.1 Arm motion primitives

In this section the motion primitives available to the robot are described.

In preparation for execution of one of these actions, the robot moves to a *ready* position 0.6 cm above the place where the phone will be touched. After the action is performed the arm returns back to this position. The exception is the *swipe* motion, where the end position is 0.6 cm above the end position of the swipe.

All motions of the arm between the ready positions are part of the control primitive and are performed in the straight line movement mode.

Even though during phone interaction the arm directly touches the screen, the collision detection capabilities are not used, rather the previous knowledge of the phone's thickness is utilized to place the tip of the robot arm at an appropriate height.

### **Click**

A *click* is the simplest action, used for example to open an application. From the ready position the tip of the arm moves down, touches the screen, waits for 25 ms and lifts back up.

When the tip is supposed to touch the screen, it does not go down to the measured phone thickness, but 2 mm above. That is because the ball at the tip of the arm is compressible and the arm detects the collision later than the device detects a touch. This measure allows us to more precisely time the touch durations.

### **Long click**

A *long click* is in principle the same as a *click* the only difference is in the time the tip is touching the screen - for the *long click* it is 750 ms.

This control action can be used for instance to open element options.

### **Double click**

A *double click* is two *clicks* 200 ms apart. The position the arm moves to after the first click is not the full 6 mm above the screen but only 3 mm, this is to limit the motion delay from the acceleration profile and allow the arm to click faster.

Its use includes but is not limited to selecting text.

### **Swipe**

The *swipe* primitive is mostly used for scrolling and it is different from a *click*. From the ready position above the start position, the arm moves down to touch the screen then drags on the screen to the end position and lifts back up.





# Teach and repeat

The *teach and repeat* functionality allows the robot operator to grab the arm and guide it through a series of actions on a touch screen device. The program then is able to identify which actions were performed and replicate them on a number of other devices.

## 6.1 Recording movement

When the *teach* function is activated, the motor torque is turned off so that the operator can freely move the arm. The current program time is recorded as a start time to calculate timestamps for each position.

Then a separate thread is launched to record how the robot's tip position changes over time. The sampling loop records a sample on average every 2.2 ms.

## 6.2 Action recognition

After the recording stops, the recorded trajectory is parsed into the individual actions.

The program iterates over the trajectory points. It first determines, whether the point is above a touch screen device. That is done by disregarding the point's height and then converting it to the phone coordinates of every known phone in the workspace. If for one of the devices both  $x$  and  $y$  coordinates are positive and smaller than the phone's width and height, the point is above said device. Because the devices cannot overlap, it is certain the point can be only above one device at a time. If the point is not above a phone, it is discarded and the next point is analyzed.

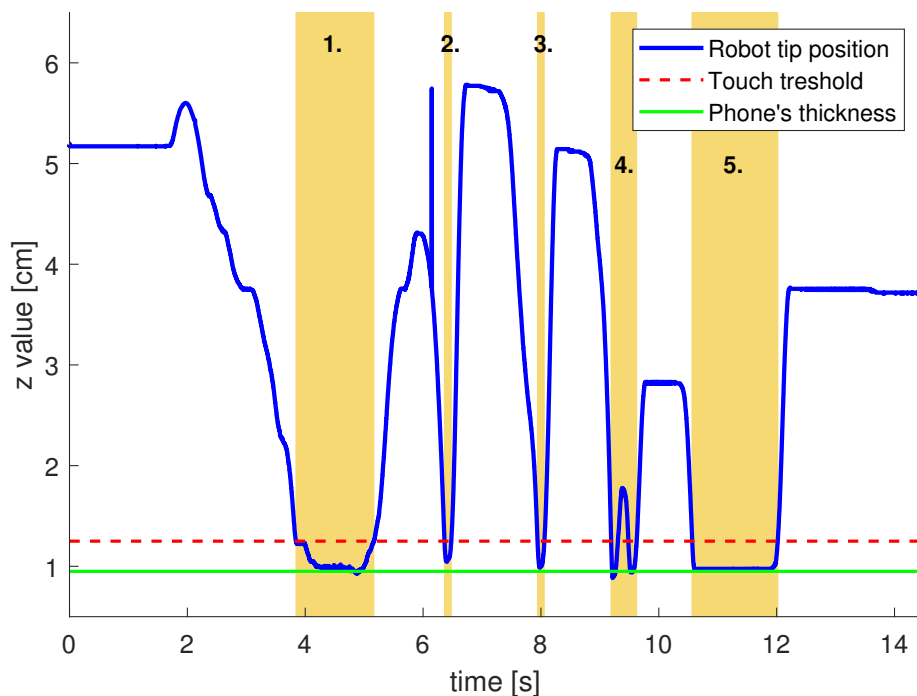
If the point is above a touch screen device, it is checked whether the tip was touching the screen or not. The thickness  $t$  of the device is known, but the compressible ball at the tip will touch the screen before the rigid structure of the arm, triggering a touch in the device but not a collision. To account for that the tip is said to have touched the phone, if the  $z$  coordinate of the position is lower than  $t + 3$  mm.

These two steps separate the trajectory into a sequence of touches. For each touch not only are the start and end positions known, but also the duration. This information is enough to determine what kind of action was performed.

If the touch ended less than 1 cm from where it started, it is some variant of a click at the start position. If this touch is longer than 0.5s, it is a *long click*. If it is shorter than that, it is a *click*. Two clicks less than 0.5s apart are a *double click*.

If the touch ends more than 1 cm away from the start position and lasts longer than 0.2s, it is a *swipe*. The time constraint on swipe had to be used because sometimes there is a motor position measurement error and one point registers as touching the device when it is not. This joint angle error translates into error in at least 2 of the real world axes, classifying it as a touch which started and ended more than 1 cm apart. (The next measurement is correct again making it seem like a fast move.) But because the error lasts only one sample, the time constraint removes these "ghost swipes".

An example of the teaching process with recognition of all of the phone interaction primitives is shown in Figure 6.1.



**Figure 6.1:** Visualization of the teach functionality. The highlighted sections correspond to the following actions: 1. swipe, 2. and 3. click, 4. double click, and 5. long click.

### 6.3 Repetition

After the trajectory is parsed, the program knows what actions at what positions and in what order to perform. To replay the taught sequence on a device, the parsed `Trajectory` object is passed to the `performTrajectory` function of the `phone` object we want to perform the sequence on.

# Results

This thesis serves mostly as a proof of concept for an industrial robot and it is hard to make quantitative assessments before it was deployed. Therefore this section will attempt to qualitatively discuss the successes and deficiencies of our solutions and suggest what might be improved if some of these deficiencies prove to be problematic during long term operation.

## 7.1 Camera calibration

The quality of camera calibration is heavily dependent on how consistent we are in locating the calibration marker. To test that, we let the robot calculate the marker's position and orientation in the camera coordinates 20 times. We repeated the experiment once with the lights turned on (normal light conditions) and once with the lights turned off (low light conditions).

It was assumed the image plane is parallel to the workplane, so only the rotation of the marker about its  $z$  axis (the yaw) is relevant. Results are summarized in Table 7.1.

**Table 7.1:** Measured marker position and orientation.

Value	Normal light		Low light	
x	0.7927	$\pm 0.0044$ cm	0.7908	$\pm 0.0025$ cm
y	-27.4095	$\pm 0.0420$ cm	-27.4468	$\pm 0.0265$ cm
z	51.1869	$\pm 0.1170$ cm	51.3064	$\pm 0.0738$ cm
yaw	0.2663	$\pm 0.0985^\circ$	0.1769	$\pm 0.0606^\circ$

The xyz positions have standard deviations on the scale of tenths of millimeters and do not differ significantly under different light conditions, therefore we can safely assume only very minor errors will be caused by this.

The yaw of the marker differs by  $0.0894^\circ$  depending on the level of light. This small angle difference would manifest as an error of 1.248 mm in y position for an object near the  $x = 80$  cm line.

That is not necessarily negligible and may need to be addressed. As discussed in Section 7.2, the marker has no effect on the phone detection algorithm, therefore a bigger marker may

be used for easier positioning. Also the marker's color can be adjusted to be more distinct in the dark.

## 7.2 Touch screen device detection

To test the touch screen device detection functionality, several touch screen devices and non-device objects were gathered. Four of the devices and four other objects were placed onto the worksurface and the detection algorithm was run. Then the positions and orientations of the tested devices/objects were changed and the detection was run once again. This was repeated 10 times and then 10 times again for the rest of the test material. The phones had different images on their screens, however these images were not changed in between detections. An example of the test setup can be seen in Figure 7.1. The tests were performed under normal light conditions similar to Section 7.1 and the results can be seen in Table 7.2.



**Figure 7.1:** The touch screen device detection test setup. The non-phone objects from left to right are: the dropper, the Kindle, the Apple Watch, the access chip, the passport, and the computer drive.

**Table 7.2:** Results of the device detection test.

Device	True positive	Object	False positive
iPhone 12 Pro	100 %	Box of screwdrivers	0 %
iPhone 13 Mini	100 %	Dropper	10 %
iPhone SE 3	100 %	Kindle	0 %
Motorola Edge 20	100 %	Apple Watch	0 %
Samsung Galaxy A32	100 %	Passport	0 %
Samsung Galaxy M21	100 %	Access chip	0 %
Xiaomi Mi A2	100 %	Computer drive	0 %
Xiaomi Mi A3	100 %	Credit card	0 %

Even though our method is robust and reliable, it still has some limitations, examples of which are shown in Figure 7.2.

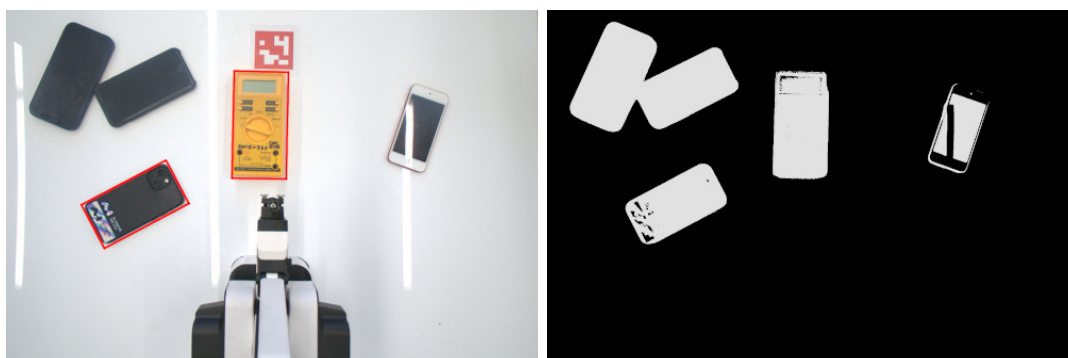
If a device has a color too similar to the white of the worksurface, some pixels do not cross the change threshold and the contour area is too small to pass as a phone, even though its edges were detected.

If two phones are too close together, their contours merge and neither is detected.

In dimmer lighting some phones are detected twice - once just their screen and once their outline.

Some shadows are large enough of a change to be detected and therefore sometimes the phones seem to be larger than they truly are. Even though this could be easily fixed by changing the threshold value, that would put even more restrictions on the color of the device and the contents of its screen as discussed above.

And if one finds an object with similar enough shape to a phone, it is detected as a touch screen device.



(a) The workspace photo.

(b) The corresponding mask.

**Figure 7.2:** An example of erroneous detections. Two phones are too close together and their contours merged. Another phone is upside down yet still detected. The same applies to the yellow multimeter, which should be ignored. And the white iPod touch is too similar to the background.

### 7.3 Arm control

The precision of the arm was measured using the camera and an Aruco marker, which was attached to the top of the arm's gripper. The arm was commanded to move between two positions ( $\vec{x}_1 = (20 \ 20 \ 6)$ ,  $\vec{x}_2 = (45 \ 20 \ 6)$ ) 20 times. When the arm reached one of those positions, the marker's location in the camera coordinate frame was recorded, giving us 40 samples.

Because only the repeatability is important, it does not matter that the positions are in camera coordinates. The standard deviations are written in Table 7.3.

**Table 7.3:** Arm positioning repeatability.

	<b>x</b>	<b>y</b>	<b>z</b>
<b>Standard deviation</b>	$\pm 0.0508 \text{ cm}$	$\pm 0.0278 \text{ cm}$	$\pm 0.0727 \text{ cm}$

Because the results are not significantly larger than the standard deviations of marker detection from Table 7.1, we can say the arm has a repeatability on the scale of tenths of a

millimeter, which is fully sufficient to interact with the control elements of the touch screen devices.

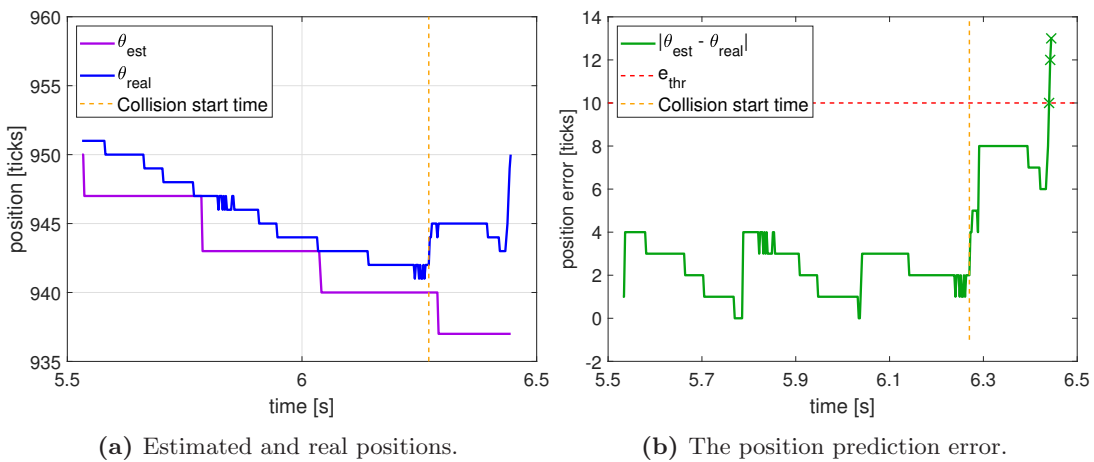
## 7.4 Collision detection

Collision detection is very hard to quantitatively measure. To at least qualitatively test how well our methods perform, we let the arm start moving and then grabbed the gripper to simulate an obstacle. Examples can be seen in Figures 7.3 and 7.4.

The interpolation method works well for collision detection. As can be seen in Figure 7.3 the robot is stopped 0.17s after the collision has occurred and in the two samples necessary to register the collision overshoots the threshold only by 3 ticks.

Unfortunately in this configuration the arm moves only 25 ticks/second, which is too slow to be considered usable. As described in Section 3.2, it is because the motors need to speed up and slow down for each interpolation step and therefore cannot reach their maximum speed.

To fix this we would have to completely rewrite the DexArm’s firmware. Either to allow for custom interpolation without stopping at each point or to utilize this method of collision detection by itself without the help of our external library. Even though this would probably work, it goes against the idea of creating a collaborative robot only being able to continuously read its position, because the big expensive industrial robots like Kuka will not let their users change the firmware.



**Figure 7.3:** Collision detection using the interpolation method. The last three samples are highlighted.

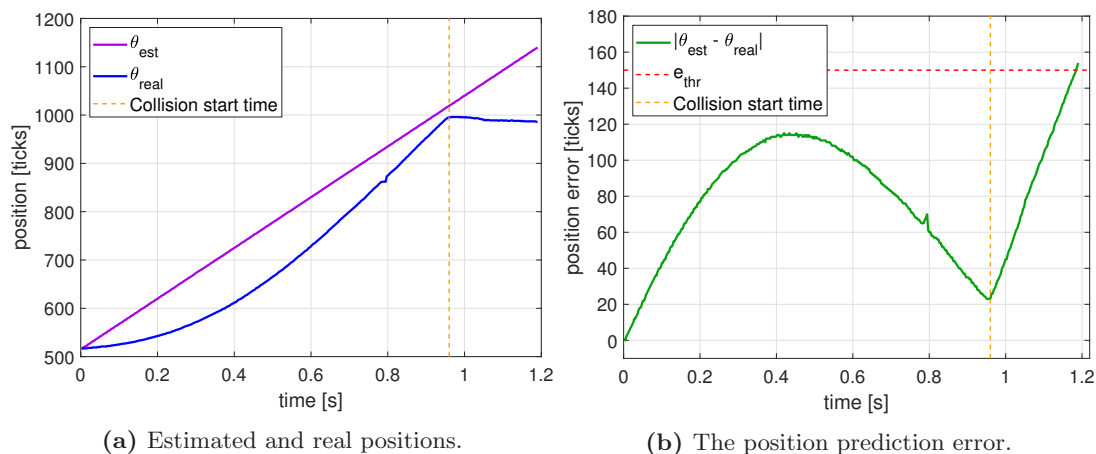
As stated previously, the average speed method did not work. The motor speeds are not constant between movements and therefore it was impossible to accurately predict the movement duration just by averaging the speeds of previous motions.

The idea behind the polynomial regressor approach was based on the observation that the duration of the motion (and therefore its speed) depends on the relative positions of the start and end of the movement. With  $e_{thr} = 150$  the arm stopped 0.23s after the collision started and overshoot 4 ticks (see Figure 7.4).

The stop time is not much larger when compared to the interpolation method, but the other important factor is how much did the arm attempt to move in that time. For interpolation it was 11 ticks -  $0.967^\circ$ , for the polynomial regression method it was 131 ticks -  $11.514^\circ$ . If the

arm was extended to the full 40 cm this would mean the difference between trying to move 0.675 cm and 8.038 cm.

Not only cannot the arm stop quickly, the threshold is not big enough to eliminate false positives.



**Figure 7.4:** Collision detection using the polynomial regression method. The scale is too large to highlight the last three samples.

We have successfully verified the software enabled collision detection behavior. However, as mentioned above, the movement speed of the robot when the collision detection is enabled is too slow for a meaningful deployment.

As the DexArm is small and lightweight, the collaborativity is not safety critical for the deployment and therefore the only critical point remains in the determination of the touch-screen device height which was remedied by adding the option to manually enter the thickness value.

Our attempts at modeling the position curves suggest straight line estimations are not the way to go. The next possible approach would be to attempt to fit the curves using an arctangent. One could also reverse engineer the firmware to obtain the function calculating these curves, but that would again defeat the purpose of smart software collision detection.

## 7.5 Touch screen device control

Because different devices from different manufacturers react slightly differently, it would be very complicated and timeconsuming to quantitatively test the precision of our control motion primitives. Therefore we performed only qualitative testing with the same devices we tested phone detection on. (See Table 7.2.) We told the robot to perform each of the primitives on every one of the devices and checked whether the actions were performed successfully.

The currently discovered pitfalls and limitations are described below.

What is intended to be a *click* gets sometimes registered as a *long click*. That may be because the current tip of the arm is a pen with a compressible ball at the end. When the arm starts moving downward the ball hits the screen and starts to get compressed. Different devices are differently sensitive, some are triggered by the slightest touch, but others require a more significant press. To trigger a touch even with the less sensitive devices, the arm has to go low and the sensitive phones are registering a longer touch than intended.



Another issue of the touch screen device control stems from a modern trend in mobile phone design - cameras sticking out of the back of the device. These cause uneven thickness and rocking when an unsupported part of the screen is touched. Some phone cases solve this issue by being thick enough to make the camera lie flat with the back of the case. The simplest solution to this problem would be to create thick trays with a cutout for the camera.

## 7.6 Teach and repeat

The teach part of the *teach and repeat* mode works reliably, as long as the operator makes the desired primitives stand out. Meaning if, for example, a *click* is wanted, one must decisively bring the robot's tip down and up again quickly and then wait for half a second to register a *click* and not a *double click*. It is important to note that a double click will be recorded even if the second click is at a completely different location on the screen from the first one. This could be solved by adding a more sophisticated and robust parser function.

The repeat part was found to be working perfectly by itself. It's only issues stem from the imperfections of the motion primitives as discussed in Section 7.5.

## Conclusion and possible future work

A robot capable of controlling small touch screen devices was created. It is capable of detecting the mobile phones in an image of its workspace and then use camera calibration matrices to accurately determine their real world locations with respect to the robot arm. The robot can use its arm to click, long click, double click and swipe the screens of the phones. The same action at the same place can be performed on several different devices, as the robot is capable of remapping the action positions between them. When a sequence of actions is performed by the robot's operator on a device using the arm, the code is capable of identifying said actions and reproducing them on the other devices.

The only feature which was not implemented completely successfully is the collaborativity. The successes we had were unusable because either the movement was too slow or the detection method too unreliable. However it was known from the beginning this is not going to be easy. We learned what works, what doesn't and why, helping us improve our future attempts.

This thesis focused on the inner workings of the robot, these will be connected to a frontend written by Ing. Čížek and Ing.Karel Frajták. The whole device will be shown to our partners at Škoda Auto.

Continuation of this work depends mainly on the feedback we receive from the users. There may be need to address some of the imperfections described in the results section. We will definitely continue our attempts at adding collaborativity to the DexArm.

Currently the robot can only interact with what the operator tells it to. The logical next steps in the development of this project are to add ways to detect what is happening on the device's screen as a way of understanding the semantics of the phone operation. For the purpose of device setup and configuration, it would be useful to detect when the contents of the screen change. Then there are of course the different elements with which the robot can interact, like buttons, textfields and their contents, which can be identified.

The work on this has already been started when in collaboration with Josef Zelinka we trained a convolutional network capable of detecting the individual keys of a virtual keyboard. However this functionality is not yet integrated into the machine.

Even though the current state of the robot can be considered a success, it is still a proof of concept with its known and unknown flaws. What improvements and fixes will be done is heavily dependent on how the robot will behave when deployed.



# Bibliography

- Bradski, G. (2000). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*.
- Davies, E. R. (2012). *Computer and machine vision: theory, algorithms, practicalities*. Academic Press, Academic Press, 2 edition.
- Faigl, J. and Čížek, P. (2019). Adaptive locomotion control of hexapod walking robot for traversing rough terrains with position feedback only. *Robotics and Autonomous Systems*, 116:136–147.
- Guennebaud, G., Jacob, B., et al. (2010). Eigen v3. <http://eigen.tuxfamily.org>.
- Haddadin, S., Luca, A. D., and Albu-Schaffer, A. (2017). Robot collisions: A survey on detection, isolation, and identification. *IEEE Transactions on Robotics*, 33(6):1292–1312.
- Rotrics (2022). Dexarm specs. <https://rotrics.com/pages/dexarm-spec>. Accessed: 16. 4. 2022.
- Siciliano, B., Sciavicco, L., Villani, L., and Oriolo, G. (2009). *Robotics: Modelling, Planning and Control*. Advanced Textbooks in Control and Signal Processing, Springer London.
- Zalm, E. (2011). Marlin firmware. [www.marlinfw.org](http://www.marlinfw.org). Accessed: 1. 4. 2022.
- Zhang, Z. (2000). A flexible new technique for camera calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(11):1330–1334.

