Diploma thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

TIAGo++: a robotic archer

Marek Jalůvka

Supervisor: RNDr. Miroslav Kulich, Ph.D.
May 2021

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Jalůvka Marek**	Personal ID number:	**466227**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Control Engineering**		
Study program:	**Cybernetics and Robotics**		
Branch of study:	**Cybernetics and Robotics**		

II. Master’s thesis details

Master’s thesis title in English:

TIAGo++: a robotic archer

Master’s thesis title in Czech:

TIAGo++: robotický lukostřelec

Guidelines:

1. Get acquainted with a two-armed mobile robot Tiago++ and tutorials for its usage in ROS.
2. Get acquainted with a simple bow model as described in [1].
3. Design and realize a gripper to hold and manipulate a bow, a string, and an arrow.
4. Design and realize a control algorithm for shooting a bow.
5. Employ and tailor some object detection algorithm to aim and shoot at the target.
6. Experimentally evaluate the realized algorithms and discuss achieved results.

Bibliography / sources:

[1] C. N. Nickman, F. Nagler, P. E. Klopsteg (eds) Archery: The Technical Side, National Field Archery Association; 1st edition, 1947
[2] TIAGo++ Tutorials, online, http://wiki.ros.org/action/info/Robots/TIAGo%2B%2B/Tutorials
[3] J. Redmon, A. Farhadi. Yolov3: An incremental improvement, arXiv preprint ArXiv:1804.02767, 2018

Name and workplace of master’s thesis supervisor:

RNDr. Miroslav Kulich, Ph.D., Intelligent and Mobile Robotics, CIIRC

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **29.01.2021** Deadline for master's thesis submission: **21.05.2021**

Assignment valid until:
by the end of summer semester 2021/2022

RNDr. Miroslav Kulich, Ph.D.	prof. Ing. Michael Šebek, DrSc.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the master’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 master’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt	Student’s signature

Acknowledgements

I would like to thank RNDr. Miroslav Kulich, Ph.D. for great guidance, valuable advice, and revision of this thesis. I am also grateful to Dr. Gaël Écorchard for the design and manufacturing of the bow holder and bowstring rings for the robot. Many thanks to Ing. Viktor Kozák for the training of a neural network target detector used as one of the detectors for this thesis. I would like to also thank Ing. Karel Košnar, Ph.D. for practical suggestions. Finally, I would like to thank my family for unwavering support not only during the writing of this thesis.

Declaration

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

Prague, May 21, 2021

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 21. května 2021

Abstract

This thesis describes a TIAGo++ robotic archer capable of shooting at a target detected by its camera. Closer specifications of the solved problem are first defined along with description of the robot and bow and arrows used. Next, the problem is decomposed into necessary subtasks - mathematical bow model adaption and simulation, design of suitable bow drawing trajectories for the robot (low torques required by robot motors), target detector development, and the process combining the previous subtasks into a control loop serving to detect, aim and shoot at the target by the robot. All of these subtasks are described in detail theoretically at first, then their implementation on the real robot is addressed, and finally, the crucial parts of the system - the bow drawing trajectories, target detector and the target shooting process - are evaluated.

Keywords: archer, TIAGo++, robot, target detection, ROS

Supervisor: RNDr. Miroslav Kulich, Ph.D.

Abstrakt

Tato práce popisuje TIAGo++ robotického lukostřelce, který je schopný střílet na terč detekovaný ve své kameře. Nejprve jsou definovány bližší specifikace řešeného problému spolu s popisem použitého robota a luku s šípy. Dále je problém dekomponován na potřebné podúlohy - převzetí a simulace matematického modelu luku, design vhodných trajektorií natahování luku pro robota (s nízkými požadavky na točivé momenty od motorů robota), vypracování detektoru terče a proces kombinující předchozí podúlohy do řídící smyčky, která slouží k detekci, zamíření a výstřelu na terč robotem. Všechny tyto podúlohy jsou detailně popsány nejprve teoreticky, poté je popsána jejich implementace na reálném robotovi a nakonec jsou stěžejní části systému - trajektorie pro natahování luku, detektor terče a proces střílení na terč - vyhodnoceny.

Klíčová slova: lukostřelba, TIAGo++, robot, detekce terče, ROS

Překlad názvu: TIAGo++: robotický lukostřelec

Contents

Figures

Tables

Chapter 1

Introduction

As the focus of modern robotics shifts from single-task industrial assembly line robots to mobile general purpose robots, there is a need to operate in various environments not designed with a robotic agent in mind, but rather tailored for humans. All our tools and infrastructure are designed with the assumption of a human using them. One branch of robots successfully operating in such environments are humanoid robots constructed to mimic human biomechanics. One of the crucial and very advantageous features of the human form are two arms for their dexterous and efficient capabilities of manipulating various objects.

Many everyday tasks require two-hand manipulation: opening various packages, unscrewing bottle lids, washing dishes, or cutting paper with scissors, just to name a few. Many tools are also designed to be wielded with two hands, e.g., a broom, shovel, wheelbarrow, or a chainsaw. Complex movement and coordination between both arms is usually necessary for these tools, as there are often only a few ways of using that particular tool properly, i.e., to serve its purpose and avoid straining its operator excessively. A perfect example of such a tool is the bow. Its main design goal is to convert as much of the physical strength of the archer as possible to the kinetic energy of the arrow. The movement of drawing and shooting bow has been perfected by humans over thousands of years as its efficiency and accuracy were pivotal for survival. Letting a robot draw and shoot a bow close to this efficiency or accuracy is a challenging, but achievable task. Shooting a bow requires strength and precision rather than speed, which makes it a convenient way for a lot of robots to potentially launch a projectile since most of the contemporary robots are not able to generate safely enough end-effector acceleration to successfully throw an object.

As a result of these properties, drawing and shooting a bow with a dual-arm robot is chosen as the main focus of this thesis.

1.1 Goal of this thesis

This thesis aims to develop and describe a robotic archer with TIAGo++ dual-arm robot. This entails the design of a physical holder, attaching the bow to the robot, designing low-torque bow drawing trajectories, creating a

system of visual detection and localization of the target and finally developing a control loop to shoot an arrow at the target.
The most emphasis is placed on the design of the bow drawing trajectories as they are the foundation of the whole system and their improper design would not only decrease the accuracy as is the case for the rest of the subtasks, but also could render the whole shooting process anywhere from energy inefficient up to completely impossible due to the torques required from the robot motors being beyond their capabilities.
The designing process of these trajectories needs to be general enough to be after modifications potentially applied to different physically demanding tasks involving complex arm movements.
As a secondary goal, the whole system needs to be as accurate as possible, which is a product accuracies of each component - the physical bow and its mathematical model, localization of the target and the shooting control loop itself.

1.2 Contribution

The main contributions of this thesis are:

- creating a simulation of a bow using a mathematical model fitted to real bow parameters
- devising an optimization approach to the design of bow drawing trajectories
- developing a method of reliable detection and localization of a target
- composing the detection and localization of the target and drawing trajectories into the detect-aim-shoot loop
- evaluating the suitability of the drawing trajectories and the accuracy of both the target localization by itself and the whole detect-aim-shoot loop.

Chapter 2
Related work

Certain component tasks of the solution of this thesis are by themselves frequent topics in the literature.
For example, an optimization-based framework to design optimal trajectories for a robotic system with respect to an arbitrary physical criterion is derived by [BMK01]. The optimization criterion is formulated such that it has analytically computable gradients enabling a reliable and efficient solution to the optimization problem. This framework is demonstrated on several tasks for a humanoid robot model like power lifting, diving, and gymnastics.
The paper by [CF17] explores specifically torque efficient robot motions with relation to the singularities of the kinematic chain. Singularities are in robotics configurations of the robot, where the end-effector looses the ability to move in one or more directions. These are usually avoided, but as shown in [CF17] the motions at or near singularities are optimally torque and energy efficient.
The archery target detection is also solved in various forms, usually as a part of an automatic scoring system for archery. For example, in [JD20], the target is detected by first filtering out the background of the target face (the whole square encompassing the concentric circles inside the target), then detecting its corners, followed by a perspective transformation and finally a Hough transformation to get the biggest circle in the target.
The paper by [DL19], on the other hand, aims to develop an efficient FPGA implementation of target detection with a robotic archer in mind. The target center is obtained by first using color segmentation in HSV color space to classify each pixel as target yellow, red, blue colors, or none of these. The target center is obtained by an algorithm trying to recognize a color sequence matching the standard target sequence of blue-red-yellow.
The closest work from the perspective of its goal to this thesis is currently a paper by [PK10], which presents a humanoid robot iCub learning the skill of archery by a reinforcement learning approach - the robot shoots at the target multiple times with different aims and draw lengths and evaluates each shot by visually detecting the arrow hit in relation to the target. This evaluation drives the learning process to progressively hit closer to the target center. Two algorithms for the learning process are compared: the expectation-maximization-based PoWER algorithm and a custom local regression algorithm called ARCHER. To find the target in the image, a

dataset of pixels in the target and in the arrow tip is collected manually by an operator as an initial calibration. These pixels are converted to the YUV color space and their U and V components (chrominance) are used to learn a Gaussian Mixture Model (GMM) of the color characteristics of the target and arrow tip in UV space. Parameters of these Gaussian distributions are later during the learning process used for position estimation of the target and arrow tip in the image.

Chapter 3

Problem definition

In this thesis, we are tackling the problem of designing a robotic archer with the capability of detecting and shooting at a target. The core components we are provided to achieve this are described in this chapter. Namely, we are using a TIAGo++ two-handed robot as the archer and a recurve bow with low draw force to shoot a suction cup arrow at the target. The bow is mounted on the robot and the arrow is nocked on the bowstring ready to shoot. The target is located in a close vicinity of the robot - at the distance of 3.5 to 5.5 m.

3.1 TIAGo++ robot

TIAGo++ robot by PAL Robotics [pala] was used as the robotic archer. The whole robot is in Figure 3.1. Its components pivotal for this thesis will be discussed in this section.

3.1.1 Arms

Both arms are 7 DoF (degrees of freedom) serial manipulators with 7 rotational joints. In other words, the position of each arm can be described by 7 parameters - rotation angles around the rotation axes in each joint. For a general serial manipulator, these angles are denoted as θ_1, θ_2, ... θ_n, where n is the number of joints. Put in a vector they create the joint configuration:

$$\boldsymbol{\theta} = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \dots \\ \theta_n \end{bmatrix}. \tag{3.1}$$

For the purposes of this thesis, we will distinguish between the joint configurations of each arm by the additional lower index l for left or r for the right arm:

$$\boldsymbol{\theta_l} = \begin{bmatrix} \theta_{l1} \\ \theta_{l2} \\ \dots \\ \theta_{l7} \end{bmatrix}, \quad \boldsymbol{\theta_r} = \begin{bmatrix} \theta_{r1} \\ \theta_{r2} \\ \dots \\ \theta_{r7} \end{bmatrix}. \tag{3.2}$$

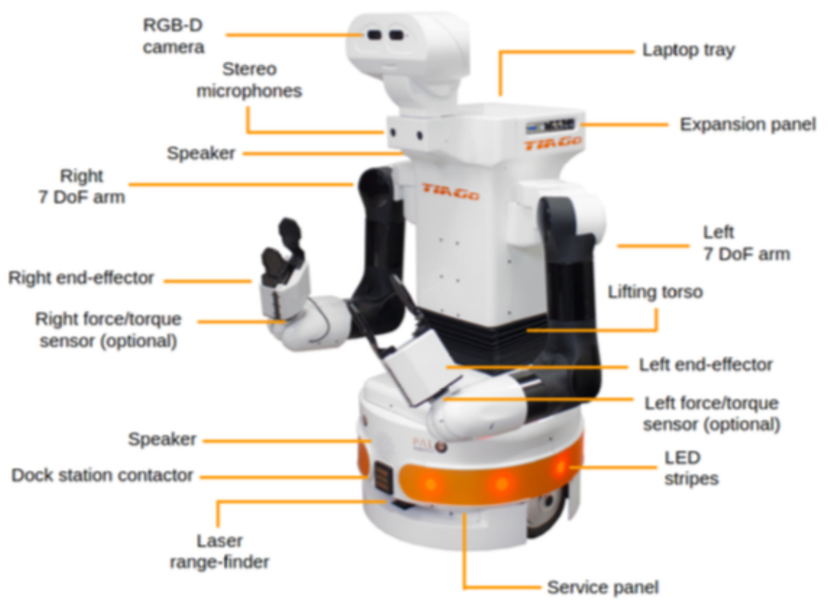

Figure 3.1: PAL Robotics TIAGo++ robot components [PALb]

Given these joint configurations and a robot model (the relative position and orientation of each joint in the kinematic chain of each arm), we are able to compute *forward kinematics* of the arms, i.e., determine the position and orientation of both end-effectors. An end-effector in robotics is the last link in a kinematic chain usually used to effect the environment. The first link in the kinematic chain, on the other hand, is generally called the base link. This is usually an object the manipulator is mounted on. The base link of the TIAGo++ robot arms (and torso) will be the mobile base with its coordinate frame called `base_footprint` and positioned as depicted in Figure 3.2.
Often, an opposite task to forward kinematics is performed called *inverse kinematics*. The input to this task is a desired position and orientation of the end-effector, often compactly given as a transformation matrix from the end-effector coordinate frame to the base link coordinate frame T_e^0, and the output is one or more possible joint configurations to reach this pose. For both TIAGo++ arms, the inverse kinematic task will be computed always for specific frames called tool link frames. They are positioned in the wrists and depicted in Figure 3.2 as `arm_left_tool_link` and `arm_right_tool_link`.

The arms are a mirror image of each other with respect to the kinematics. Regarding other properties like dynamics and motor specifications, they are identical. An image of one of these arms is in Figure 3.3. The only specifications of the arm motors relevant for this thesis are summarized in Table 3.1.

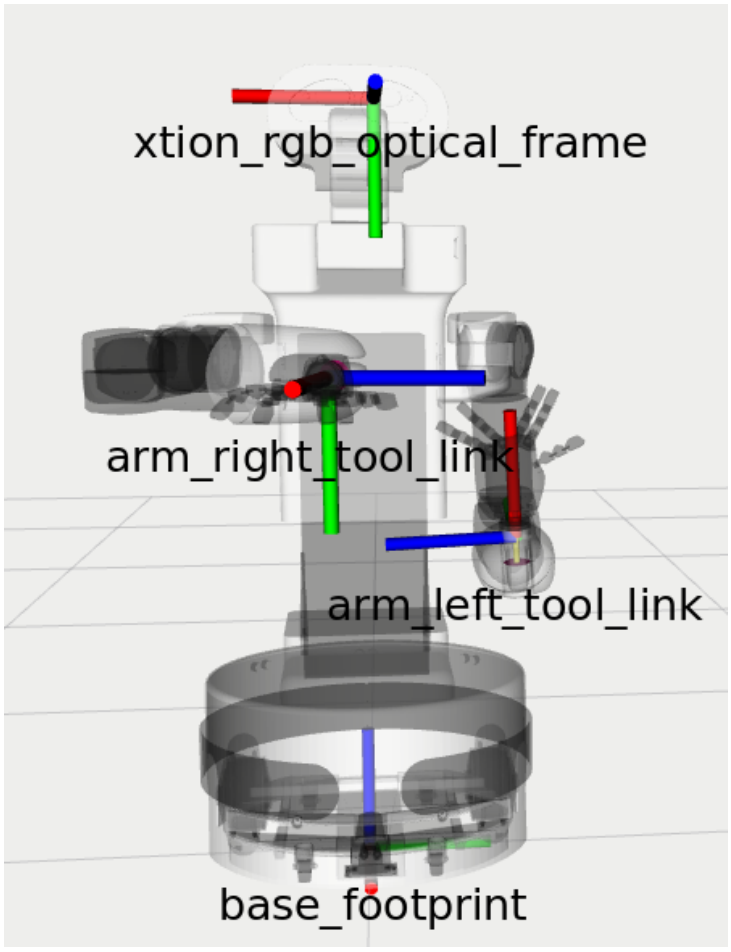

Figure 3.2: TIAGo++ robot in RViz - important coordinate frames

Actuator	Gear transmission ratio	Nominal torque [Nm]
1st module	100:1	39
2nd module	100:1	39
3rd module	100:1	22
4th module	100:1	22
Wrist 1st DoF	441:1	3
Wrist 2nd DoF	441:1	5
Wrist 3rd DoF	441:1	5

Table 3.1: Motor specifications [PALb]

The joints are not fitted with torque sensors, but provide information about currents through the motors. This will be later used to roughly estimate the motor torques.

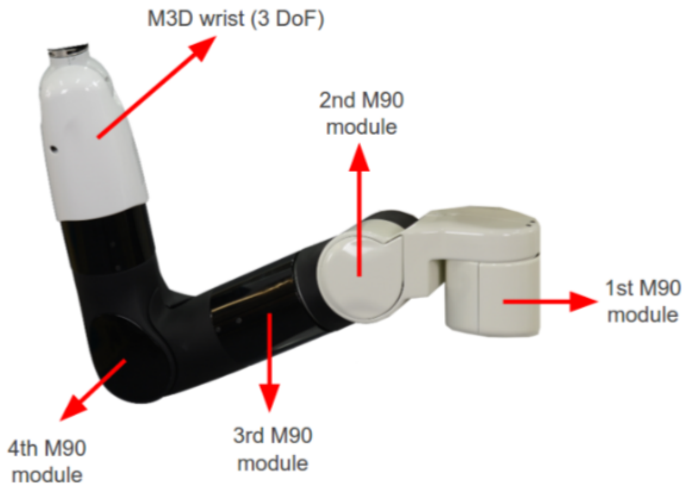

Figure 3.3: TIAGo++ robot arm [PALb]

Torso lifter

The robot torso is mounted on a prismatic joint with a range of 35 cm. The height of the whole robot can be anywhere from 110 to 145 cm.

3.1.2 Hey5 Hand End-Effector

The Hey5 Hand end-effector was used as the drawing left hand. It is illustrated in Figure 3.4. The fingers are controlled by 3 motors - one for the thumb, one for the index finger and the last for the remaining fingers, which are represented by one joint called "mrl" (middle ring little).

3.1.3 Head

TIAGo++'s head is mounted on a 2 DoF pan-tilt mechanism. It is equipped with a RGB-D Orbbec Astra S camera. The depth sensor unfortunately has a range of 0.4 - 2 m, which renders it not usable for this thesis. The image mode used is 640x480 resolution with 30 fps.
The frame the target detection will be later performed in is depicted in Figure 3.2 as `xtion_rgb_optical_frame` and will be from here on referred to as the camera frame.

3.1.4 Mobile base

The mobile base has a differential drive, laser-range finder and three rear sonars. These enable robot localization and navigation with obstacle avoidance.

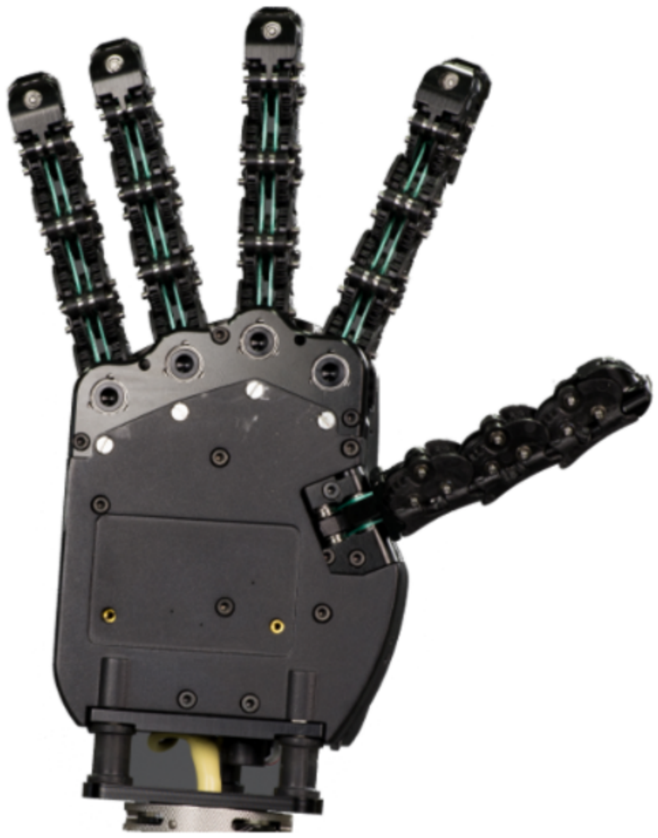

Figure 3.4: Hey5 Hand [PALb]

3.2 Bow and arrows

The bow and arrows used for the realization of this thesis were from the Geologic's Softarchery 100 set [bow] illustrated in Figure 3.5. The reason behind that was the full draw force of 20 lbf or 89 N (as provided by the manufacturer) was within the robot's capabilities and the suction cup arrows excluded the possibility of a serious injury after a potential accident.

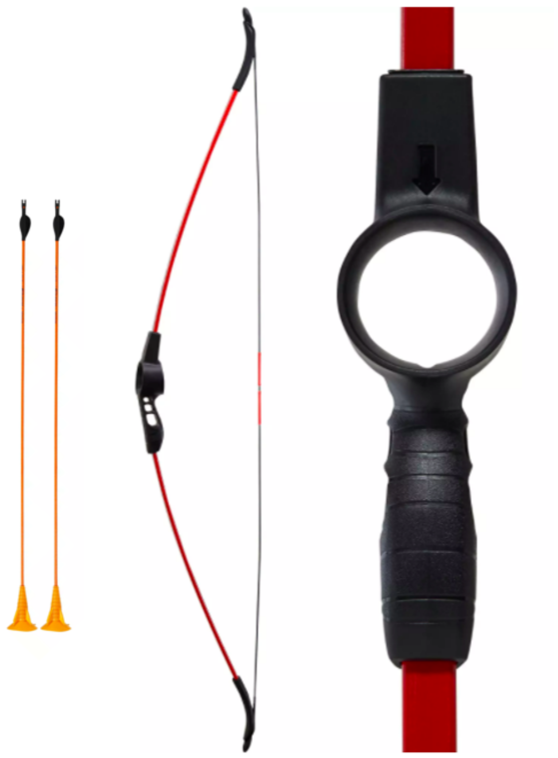

Figure 3.5: Geologic Softarchery 100 bow and arrows [bow]

Chapter 4

Approach

4.1 Robotic archer overview

The task of realizing a robotic archer capable of recognizing a target and shooting at it consists of several subtasks.
First, a mathematical model of the bow needs to be adopted to serve as a simulation tool.
Next, it is needed for the robot to be able to handle the bow accurately and predictably. This was achieved by a custom 3D printed end-effector designed to mount the bow on the robot.
Then, some bow-drawing trajectories need to be designed: multiple trajectories with different bow elevation (pitch) angles to provide more options to choose from for a given robot position relative to the target. These trajectories need to put as little strain as possible on the robot motors - i.e., the torques need to be minimized. The reason for this is that a bad draw trajectory can give a high load on small motors (like a wrist motor) not designed for such a load. This task is divided into two parts:

- finding optimal draw configurations - for a known final draw length, finding possible arm configurations, ensuring proper torque distribution across all motors
- design a low-torque trajectory to get to a given final draw configuration.

Subsequently, the target needs to be detected and localized. A camera inside the robot's head was used for this purpose. Two methods are used to find the target in the image: an object detection neural network trained on the target and a concentric circles detector.
Finally, the detection and bow drawing need to be composed into a detect-aim-shoot loop.
More detailed description of these components follows.

4.2 Bow model

For the purposes of planning robot movements suitable for drawing and shooting a bow, it was necessary to create first a mathematical model of a

bow for the robot model to interact with. A lot of such models have been developed to this day, but for our purposes, only very simplified model relating draw length to the necessary draw force would suffice. Such model has been adopted from [CNH47].
The model assumes a bow consisting of a middle rigid part and two flexible limbs (upper and lower). The tips of both of these limbs are assumed to move on trajectories in the shape of arcs of a circle. This assumption is reasonable as practically all bowyers construct the bows to bend like this (this property results in all sections of the bow stressed equally when drawn). A bow model with all its describing parameters is displayed in the Figure 4.1.
The meaning of individual parameters, which are used throughout this thesis, as well as important relations with other parameters are summarized in the Table 4.1. All these parameters express the kinematic properties of the bow given its geometry (various lengths and angles). The only exception is the bow constant C which relates angle A of limb deflection and static force at bow tips f. This constant is bow specific and is influenced by bow dimensions and material.
These relations can readily provide the draw length and all forces and tension given the angles A and E (and bow dimensions). Unfortunately, the most important dependency for bow drawing is the static draw force as a function of draw length $F(D)$ which is not so easily obtained from the expressions. However, it can be still derived from the diagram in Figure 4.1. Using the cosine formula and the Pythagorean theorem:

$$\begin{aligned} Q &= \sqrt{D^2 + (\frac{B_1}{4} + L)^2} \\ G &= \arccos\left[\frac{2}{3B_1Q}(\frac{9}{16}B_1^2 + Q^2 - S^2)\right] \\ M &= \arccos\left[\frac{2}{3B_1S}(\frac{9}{16}B_1^2 + S^2 - Q^2)\right] \\ K &= \arccos\frac{4L + B_1}{4Q}. \end{aligned} \tag{4.1}$$

The angles A and E are then:

$$\begin{aligned} A &= \pi - G - K \\ E &= M - A. \end{aligned} \tag{4.2}$$

This is enough to obtain the dependency $F(D)$ by the respective substitutions.

The dependency $F(D)$ is important because it also can be directly used to compute the potential energy stored in the bow for a particular draw length:

$$E_p = \int_{H_0}^{D} F(l)dl. \tag{4.3}$$

parameter	parameter meaning	relation to other parameters
B	half of bow length	-
L	half of the length of the rigid handle	-
B_1	length of active bending portion of each limb	$B_1 = B - L$
S	half of the length of the string	-
H	distance from middle of bow to middle of line connecting bow tips	$H = \frac{3}{4}B_1 \sin A$
H_0	bow to string distance for undrawn bow	-
P	distance from arrow nock to middle of line connecting bow tips	$P = S \sin E$
D	draw length	$D = H + P$
Y	half of the length of line connecting bow tips	$Y = \frac{3}{4}B_1 \cos A + \frac{B_1}{4} + L$
O	center of circular motion of bow tip	located at distance $\frac{3}{4}B_1$ from undeflected bow tip
A	angle between the line connecting bow tip and point O and the undeflected limb	-
E	angle made by string with the line connecting the bow tips	-
f	static force at each bow tip in a direction tangent to its path	$f = C \cdot A$
T	static tension in the string	$T = \frac{f}{\sin(A+E)}$
F	static draw force	$F = 2T \sin E$
N	distance along the path made by the bow tip during the draw	-
C	bow constant	-

Table 4.1: Bow parameters

This potential energy is during the shot converted to the kinetic energy of the arrow, but also to vibrations of the string and limbs, noise and elastic energy of the arrow. Bow efficiency describes the portion of the potential energy actually converted to kinetic energy:

$$E_k = \eta E_p, \tag{4.4}$$

where η is efficiency, E_k is kinetic energy of the arrow and E_p is the potential

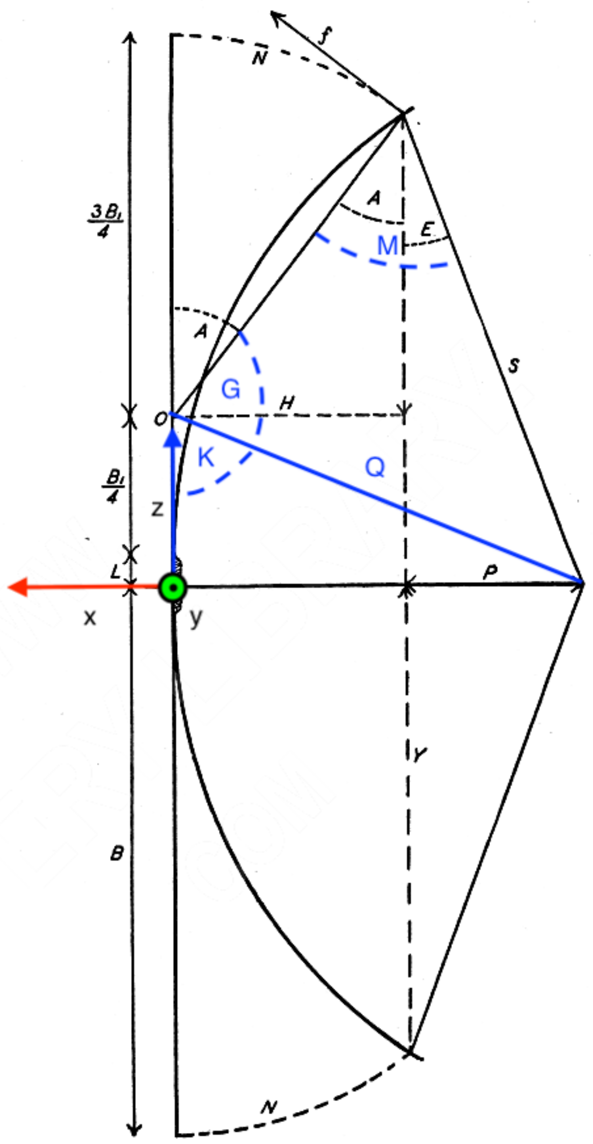

Figure 4.1: Bow model diagram [CNH47], edited

energy stored in the bow. The velocity of the arrow on departure from the bow is then:

$$v = \sqrt{\frac{2\eta E_p}{m}}, \tag{4.5}$$

where m is mass of the arrow.

4.3 Search for optimal bow draw configurations

The process of finding optimal draw configurations for a given draw length will be described here. This process was employed only to find the final draw configurations, not the intermediary points during drawing under the

assumption that the final draw configurations require the biggest force and therefore torques on motors. Also, the robot is expected to hold the bow in the final configuration for the longest time during aiming.
The following paragraphs will describe in detail the criterion of optimality for the bow draw configurations - its computation and the reasoning behind it. Using this criterion, an optimization task will be formulated to find an optimal bow draw configuration. Finally, a method to solve this task will be stated.
As was previously mentioned, it was crucial for the drawing motion to require as little torque as possible from robot motors and the torque needed to be distributed properly between the motors, i.e., each motor needed to operate only within its nominal torque range. To better characterise the torque load on a particular motor in relation to its capabilities, normalized torques are used throughout this thesis:

$$\tau_{s,i}^{n} = \frac{\tau_{s,i}}{n_i}, \quad s = \{l, r\}, \quad i = 1...7, \tag{4.6}$$

where $\tau_{s,i}$ is the torque exerted on a motor number i of arm s (left or right), n_i is nominal torque for motor i (motors with the same number are identical on both arms) and $\tau_{s,i}^{n}$ is the normalized torque.

The torques of the motors required for the robot's end-effector to exert some force and moment on the environment can be generally computed like in [Asa05]:

$$\boldsymbol{\tau} = \boldsymbol{J}(\boldsymbol{\theta})^T \boldsymbol{F}, \quad \boldsymbol{\theta} = \begin{bmatrix} \theta_1 \\ \theta_2 \\ ... \\ \theta_n \end{bmatrix}, \quad \boldsymbol{\tau} = \begin{bmatrix} \tau_1 \\ \tau_2 \\ ... \\ \tau_n \end{bmatrix}, \quad \boldsymbol{F} = \begin{bmatrix} \boldsymbol{f_{n,n+1}} \\ \boldsymbol{N_{n,n+1}} \end{bmatrix} \tag{4.7}$$

where $\boldsymbol{\tau}$ is a vector of torques of each joint in an open kinematic chain, $\boldsymbol{J}$ is the Jacobian matrix computed for some joint configuration $\boldsymbol{\theta}$ and $\boldsymbol{F}$ composes of a 3x1 vector of end-effector force and a 3x1 vector of end-effector moment on the environment.
For the purposes of this task, I will model not only the motor torques required to counteract the forces of the draw, but also the torques counteracting the weight force of each arm link as these are significant, even dominant over draw forces for certain draw configurations. To compute such torques, which are the results of forces acting on a different point in the kinematic chain other than the end-effector - center of mass of that particular link - it is necessary to essentially compute the Jacobian as if the chain had a virtual end-effector in that point. Such a Jacobian computed for some joint configuration $\boldsymbol{\theta}$ at a point $\boldsymbol{p}$ from the kinematic chain expressed in the base link coordinate frame will be here denoted as $\boldsymbol{J}(\boldsymbol{\theta}, \boldsymbol{p})$. The torques needed by the motors of one robotic arm with n different links (all links counted including the finger links or the bow itself) to counteract the drawing force and gravity of the links are

then:

$$\boldsymbol{\tau}_s = \boldsymbol{J}(\boldsymbol{\theta}_s, \boldsymbol{e}_s)^T \begin{bmatrix} \boldsymbol{f}_s \\ \mathbf{0} \end{bmatrix} + \sum_{i=1}^{i=n} \boldsymbol{J}(\boldsymbol{\theta}_s, \boldsymbol{cm}_{s,i})^T \begin{bmatrix} m_{s,i}\boldsymbol{g} \\ \mathbf{0} \end{bmatrix}, \quad s = \{l, r\}, \tag{4.8}$$

where $\boldsymbol{e}_s$ is the end-effector point for the arm, $\boldsymbol{f}_s$ is a 3x1 vector of the drawing force expressed in the base link frame, $\boldsymbol{cm}_{s,i}$ is the center of mass of i-th link on the arm and $m_{s,i}$ is its mass. Finally

$$\boldsymbol{g} = \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} \tag{4.9}$$

with g being gravitational acceleration. The vector is pointing up because that is the direction the motors apply force to the links to counteract gravity. Furthermore, the relation $\boldsymbol{f}_r = -\boldsymbol{f}_l$ holds in the base link frame.
Now that we have a way to compute the joint torques given a particular joint configuration, we must find possible joint configurations. The robot has a bow mounted in its right hand and draws it with its left hand. The bow pose coordinate frame will be fixed to the bow and positioned as shown in the Figure 4.1. Let us have the bow pose in the right hand as a variable and fix some draw length. The orientation of the left hand should match that of the bow pose (to avoid large torques on the wrist) and its position will be determined by the draw and some fixed offsets. Therefore, the left hand pose will depend only on the bow pose which will be the only variable.
As already mentioned in the previous chapter, the links, we can readily compute the inverse kinematics for will be called tool links here, more specifically the left tool link and the right tool link for both arms. The right tool link pose is obtained from the bow pose according to the static transformation between the two links defined by the physical dimensions of the handle mounting bow on the robot. The pose of the left tool link is obtained as a transformation from the bow pose to the left hand grasping frame located on the left hand palm near the fingers and then by a static transformation to the left tool link. The transformation matrix of the left hand grasping frame to the base link frame is:

$$T^0_{lhgf} = T^0_{bow} \cdot \begin{bmatrix} 1 & 0 & 0 & -D + x_{off} \\ 0 & 1 & 0 & y_{off} \\ 0 & 0 & 1 & -z_{off} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.10}$$

where D is the draw length, x_{off} is an offset in the x direction, y_{off} in the y direction and z_{off} in the negative z direction. Some z_{off} is always necessary so that the left hand is positioned below the arrow on the string. The Figure 4.2 is provided to better illustrate the coordinate frames involved.

Upon reception of the necessary poses of both tool links, all possible (or at least most) joint configurations able to reach these poses were obtained from the inverse kinematics. Subsequently, an optimal configuration was picked

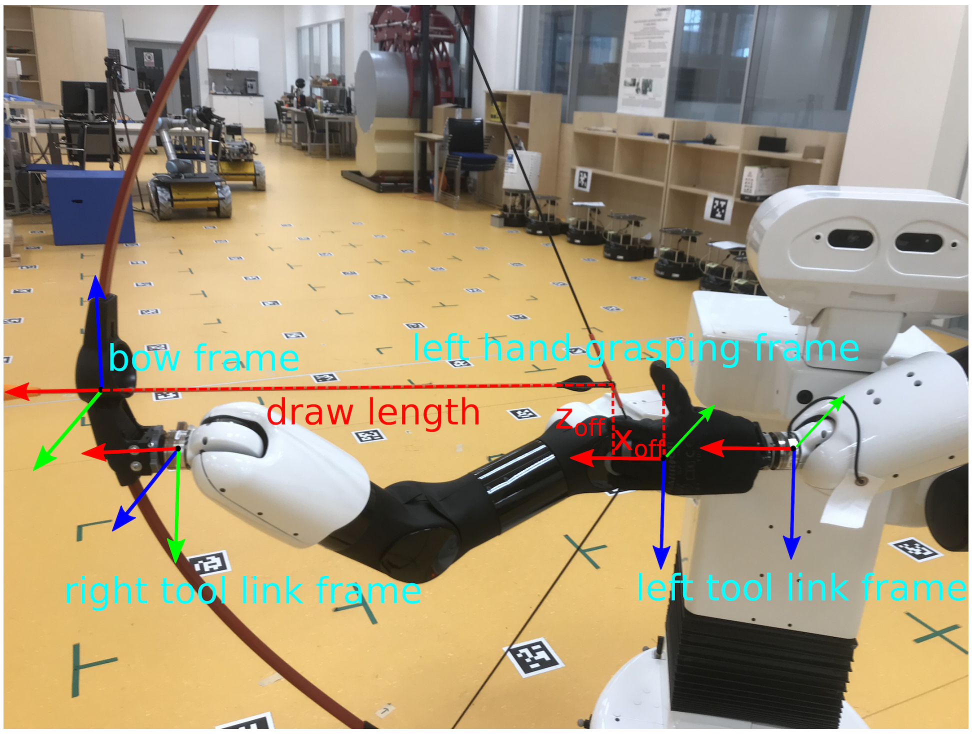

Figure 4.2: Draw configuration frames

from these joint configurations for each arm independently. The criterion for this selection was a minimal Euclidean norm of the normalized torques for that configuration:

$$norm_s = ||\boldsymbol{\tau}_s^n||, \quad s = \{l, r\}, \tag{4.11}$$

where $\boldsymbol{\tau}_s^n$ is the vector of normalized torques computed from $\boldsymbol{\tau}_s$ according to the Equation 4.6, which in turn was computed with the particular joint configuration $\boldsymbol{\theta}$ according to the Equation 4.8. The load on the robot motors for a particular draw configuration can then be defined as:

$$load = norm_l^* + norm_r^*, \tag{4.12}$$

where $norm_l^*$ and $norm_r^*$ are the norms from Equation 4.11 for the optimal joint configurations.
A pseudocode for a function `torquesNormSum`, which takes the bow pose in the form of x, y, z position and Euler angle representation of orientation (yaw (ψ)–>pitch (α)–>roll (ϕ)) and returns the value of *load*, is in Algorithm 1. Most component functions and variables of this algorithm have been described in the previous paragraphs. Concerning the rest: a variable rm stands for robot model and is a data structure holding all the information about robot kinematics. More specifically, the robot physical model is kept here. Individual joints of the robot are accessed via their respective joint group - $rm.left$ and $rm.right$ for arms and $rm.torso$ for the torso. The function `setJointPositions` sets joint configuration of some joint group in a robot model variable to its input argument. The function `isInSelfCollision`

unsurprisingly checks for self-collision of the robot model after its state has been modified. Finally, the variables $\boldsymbol{\theta}_{lcf}$ and $\boldsymbol{\theta}_{rcf}$ are some joint configurations chosen such that the left or right arm, respectively, is collision-free with the rest of the robot.
The function `torquesNormSum` was used as a minimization criterion in the search for optimal draw configurations. Furthermore, an equality constraint is needed to filter out most of the infeasible points, i.e., the configurations not reachable by the inverse kinematics. The function `findAllIKSolutions` in `torquesNormSum` returns empty sets as well when the inverse kinematics is unable to find a solution, but is way too computationally expansive to check for all search points as the search space is mostly composed of infeasible points. A pseudocode of this equality constraint is in Algorithm 2. The function `setFromIK` sets the joint state of a particular joint group from the inverse kinematics computed for an end-effector pose given as its second argument. This looks for any solution. It returns true on success or false otherwise setting the Boolean variables $foud\_ik\_left$ and $foud\_ik\_right$. This equality constraint returns 0 upon finding a feasible point or else it returns 1. This logic was chosen to comply with the standardized definition of an equality constraint of an optimization task as $f(x) = 0$.
The optimization task we are trying to solve during this search for optimal draw configurations is the following:

$$
\begin{aligned}
\min_{x,y,z,\psi} \quad & \texttt{torquesNormSum}(x,\ y,\ z,\ \psi,\ \alpha,\ \phi,\ rm,\ \boldsymbol{\theta}_{lcf},\ \boldsymbol{\theta}_{rcf}) && (4.13a)\\
\text{subject to} \quad & \texttt{isInfeasible}(x,\ y,\ z,\ \psi,\ \alpha,\ \phi,\ rm) = 0 && (4.13b)\\
& x_{min} \leq x \leq x_{max} && (4.13c)\\
& y_{min} \leq y \leq y_{max} && (4.13d)\\
& z_{min} \leq z \leq z_{max} && (4.13e)\\
& \psi_{min} \leq \psi \leq \psi_{max}. && (4.13f)
\end{aligned}
$$

The minimization variables are only x, y, z and ψ, the rest is fixed for a given draw configuration search. The searches were conducted multiple times for multiple pitch angles to accommodate for various robot-to-target positions. The roll angle ϕ was set to zero.

A lot of classical approaches to solving optimization problems rely on some convenient properties of the optimization criterion and constraints, e.g., smoothness or even convexity. These enable the solver to rather quickly find a local or even global optimum using the gradient of the criterion.
Unfortunately, the functions `torquesNormSum` and `isInfeasible` are both nonlinear non-smooth functions mainly due to the need to compute the inverse kinematics solution/s in a non-analytical way and pick the best ones. The gradient does not exist for these functions.
In such cases, we are unable to claim about any solution, it is truly optimal (globally or locally) unless we perform an exhaustive search through all feasible points. This is, however, even with modern computers impossible to do in a reasonable time for nontrivial problems. The methods actually usable for these kinds of problems - and used to find draw configurations for this thesis

Algorithm 1: torquesNormSum

```
Function torquesNormSum($x$, $y$, $z$, $\psi$, $\alpha$, $\phi$, $rm$, $\boldsymbol{\theta}_{lcf}$, $\boldsymbol{\theta}_{rcf}$):
    $T_l^0$, $T_r^0 \leftarrow$ transformToToolLinks($x$, $y$, $z$, $\psi$, $\alpha$, $\phi$, $rm$);
    $\Theta_l$, $\Theta_r \leftarrow$ findAllIKSolutions($T_l^0$, $T_r^0$, $rm$);
    if $\Theta_l \neq \emptyset$ & $\Theta_r \neq \emptyset$ then
        $n_l^* \leftarrow \infty$;
        foreach joint configuration $\boldsymbol{\theta}_l$ in $\Theta_l$ do
            $rm.left \leftarrow$ setJointPositions($\boldsymbol{\theta}_l$);
            $rm.right \leftarrow$ setJointPositions($\boldsymbol{\theta}_{rcf}$);
            if isInSelfCollision($rm$) then
                continue;
            end
            $n_l \leftarrow$ getTorquesNorm($rm$, "left");
            if $n_l < n_l^*$ then
                $n_l^* \leftarrow n_l$;
            end
        end
        $n_r^* \leftarrow \infty$;
        foreach joint configuration $\boldsymbol{\theta}_r$ in $\Theta_r$ do
            $rm.left \leftarrow$ setJointPositions($\boldsymbol{\theta}_{lcf}$);
            $rm.right \leftarrow$ setJointPositions($\boldsymbol{\theta}_r$);
            if isInSelfCollision($rm$) then
                continue;
            end
            $n_r \leftarrow$ getTorquesNorm($rm$, "right");
            if $n_r < n_r^*$ then
                $n_r^* \leftarrow n_r$;
            end
        end
        return $n_l^* + n_r^*$;
    else
        return $\infty$;
    end
```

Algorithm 2: isInfeasible

```
Function isInfeasible($x$, $y$, $z$, $\psi$, $\alpha$, $\phi$, $rm$):
    $T_l^0$, $T_r^0 \leftarrow$ transformToToolLinks($x$, $y$, $z$, $\psi$, $\alpha$, $\phi$, $rm$);
    $found\_ik\_left \leftarrow$ setFromIK($rm.left$, $T_l^0$);
    $found\_ik\_right \leftarrow$ setFromIK($rm.right$, $T_r^0$);
    return ($found\_ik\_left$ & $found\_ik\_right$) ? 0 : 1;
```

- perform a sampling of the search space to find a *population* of candidate solutions, which is refined until some condition (low speed of improvement, a defined time has passed ...) is met and finally combined to a solution. These methods can find only a "sufficiently good" solution, but we will call their final product "optimal" for brevity throughout this thesis.
A portion of these methods like Tabu Search use their previously acquired search results to guide them through the search space in a deterministic way, whereas genetic or evolutionary algorithms take a more stochastic approach.
An actual algorithm used for the draw configuration search was an evolutionary algorithm ISRES (Improved Stochastic Ranking Evolutionary Search).
In general, evolutionary algorithms refine their population of candidate solutions by a random *mutation* of the individual, *crossover* or the random recombination of several existing candidates and a random *selection* giving greater chance of better candidates to survive to the next generation [fls].
The ISRES algorithm specifically uses stochastic ranking of the candidate solutions to strike a good balance between avoiding evolving infeasible solutions too much, resulting in a bad feasible space exploration (especially for disjoint feasible space), and between evolving them too much leading to a bad feasible space exploitation. This ranking has been introduced in [RY00] and improved in [YR05].

4.4 Design of bow drawing trajectories

After some optimal draw configurations had been found, it was necessary to design for each one the whole trajectory from some undrawn state to that particular optimal draw configuration. This trajectory should not stress the motors disproportionately as well, otherwise all the work put into finding the optimal final draw configuration would be in vain. To achieve this, some constraints needed to be put on the trajectories:

- Orientations of both end-effectors needed to be (roughly) the same at all times
- The position of the left end-effector needed to be (roughly) in the bow plane (x-z plane in the bow coordinates in Figure 4.1) at all times, from here on called "synchronous" movement

The procedure of the design itself using motion planning was started with the robot in a particular optimal draw configuration with the left hand closed and progressed towards an undrawn bow configuration. Some key joint configurations (in each phase of the undraw) were stored along the way and thanks to them the process could be planned in the opposite direction, giving the draw trajectory we were looking for. This method was chosen because the final draw configuration was known and a specific undrawn configuration was not required as it is possible to move between various undrawn configurations (the robot does not hold the bow string) without any constraints.
The undraw trajectory was designed in 4 phases:

1. Move both hands on a (more or less) straight line towards each other as close to each other as possible

2. Move both hands to draw length of 0.3 m, synchronous movement of the hands parallel to the xy bow plane is allowed

3. Undraw from 0.3 m completely, movement in any direction allowed as long as hands are synchronous

4. Open the left hand and move it away from the string, movement in a plane parallel to the xy bow plane allowed

Before diving in depth in the algorithms in each phase, it is appropriate to first introduce the function they will rely on: `moveArmsToPoses`. A pseudocode of this function is in Algorithm 3. Besides some variables we already encountered like tool link poses T_l^0 and T_r^0 and robot model rm, it takes in the torso joint value l, a string $first\_arm$ to specify which arm to move first and three data structures mg, mg_left and mg_right. The abbreviation mg stands for "move group" and these data structures provide a way to plan and execute movements for a certain joint group - mg_left for left arm, mg_right for right arm and mg for both arms + torso prismatic joint. These variables encompass a communication interface with the movement planner and joint motor controllers. This will be discussed in detail in Section 5.3.
The target joint configuration for a movement is set by `setJointValueTarget` and the function `move` plans and if successful, executes movement to the last joint target. It returns true on success, false otherwise. In short, the `moveArmsToPoses` function gets the closest inverse kinematics solution for some given tool link poses and tries to move the robot to this joint configuration either both arms at once or one arm after another. If the movement of the second arm fails, the first arm must be moved back to its initial position. A pseudocode of the function `findClosestSolution` is also provided in the Algorithm 4. This function returns a Boolean success/fail variable and the closest inverse kinematics solution to some initial joint configurations. This is needed as a big difference in initial and target joint configurations signifies an unnecessarily complicated movement undesirable for the draw trajectory. The inefficient double loop is required, in contrast to the Algorithm 1, because the arms are expected to operate close to each other at times, thus the self-collision needs to be checked for every possible combination of joint configurations provided by the inverse kinematics. The `distance` function performs Euclidean distance computation between its argument vectors.

Now, the individual phases will be described. All the phases use the function `moveArmsToPoses` to implement a search for a bow pose (and draw length sometimes) that is reachable by both inverse kinematics and the planner while adhering to some path constraints characterising that phase. These constraints were of two kinds - orientation and end-effector position constraints. The orientation constraints ensured only a small deviation of the end-effectors' orientation quaternions throughout that phase. These were the

Algorithm 3: moveArmsToPoses

```
Function moveArmsToPoses($T_l^0$, $T_r^0$, $l$, $first\_arm$, $mg$, $mg\_left$, $mg\_right$, $rm$):
    $\boldsymbol{\theta}_{l,init}$ ← getCurrentJointValues($mg\_left$);
    $\boldsymbol{\theta}_{r,init}$ ← getCurrentJointValues($mg\_right$);
    $found\_solution$, $\boldsymbol{\theta}_l$, $\boldsymbol{\theta}_r$ ← findClosestSolution($T_l^0$, $T_r^0$, $\boldsymbol{\theta}_{l,init}$, $\boldsymbol{\theta}_{r,init}$, $rm$);
    if $found\_solution$ then
        setJointValueTarget($mg$, [$l$, $\boldsymbol{\theta}_l$, $\boldsymbol{\theta}_r$]);
        setJointValueTarget($mg\_left$, $\boldsymbol{\theta}_l$);
        setJointValueTarget($mg\_right$, $\boldsymbol{\theta}_r$);
        if $first\_arm$ = "both" then
            $found\_solution$ ← move($mg$);
        else if $first\_arm$ = "left" then
            if move($mg\_left$) then
                if not move($mg\_right$) then
                    setJointValueTarget($mg\_left$, $\boldsymbol{\theta}_{l,init}$);
                    move($mg\_left$);
                    $found\_solution$ ← false;
                end
            else
                $found\_solution$ ← false;
            end
        else if $first\_arm$ = "right" then
            if move($mg\_right$) then
                if not move($mg\_left$) then
                    setJointValueTarget($mg\_right$, $\boldsymbol{\theta}_{r,init}$);
                    move($mg\_right$);
                    $found\_solution$ ← false;
                end
            else
                $found\_solution$ ← false;
            end
        end
    end
    return $found\_solution$
```

Algorithm 4: findClosestSolution

```
Function findClosestSolution($T_l^0$, $T_r^0$, $\boldsymbol{\theta}_{l,init}$, $\boldsymbol{\theta}_{r,init}$, $rm$):
    $\Theta_l$, $\Theta_r \leftarrow$ findAllIKSolutions($T_l^0$, $T_r^0$, $rm$);
    $found\_solution \leftarrow$ false;
    $d^* \leftarrow \infty$;
    if $\Theta_l \neq \emptyset$ & $\Theta_r \neq \emptyset$ then
        foreach joint configuration $\boldsymbol{\theta}_l$ in $\Theta_l$ do
            foreach joint configuration $\boldsymbol{\theta}_r$ in $\Theta_r$ do
                setJointPositions($rm.left$, $\boldsymbol{\theta}_l$);
                setJointPositions($rm.right$, $\boldsymbol{\theta}_r$);
                if isInSelfCollision($rm$) then
                    continue;
                end
                $found\_solution \leftarrow$ true;
                $d \leftarrow$ distance($\boldsymbol{\theta}_l$, $\boldsymbol{\theta}_{l,init}$ ) + distance($\boldsymbol{\theta}_r$, $\boldsymbol{\theta}_{r,init}$ );
                if $d < d^*$ then
                    $d^* \leftarrow d$;
                    $\boldsymbol{\theta}_l^* \leftarrow \boldsymbol{\theta}_l$;
                    $\boldsymbol{\theta}_r^* \leftarrow \boldsymbol{\theta}_r$;
                end
            end
        end
    end
    return $found\_solution$, $\boldsymbol{\theta}_l^*$, $\boldsymbol{\theta}_r^*$
```

same for all phases, making orientation of the end-effectors roughly the same as the final draw orientation for the whole draw trajectory. The end-effector position constraints, on the other hand, kept the particular end-effector in a box in space. The dimensions and center of this box were phase-specific.
Phase 1 tries to find a straight line undraw movement, thus the end-effectors' positions were constrained mainly in the y and z directions in the bow coordinated frame. A pseudocode in Algorithm 5 describes this search. This function takes in the bow pose, torso joint value, l, minimal, maximal, and step values for draw length, maximal and step values for bow position difference in x direction and robot data rd. The robot data variable is a container for the robot model rm and all the necessary move groups, namely mg, mg_left and mg_right. This function essentially shifts the bow pose for each draw length (from minimal to maximal) along its x axis until it is able to move the arms to this configuration. The function `transformToToolLinks` is overloaded to accept different inputs here - the bow pose as a transformation matrix and the draw length, which was in the previous section set globally and fixed. The function `shiftInOwnCoords` performs for an input transformation matrix T

and a translation vector $[x, y, z]$ the following computation:

$$T \cdot \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (4.14)$$

Algorithm 5: lookForMinDrawOnLine

```
Function lookForMinDrawOnLine($T^0_{bow}$, $l$, $D_{min}$, $D_{max}$, $D_{step}$,
  $\Delta_{bow,max}$, $\Delta_{bow,step}$, $rd$):
    $found\_solution \leftarrow$ false;
    $D \leftarrow D_{min}$;
    while $D \leq D_{max}$ & not $found\_solution$ do
        $\Delta_{bow} \leftarrow 0$;
        while $\Delta_{bow} \leq \Delta_{bow,max}$ & not $found\_solution$ do
            $T^0_l$, $T^0_r$ ← transformToToolLinks($T^0_{bow}$, $D$, $rd.rm$);
            if moveArmsToPoses($T^0_l$, $T^0_r$, $l$, "right", $rd.mg$, $rd.mg\_left$,
              $rd.mg\_right$, $rd.rm$) then
                $foud\_solution \leftarrow$ true;
            end
            $T^0_{bow}$ ← shiftInOwnCoords($T^0_{bow}$, $[-\Delta_{bow,step}, 0, 0]$);
            $\Delta_{bow} \leftarrow \Delta_{bow} + \Delta_{bow,step}$;
        end
        $T^0_{bow}$ ← shiftInOwnCoords($T^0_{bow}$, $[\Delta_{bow,step} + \Delta_{bow,max}, 0, 0]$);
        $D \leftarrow D + D_{step}$;
    end
    return $found\_solution$;
```

Phase 2 follows with the search for a bow pose on the horizontal bow plane and the draw length of 0.3 m. The movement in the z direction in the bow frame was mainly limited by the position constraints. The pseudocode of this search is in Algorithm 6. It essentially places the bow pose at successive points in a grid in the xy direction with a given draw length D until it finds a reachable pose.

Phase 3 meant the string was returned to its undrawn position - for our bow at the distance of 0.175 m from the bow. Because of this small string-to-bow distance, the left hand actually needed to move down during this phase to avoid collision with the right arm. This is problematic in two ways: first, the left hand needed to slide along the string still under some tension creating friction between the glove and the string, and second, the left hand being lower than the right hand while drawing creates additional torque on the right end-effector. For this reason, we increment z_{off} during the search in phase 3 from some minimal value. Usually, it was enough to only change z_{off} with a fixed bow pose (and a fixed draw length of 0.175 m) using the process described in Algorithm 7. In some cases, this was not sufficient and the bow

Algorithm 6: lookForXYBowPose

```
Function lookForXYBowPose($T^0_{bow}$, $l$, $D$, $x_{max}$, $x_{step}$, $y_{max}$, $y_{step}$, $rd$):
    $found\_solution \leftarrow$ false;
    $x \leftarrow 0$;
    $y \leftarrow 0$;
    while $x \leq x_{max}$ & not $found\_solution$ do
        while $y \leq y_{max}$ & not $found\_solution$ do
            $T^0_l, T^0_r \leftarrow$ transformToToolLinks($T^0_{bow}$, $D$, $rd.rm$);
            if moveArmsToPoses($T^0_l$, $T^0_r$, $l$, "both", $rd.mg$, $rd.mg\_left$, $rd.mg\_right$, $rd.rm$) then
                $foud\_solution \leftarrow$ true;
            end
            $T^0_{bow} \leftarrow$ shiftInOwnCoords($T^0_{bow}$, $[0, y_{step}, 0]$);
            $y \leftarrow y + y_{step}$;
        end
        $T^0_{bow} \leftarrow$ shiftInOwnCoords($T^0_{bow}$, $[x_{step}, -y_{max} - y_{step}, 0]$);
        $x \leftarrow x + x_{step}$;
        $y \leftarrow 0$;
    end
    return $found\_solution$;
```

pose needed to be shifted in a cube in the space around the starting position of this phase. This is illustrated in the Algorithm 8.

Finally, phase 4 was the easiest to design as after the opening of the left hand it was sufficient to get both hands sideways apart. The movement occurred in a plane parallel to the xy bow plane in order for the left hand not to collide with the right arm or the arrow. The search was very similar to the Algorithm 6, except the fact that only one of the tool links was shifted along the x and y directions in the bow frame instead of the whole bow pose. As mentioned earlier, the start and final joint configurations of each phase of the undraw trajectory together with the constraints for each phase were stored and allowed an easy planning of the draw trajectory. These trajectories were stored in files, this allowed them to be replayed later exactly without any uncertainty coming from the planner.

4.5 Target detector

The task for the target detector is to recognize the target in an image and estimate its position in the camera frame. To do that, a dimension in pixels of the target must be also extracted from the image and matched to its corresponding real world dimension. The target with the radii of all of its circles marked, is in Figure 4.3.

Algorithm 7: lookForFinalUndraw

```
1  Function lookForFinalUndraw($T^0_{bow}$, $l$, $D$, $z_{off,min}$, $z_{off,max}$,
     $z_{off,step}$, $first\_arm$, $rd$):
2      $found\_solution$ ← false;
3      $z_{off}$ ← $z_{off,min}$;
4      while $z \leq z_{off,max}$ & not $found\_solution$ do
5          $T^0_l$, $T^0_r$ ← transformToToolLinks($T^0_{bow}$, $D$, $z_{off}$, $rd.m$);
6          if moveArmsToPoses($T^0_l$, $T^0_r$, $l$, $first\_arm$, $rd.mg$,
             $rd.mg\_left$, $rd.mg\_right$, $rd.rm$) then
7              $foud\_solution$ ← true;
8          end
9          $z_{off}$ ← $z_{off}$ + $z_{off,step}$;
10     end
11     return $found\_solution$;
```

Algorithm 8: lookForFinalUndrawVarBow

```
1  Function lookForFinalUndrawVarBow($T^0_{bow}$, $l$, $D$, $z_{off,min}$, $z_{off,max}$,
     $z_{off,step}$, $x_{max}$, $x_{step}$, $y_{max}$, $y_{step}$, $z_{max}$, $z_{step}$, $first\_arm$, $rd$):
2      $found\_solution$ ← false;
3      $x$ ← 0;
4      $y$ ← 0;
5      $z$ ← 0;
6      while $x \leq x_{max}$ & not $found\_solution$ do
7          while $y \leq y_{max}$ & not $found\_solution$ do
8              while $z \leq z_{max}$ & not $found\_solution$ do
9                  if lookForFinalUndraw($T^0_{bow}$, $l$, $D$, $z_{off,min}$, $z_{off,max}$,
                     $z_{off,step}$, $first\_arm$, $rd$) then
10                     $foud\_solution$ ← true;
11                 end
12                 $T^0_{bow}$ ← shiftInOwnCoords($T^0_{bow}$, $[0, 0, z_{step}]$);
13                 $z$ ← $z + z_{step}$;
14             end
15             $T^0_{bow}$ ← shiftInOwnCoords($T^0_{bow}$, $[0, y_{step}, -z_{max} - z_{step}]$);
16             $y$ ← $y + y_{step}$;
17             $z$ ← 0;
18         end
19         $T^0_{bow}$ ← shiftInOwnCoords($T^0_{bow}$, $[-x_{step}, -y_{max} - y_{step}, 0]$);
20         $x$ ← $x + x_{step}$;
21         $y$ ← 0;
22     end
23     return $found\_solution$;
```

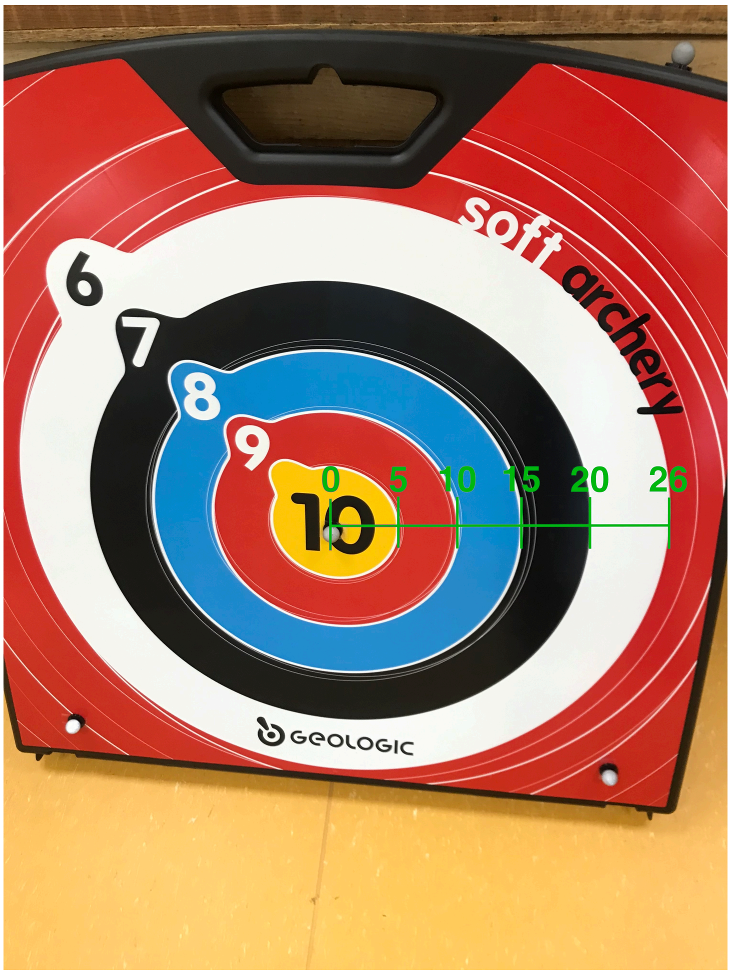

Figure 4.3: Target with dimensions in cm

There are multiple possible approaches to this problem. Two different methods were used for the target detection. Both of these methods were fairly easily implementable and sufficient for the purposes of this thesis, but recognizably suboptimal with room for improvement as discussed later in the Experimental evaluation chapter.
The first method was a concentric circles detector, whose job was to find some concentric circles in the image and perform a color segmentation to clarify which circle was actually found. The second one was a neural network trained for finding the whole target and drawing a bounding box around it in the image.
To compute the target position estimate given the target center pixel coordinates, a pixel dimension and a matching real world dimension, both of these methods used the formulas derived in the following paragraphs.
A standard pinhole camera model illustrated in Figure 4.4 is used. Let us have two points with some unknown coordinates in the camera frame labeled $[U, V, W]$ and $[X, Y, Z]$. The pixel coordinates of these points according to

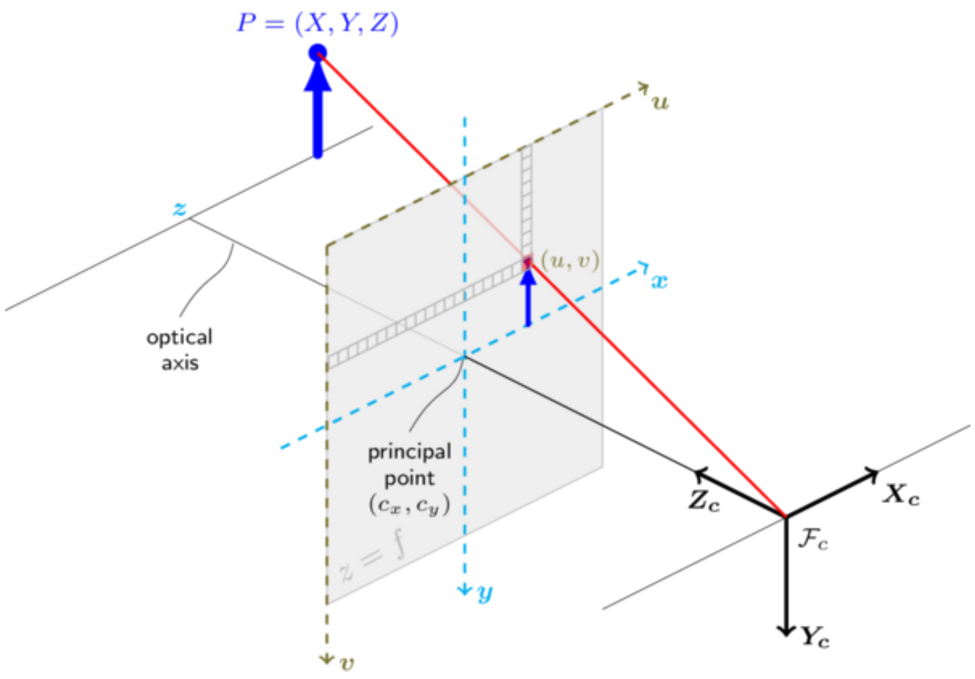

Figure 4.4: Pinhole camera model, from [ope19].

the camera equation are:

$$\lambda_1 \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = P \begin{bmatrix} U \\ V \\ W \\ 1 \end{bmatrix} \tag{4.15}$$

$$\lambda_2 \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = P \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}, \tag{4.16}$$

where

$$P = \begin{bmatrix} f_x & 0 & c_x & 0 \\ 0 & f_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{4.17}$$

is a matrix composed of camera intrinsic parameters like the focal lengths f_x and f_y and the principal point coordinates c_x and c_y. We assume both points lie on a plane parallel to the image plane, thus $W = Z = \lambda_1 = \lambda_2 = \lambda$. Moreover, we know the distance of these points in pixels called r and in the real world R. From this information, we would like to compute the 3D coordinates of one of the points U, V and W. After subtracting the Equation

4.15 from the Equation 4.16 (and $\lambda_1 = \lambda_2 = \lambda$) we get:

$$\lambda \begin{bmatrix} x-u \\ y-v \\ 0 \end{bmatrix} = P \begin{bmatrix} X-U \\ Y-V \\ 0 \\ 0 \end{bmatrix}. \tag{4.18}$$

Now we need to multiply the Equation from the left by the transpose of the left side:

$$\begin{bmatrix} x-u & y-v & 0 \end{bmatrix} \lambda^2 \begin{bmatrix} x-u \\ y-v \\ 0 \end{bmatrix} = (\begin{bmatrix} x-u & y-v & 0 \end{bmatrix} \lambda) P \begin{bmatrix} X-U \\ Y-V \\ 0 \\ 0 \end{bmatrix}, \tag{4.19}$$

then we need to substitute the transposition of the right side of the Equation 4.18 to the right side and simplify the left side:

$$\lambda^2((x-u)^2 + (y-v)^2) = \begin{bmatrix} X-U & Y-V & 0 & 0 \end{bmatrix} P^T P \begin{bmatrix} X-U \\ Y-V \\ 0 \\ 0 \end{bmatrix}. \tag{4.20}$$

This gives us the square of the pixel distance on the left side. Now we need to exploit the inner structure of P:

$$P^T P = \begin{bmatrix} f_x^2 & 0 & f_x c_x & 0 \\ 0 & f_y^2 & f_y c_y & 0 \\ f_x c_x & f_y c_y & c_x^2 + c_y^2 + 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}. \tag{4.21}$$

After substituting this into the Equation 4.20 we get finally:

$$\lambda^2 r^2 = f_x^2 (X-U)^2 + f_y^2 (Y-V)^2. \tag{4.22}$$

As f_x and f_y are in practice very similar in value, we can replace them both in this equation by

$$\bar{f} = \frac{f_x + f_y}{2} \tag{4.23}$$

and factor it out:

$$\lambda^2 r^2 = \bar{f}^2((X-U)^2 + (Y-V)^2) = \bar{f}^2 R^2 \tag{4.24}$$

giving us finally:

$$\lambda = W = \bar{f}\frac{R}{r}. \tag{4.25}$$

This, of course, introduces some imprecision to our computation, which is however for our purposes justifiably negligible. We can estimate roughly some error upper bound by first expressing f_x and f_y by $\bar{f}$ as $f_x = \bar{f} - c$ and $f_y = \bar{f} + c$, then substituting to 4.22:

$$\lambda^2 r^2 = (\bar{f}^2 - 2c\bar{f} + c^2)(X - U)^2 + (\bar{f}^2 + 2c\bar{f} + c^2)(Y - V)^2 \tag{4.26}$$

and this gives us after some rearrangement:

$$\lambda^2 r^2 = \bar{f}^2 R^2 + c^2 R^2 + 2c\bar{f}((Y - V)^2 - (X - U)^2). \tag{4.27}$$

The relative error can be computed as follows:

$$\lambda_{err} = \left| \frac{\lambda - \lambda_{approx}}{\lambda} \right| = \left| \frac{\sqrt{\bar{f}^2 R^2 + c^2 R^2 + 2c\bar{f}((Y - V)^2 - (X - U)^2)} - \bar{f}R}{\sqrt{\bar{f}^2 R^2 + c^2 R^2 + 2c\bar{f}((Y - V)^2 - (X - U)^2)}} \right|. \tag{4.28}$$

After plugging our fixed values $\bar{f} = 525$ and $c = 1$ and the worst case values of the variable $R = 0.75$, and the expression $(Y - V)^2 - (X - U)^2 = 0.75^2$, we arrive at:

$$\lambda_{err} \approx 0.0019. \tag{4.29}$$

At the distance we are shooting the bow - around 5 m - this gives us errors in λ below 1 cm, which is an order of magnitude less than the actual imprecision as discussed in the Experimental evaluation chapter.
The rest of the point coordinates can be obtained from the Equation 4.15, so for known pixel coordinates u, v, we get:

$$\begin{aligned} U &= \frac{W}{f_x}(u - c_x) \\ V &= \frac{W}{f_y}(v - c_y) \\ W &= \bar{f}\frac{R}{r}. \end{aligned} \tag{4.30}$$

4.5.1 Concentric circles detector

The concentric circles detector looks for concentric circles in the image, finds their center and the pixel radius of the biggest circle. It also needs to recognize which of the circles in the real target matches the biggest circle to use the correct real world radius R. This is done by color segmentation.
A pseudocode of a function `detectTargetCC`, which describes the main part of the detector, is in Algorithm 9. This function takes in the image *img* and the matrix P of camera intrinsic parameters. The function `findAllCircles` tries to find all circles present in the image and returns them as an array of triples in the form $[center_x, center_y, radius]$. The function `findMaxConcentricSet` takes in the array of circles and a maximal distance of circle centers to be still considered concentric. It returns a maximal set of concentric circles. After getting this maximal concentric set of circles, they are sorted by radius in an ascending order and the biggest one - c_{max} - is used to crop the image around it by the `cropToTarget` function.

The color segmentation is then applied to this cropped image inside the function `sortColorSegmentsByArea` and the color segments are sorted by their area from the smallest to the biggest. This function returns an array of indexes representing each color (0 - red, 1 - blue, 2 - black, 3 - white) in this sorted order. The yellow central segment is not considered as it is impossible for it to be the outermost circle in any group of two or more concentric circles. The color region with the biggest area is assumed to belong to the outermost target segment in the cropped image. The index of this color is used to pick the right real world radius from the array $\mathcal{R}$ when calling the function `getTargetPosition`. This function essentially implements the Equation 4.30. Finally, the validity of the color segmentation is checked by the function `isColorSchemeValid`. Mainly, the order of the areas of the segmented color regions must match the order of the colors in the target.

Algorithm 9: detectTargetCC

```
Function detectTargetCC($img$, $P$):
    $\mathcal{R} \leftarrow [10, 15, 20, 26]$;
    $d_{max} \leftarrow 3$;
    $\mathcal{C} \leftarrow$ findAllCircles($img$);
    $\mathcal{C}_{con} \leftarrow$ findMaxConcentricSet($\mathcal{C}$, $d_{max}$);
    $t \leftarrow [0, 0, 0]$;
    $position\_valid \leftarrow$ false;
    if $\mathcal{C}_{con} \neq \emptyset$ then
        $\mathcal{C}_{con,sorted} \leftarrow$ sortByRadius($\mathcal{C}_{con}$);
        $c_{max} \leftarrow \mathcal{C}_{con,sorted}[-1]$;
        $img_{crop} \leftarrow$ cropToTarget($img$, $c_{max}$);
        $index\_array \leftarrow$ sortColorSegmentsByArea($img_{crop}$);
        $i \leftarrow index\_array[-1]$;
        $t \leftarrow$ getTargetPosition($c_{max}[0]$, $c_{max}[1]$, $c_{max}[2]$, $\mathcal{R}[i]$, $P$);
        if isColorSchemeValid($index\_array$) then
            $position\_valid \leftarrow$ true;
        end
    end
    return $position\_valid$, $t$;
```

The color segmentation is performed internally as checking for a certain pixel, if it belongs to a certain box in a 3D space of colors in HSV color space. This color space is often used for color segmentation as distinct colors in an image are usually very well separable in it by box regions [Rub20] [Sto]. An example of a detection of the target in the robot camera together with the image cropped to the target and the segments as provided by the color segmentation are in Figure 4.5.

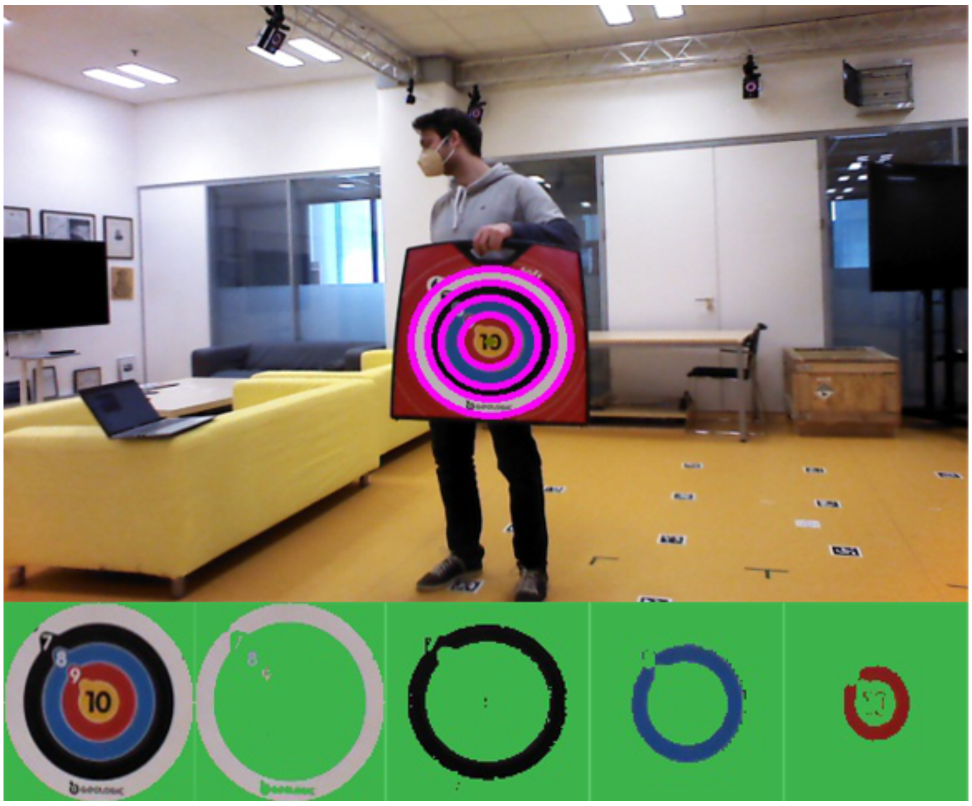

Figure 4.5: Concentric circles detected on target in robot camera (upper image), the bottom images from left to right: image cropped to the target, white color segment, black color segment, blue color segment, red color segment

4.5.2 Neural network detector

A neural network was trained by Ing. Viktor Kozák to recognize the whole target. The system draws a bounding box around the detected target. The center of this bounding box is considered the target position estimation.
Neural networks are very general data structures in artificial intelligence used in a wide range of applications like computer vision, speech recognition, vehicle routing and many more tasks. These networks consist of nodes called neurons interconnected with oriented edges as depicted in Figure 4.6. The network represents a computation graph with each neuron being a mathematical function of the values of the input connections, which returns a value for the output connections. Individual networks differ in the topology and the specific mathematical function each neuron performs. These functions have adjustable parameters called *weights* that grant neural networks their most important ability - they can be trained.
The most straight-forward example of such a network can be a classifier tasked with distinguishing different classes of objects in images. Such a classifier can be trained in a supervised way by providing a lot of labeled images. The weights are in the training process adjusted so that the network returns the correct labels for most of the training data. With the training done correctly

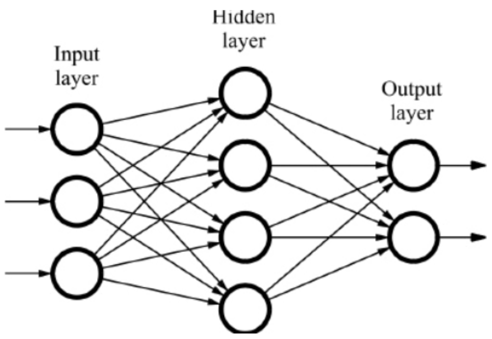

Figure 4.6: A general neural network diagram [dat]

(the network neither underfits nor overfits on the training data), this network is usable for classification on previously unseen images.

The network in Figure 4.6 is a fully-connected neural network: the neurons in each layer are connected to each neuron in the neighboring layers. For computer vision tasks, however, more sparse connections are used - a neuron in the current layer is influenced only by a few neurons close to each other in the previous layer. The image on the output of a layer, called a feature map, is obtained by convolution of the image on its input with a kernel of weights, hence the name Convolutional Neural Networks (CNNs).

Nevertheless, for the purposes of this thesis, an object detection neural network is required rather than a classifier - meaning it must also locate the target in the image, if present. A great choice for this is a Regions with CNN features (R-CNN) network proposed in [GDDM13] . This architecture first finds the regions in the image likely to contain an object (e.g., by Selective Search or Edge Boxes) and then runs a classifier CNN on the regions. The specific network used for the target detection is a Faster R-CNN. Faster R-CNN, proposed in [RHGS15], uses a convolutional neural network even for the detection of regions - Region Proposal Network (RPN). This network predicts object bounds and objectness scores at each position and provides the results to a Fast R-CNN detector [Gir15].

4.6 Detect - Aim - Shoot loop

The final functionality combining bow drawing and target detection is shooting the detected target. After each request to shoot the target, the robot performs a 4-step process:

1. detect the target
2. localize the detected target in the base link coordinate frame
3. pick the best draw configuration to shoot the target and change orientation accordingly
4. draw the bow, change lift joint accordingly and release the string.

Each step will be discussed further in the rest of this section.
The target detection in step 1 is achieved by letting the robot turn in place slowly until the target is detected by the camera. This utilizes the concentric circles target detection, because it is faster than the neural network. After the detection, the head of the robot is pointed to the target.
This is where step 2 begins, consisting mainly of letting the robot look at the target for some time and receive some neural network detection together with several concentric circles detections, that can be averaged out. This gives us some estimation of the target position in the robot base link frame already usable for shooting.
Step 3 is the most complex, because we need to find for each of the available draw configurations the angle of rotation of the robot and the lift joint value required to hit the target with the arrow. These are compared and the draw configuration requiring the least amount of rotation is picked. How the computation of the necessary angle of rotation and the lift joint value were derived will be discussed next.
The derivation of the rotation needed for the alignment with the target can be explained using the Figure 4.7, which is looking at the situation from above. The x and y axes represent the robot base link frame, which is at the center of the robot mobile base. Bow x, y position in this frame is represented by a blue dot with a blue position vector and the label "bow". Bow is at the distance of r from the base link frame origin and its orientation is illustrated by the orange vector going from the bow position having an angle of ψ with the x axis. The target is marked by a red point and it has the magenta position vector in the base link frame. We will compute the angle Δrot between the position vector of "bow" and the position vector of "bow goal" to aim at the target. This angle can be computed as:

$$\Delta rot = -\omega + \varphi + \delta. \tag{4.31}$$

Two of these constituent angles can be easily computed from the known data:

$$\omega = \text{atan2}(bow_y, bow_x) \tag{4.32}$$

and

$$\varphi = \text{atan2}(target_y, target_x). \tag{4.33}$$

To get δ we first need a few more computations. We can get β from a known ψ as

$$\beta = \psi + \pi, \tag{4.34}$$

which can get us ϵ like this:

$$\epsilon = \omega - \beta + \pi = \omega - \psi. \tag{4.35}$$

Bow orientation does not change with respect to the bow position vector after the rotation, therefore we can get :

$$\gamma = \pi - \epsilon. \tag{4.36}$$

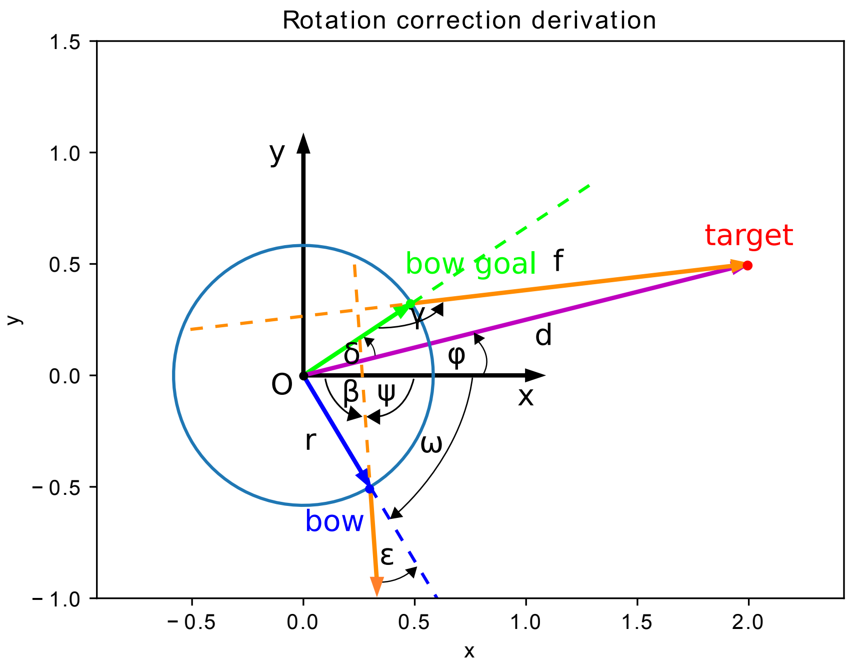

Figure 4.7: Derivation of rotation correction computation

We now have a triangle with vertices "O"-"target"-"bow goal" with two sides known and an angle opposite of the longer side (we can assume $d > r$ always holds). The cosine formula gives us:

$$d^2 = f^2 + r^2 - 2fr\cos\gamma, \tag{4.37}$$

after solving the quadratic equation for f and picking only the positive solution, we get:

$$f = r\cos\gamma + \sqrt{r^2\cos^2\gamma - (r^2 - d^2)}. \tag{4.38}$$

Now, to finally compute δ, we can get its cosine from another cosine formula:

$$\cos\delta = \frac{r^2 + d^2 - f^2}{2rd} \tag{4.39}$$

and its sine from the sine formula:

$$\sin\delta = \frac{f\sin\gamma}{d}. \tag{4.40}$$

Finally

$$\delta = \operatorname{atan2}(\sin\delta, \cos\delta). \tag{4.41}$$

The derivation of the lift joint correction computation will be explained using the Figure 4.8 and its detailed version 4.9. We essentially need to compute the z-coordinate of the intersection of the arrow with the vertical plane containing the target, denoted z_{hit}. We must then modify the lift joint such that the new $z_{hit} = target_z$ to hit the target. The lift joint correction is in that case

$$l_{corr} = target_z - z_{hit}. \tag{4.42}$$

To compute z_{hit}, we model the arrow to have some initial velocity v obtained from the Equation 4.5 with some initial pitch angle α_0. We assume the center of mass of the arrow to be at half its length from the tip. The initial coordinate of this point in s-axis of Figure 4.8 is:

$$s_0 = (\frac{l_{arrow}}{2} - H_0)\cos\alpha_0, \tag{4.43}$$

where l_{arrow} is the arrow length and H_0 is the bow-to-string distance for the undrawn bow. Similarly, its z-coordinate is

$$z_0 = bow_z + (\frac{l_{arrow}}{2} - H_0)\sin\alpha_0. \tag{4.44}$$

With only gravity assumed to act on the arrow, we can obtain the trajectory of the center of mass as follows:

$$\begin{aligned} s &= s_0 + v_{s0}t \\ z &= z_0 + v_{z0}t - \frac{1}{2}gt^2, \end{aligned} \tag{4.45}$$

where $v_{s0} = v\cos\alpha_0$, $v_{z0} = v\sin\alpha_0$, t is time and g is the gravitational acceleration. To obtain the curve representing this trajectory as $z(s)$ we get t from the first equation and substitute it into the second:

$$z(s) = z_0 + \frac{v_{z0}}{v_{s0}}(s - s_0) - \frac{1}{2}g\frac{(s - s_0)^2}{v_{s0}^2}. \tag{4.46}$$

The arrow is assumed to be tangent to this trajectory at every point. Because of this and its nonzero length, it actually hits the vertical plane slightly above the intersect of the trajectory with the plane, as can be seen in Figure 4.9. This makes the computation more complicated, but doable.
We will try to find s_f and z_f from the Figure 4.9. According to the orange right triangle in this figure:

$$s_f^2 + z_f^2 = \frac{l_{arrow}^2}{4}. \tag{4.47}$$

A second equation containing s_f and z_f can be obtained by first taking a derivative of the function $z(s)$ with respect to s:

$$\frac{dz}{ds} = \frac{v_{z0}}{v_{s0}} - \frac{g}{v_{s0}^2}(s - s_0) = \tan(\alpha(s)), \tag{4.48}$$

where $\alpha(s)$ is the arrow pitch angle as a function of s. Furthermore, it holds that

$$\tan(\alpha(f - s_f)) = \frac{z_f}{s_f}, \tag{4.49}$$

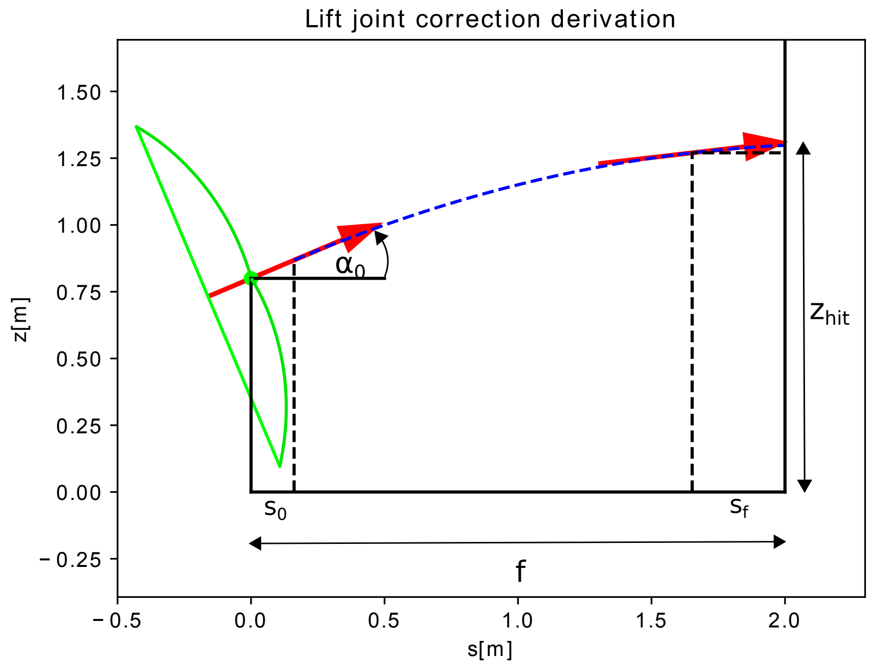

Figure 4.8: Derivation of lift joint correction computation

where f is the bow-to-target distance from the rotation correction computation. This gives us the second equation as

$$\frac{z_f}{s_f} = \frac{v_{z0}}{v_{s0}} - \frac{g}{v_{s0}^2}(f - s_f - s_0). \tag{4.50}$$

After expressing z_f from 4.50, substituting to 4.47 and simplifying, we get

$$\frac{g^2}{v_{s0}^4}s_f^4+2\frac{g}{v_{s0}^3}(v_{z0}-\frac{gf'}{v_{s0}})s_f^3+(\frac{v_{z0}^2}{v_{s0}^2}-2\frac{v_{z0}gf'}{v_{s0}^3}+\frac{g^2f'^2}{v_{s0}^4}+1)s_f^2-\frac{l_{arrow}^2}{4} = 0, \tag{4.51}$$

where $f' = f - s_0$. This equation was solved for s_f by finding the roots of the polynomial on the left, from which a positive s_f with zero imaginary part was picked. z_f was then computed from the Equation 4.50. The z-coordinate of the arrow center of mass on hit can be computed from the Equation 4.46 with the substitution $s = f - s_f$ giving us finally

$$z_{hit} = z(f - s_f) + z_f. \tag{4.52}$$

The function performing the search through the draw configurations has a pseudocode in Algorithm 10. This function takes in an array of draw configurations *draw_configs*, target coordinates in base link frame and the draw length. Each element of the array *draw_configs* is a data structure containing the identification of that draw configuration - pitch angle and configuration number (starting at 0 for each pitch angle) - and the configuration itself given as joint coordinates $\boldsymbol{\theta}$. The `getDrawConfigCorrectionToShootTarget`

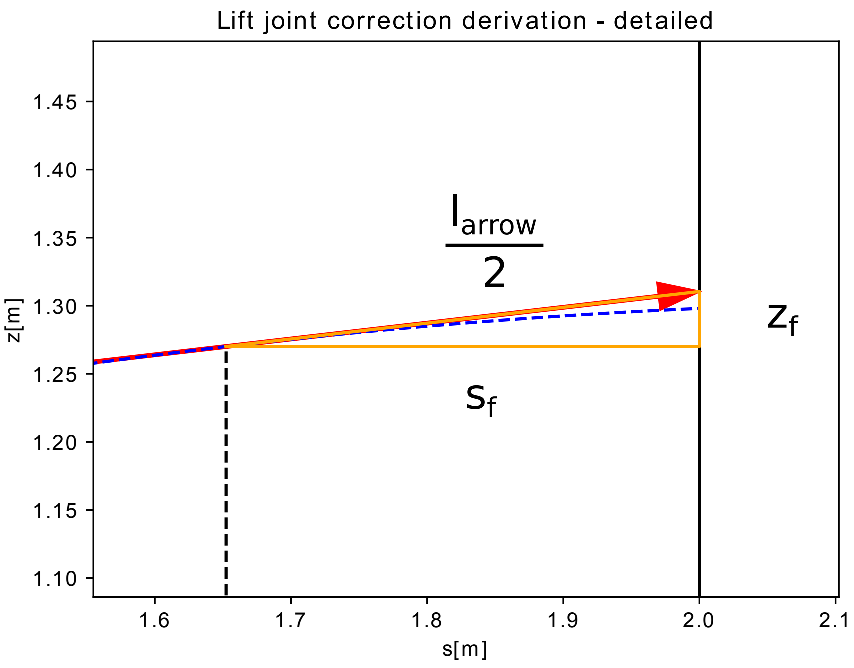

Figure 4.9: Derivation of lift joint correction computation - detailed

function sets an internal robot model to the joint coordinates provided, gets the bow pose for that configuration and then computes the corrections as discussed in the previous paragraphs. It also checks if the lift joint correction is admissible - meaning within the range of the joint and the lower bow tip has some clearance above the ground. It returns infinity for the rotation correction if the lift joint correction is not admissible. The variable dcd^* stands for "optimal draw configuration data" and contains all the information needed to shoot the target, i.e., the draw configuration identification and the corrections. After getting these corrections, the robot is slowly turned in place by the angle Δrot.
Finally, in step 4 the robot draws the bow to the desired configuration, adjusts the lift joint and releases the string.

Algorithm 10: getClosestDrawConfigurationToShootTarget

```
1  Function
   getClosestDrawConfigurationToShootTarget($draw\_configs$, $target_x$, $target_y$, $target_z$, $D$):
2      $\Delta rot^* \leftarrow \infty$;
3      $admissible\_config\_found \leftarrow$ true;
4      $dcd^* \leftarrow [-1, -1, 0, 0.124]$;
5      foreach $dc$ in $draw\_configs$ do
6          $correction \leftarrow$ getDrawConfigCorrectionToShootTarget($dc.\boldsymbol{\theta}$,
            $target_x$, $target_y$, $target_z$, $D$ );
7          if $|correction[0]| < \Delta rot^*$ then
8              $\Delta rot^* \leftarrow |correction[0]|$;
9              $dcd^* = [dc.pitch, dc.number, correction[0], correction[1]]$;
10             $admissible\_config\_found \leftarrow$ true;
11         end
12     end
13     return $admissible\_config\_found$, $dcd^*$;
```

Chapter 5

Implementation

5.1 Implementation overview

This chapter describes the control of the robot via ROS and the actual implementation of the functionalities introduced theoretically in the previous chapter. The sections provide some information about the software libraries used, the pitfalls encountered, and their solutions. Examples of some specific results are added, too. The source code responsible for each functionality provided as an attachment to this thesis is also referenced.

5.2 ROS

This section provides a basic description of ROS as it will be heavily referenced in further sections. Robot Operating System (ROS) is an open source middleware designed for the purposes of robotics, i.e., to enable communication between all components of a usually heterogeneous robotic system and create a unified interface for the developers of robotic applications to interact with the hardware. It also provides other useful tools for package management, debugging, visualization, and more [ros]. ROS was the main tool for the software development needed for this thesis with almost all the source code contained in ROS package `tiago_archer`.
ROS behaves like a distributed system: every application usually has multiple independent programs running at the same time called ROS nodes. These nodes are designed such that they can be run on the same PC or can be scattered along different computers and communicate via a network with no change to their source code. For this to be possible, there must be a main node called ROS Master running at all times. All other nodes register with this node and the communication between any other nodes is initiated by Master. Asynchronous channels used for communication between nodes are called topics. After initiation by master, the topics are used for direct communication between nodes - peer-to-peer model. This is evident from the diagram in Figure 5.1.

ROS topics were used extensively throughout this project. Each topic has a

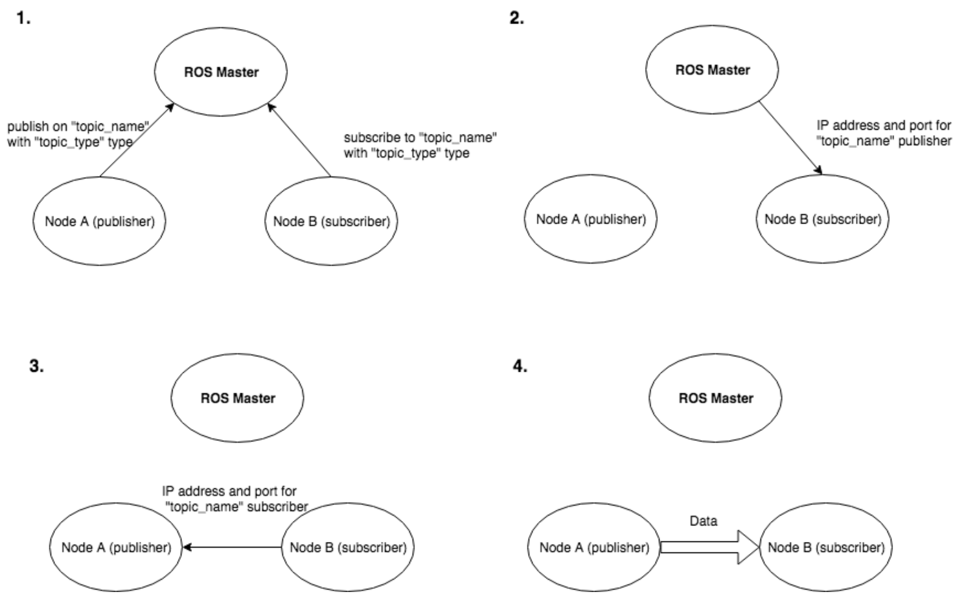

Figure 5.1: Initialization of communication via a ROS topic. **1.** Registration with the Master, **2.** Master sends contact information for publisher to the subscriber, **3.** subscriber contacts the publisher directly with its contact information, **4.** publisher sends data to subscriber. [Jal19]

unique name and supports only one type of message. A message type can be composed of some basic message types in the `std_msgs` package or from other non-trivial message types as demonstrated by a custom message definition from the `tiago_archer` package below.

Listing 5.1: `tiago_archer/RobotTrajectoryArray`

```
string id
moveit_msgs/RobotTrajectory[] trajectory_array
float64[] start_joint_values_left
float64[] start_joint_values_right
float64 start_torso_value
int16 open_close_hand_index
```

Other means of communication between nodes (implemented internally using topics) also utilized within this thesis are services and actions. Services enable synchronous communication conforming to the client-server model, i.e., the client node sends a request and the server node after performing the requested task sends a response to the client.
Actions are similar to services, but the processing of a request called "goal" can be preempted by a different goal or canceled completely. The action server can also send some feedback messages during the execution to inform about its progress before sending the final result. This is fitting for some tasks taking a long time to execute - like a robot movement.

5.3 TIAGo++ robot control

The robot was controlled completely via ROS. The specific ways of controlling the most important robot parts are discussed in this section.
There are several ways the arms can be controlled. Some manual joint control using a GUI from the `rqt_joint_trajectory_controller` ROS package or the WebCommander [PALb] was sometimes used, but most of the controlling was done through code using MoveIt motion planning framework [mova].
MoveIt is a widely used software in modern robotics capable of computing kinematics and planning tasks and moving the joints for any robot given its `robot_description` parameter from the ROS parameter server (and some other configuration parameters).
MoveIt interfaces with the user, ROS and the robot through a ROS node called `move_group`. A diagram illustrating some of the various topics, services and actions this node is communicating through is in Figure 5.2. Essentially, `move_group` uses current joint information obtained from the robot via the `joint_states` topic, the robot model from the `robot_description` parameter, robot position in the map from the ROS TF library and the information about other objects in the workspace from sensors or directly set by the user, to maintain the *planning scene*. This is a representation of the robot in its current state and the world. This planning scene is then used for motion planning, which generates a collision-free path in the joint space of the robot to get to some desired joint or cartesian configuration. This path is then upgraded to a trajectory by adding some time parametrization accounting for the joint velocity and acceleration limits. The finished trajectory is finally handed over to the joint motor controllers to execute it.
Since the main part of the code for this thesis is programmed in C++, the `move_group_interface` is used to communicate with the `move_group` node. More specifically, a class called `MoveGroupInterface` [mgi] from the `moveit::planning_interface` namespace was employed.

A joint group is specified as a constructor argument of the `MoveGroupInterface` class, so each instance can control one joint group at once. The arms can be controlled separately as joint groups "arm_left" and "arm_right" or with torso as "arm_left_torso" and "arm_right_torso" or together as "both_arms_torso".
The torso is also controlled via MoveIt. Apart from the joint groups, where it is combined with the joints of one or both arms, it can also be controlled separately as "torso" joint group.
Unlike the torso, the Hey5 hand end-effector can be controlled only via its dedicated MoveIt joint group "hand_left".
The head orientation is controlled via a convenient action interface called `/head_controller/point_head_action`. This allows to look at a specific 3D point in a given frame.
The mobile base can be manually controlled using a gamepad. To control the base programatically, velocity commands need to be sent to the `/mobile_base_controller/cmd_vel` ROS topic. This topic is of type

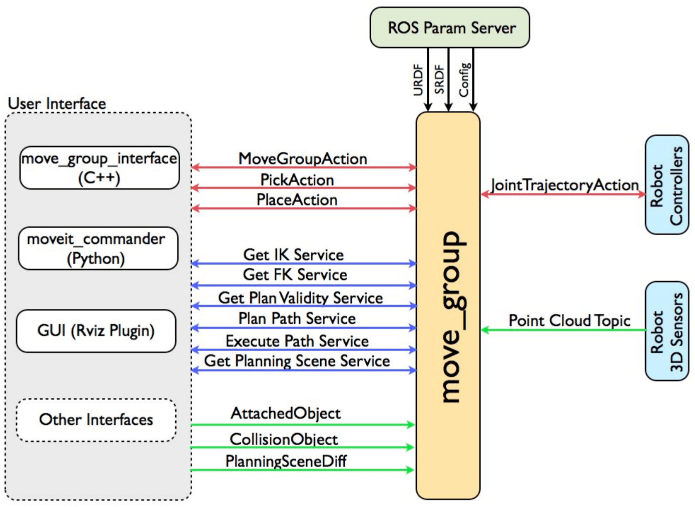

Figure 5.2: `move_group` ROS node [movb]

`geometry_msgs/Twist` and requires a 3D vector of linear and a 3D vector of angular velocities.

5.4 Bow end-effector & bowstring rings

The bow was mounted on the robot by a holder designed and printed on a 3D printer by Dr. Gaël Écorchard. Two rings were also printed for the left hand to decrease friction between the hand and the string when drawing. The bow end-effector and the rings are depicted in Figure 5.3.

5.5 Bow model

This section will discuss the implementation of the bow model in the simulation and compare it to the real bow.
A bow model has been constructed using Gazebo simulator [gaz]. This is a robotics-oriented free software able to build and visualize robot models, simulate their dynamics, generate sensor data etc.
Initially, a SDF (Simulation Description Format) model of the bow has been created using the mathematics of the Bow model in the previous chapter. The rigid handle has been modelled in Blender [ble] and its mesh imported to SDF, while the rest of the model are SDF box links for the limbs and SDF cylinder links for the string. A limb (upper or lower) is represented by a box fixed to the handle of length $\frac{B_1}{4}$ connected by a rotational joint in the point O from the Table 4.1 to a box of length $\frac{3B_1}{4}$. The string is modelled as two rigid cylinders connected to the limbs and to each other by a rotational joint.
A gazebo plugin was used to give the model its dynamic properties: an angle A was constantly checked for both limbs and the torque of the joint in point

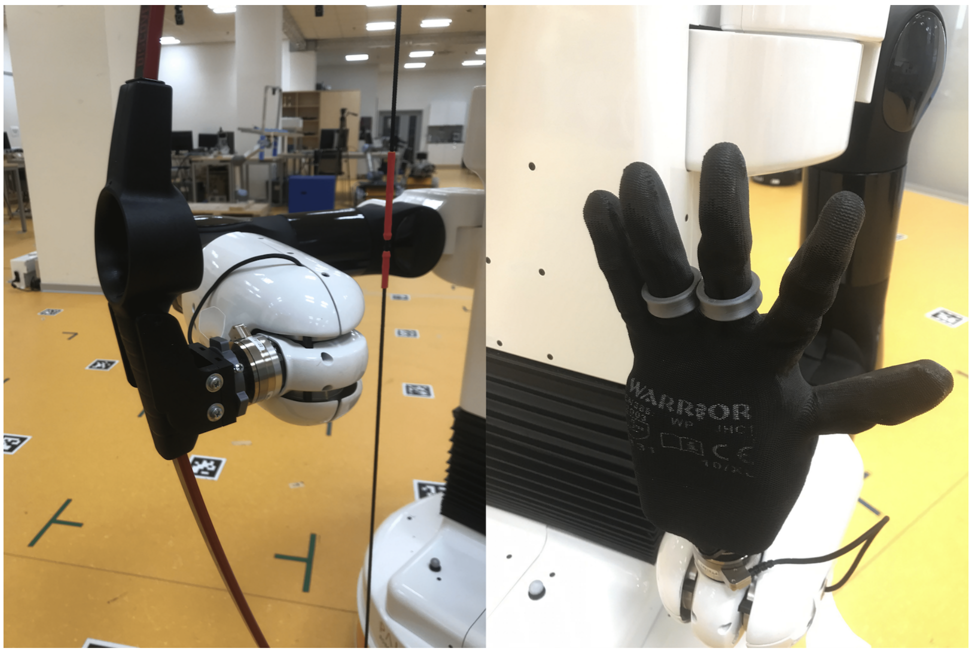

Figure 5.3: Bow end-effector (left) and bowstring rings (right)

O has been set to

$$\tau_{limb} = \frac{3B_1CA}{4} \tag{5.1}$$

to create the force $f = CA$ from the Table 4.1 on the bow tip. The bow model in Gazebo simulator in its undrawn and fully drawn state is depicted in Figure 5.4.

The force-draw curve has been measured on the real bow using a force sensor from [mer] and the constant C has been set so that the analytical force-draw curve obtained from the bow model in the previous chapter and the real measurement differed minimally. More specifically, a linear regression using the least squares method has been used.
It is apparent from the expressions in Table 4.1 the constant C is just a scaling factor of the force-draw curve:

$$F = 2T \sin E = \frac{2CA \sin E}{\sin{(A+E)}} = CF', \tag{5.2}$$

where $F' = \dfrac{2A \sin E}{\sin{(A+E)}}$. This allows us to formulate the problem of finding C minimizing the sum of squares of the differences between the forces measured and the analytical forces for the same draw length as follows:

$$\min_{C} \sum_{i=1}^{i=n} ||CF'_a(D_i) - F_m(D_i)]||^2, \tag{5.3}$$

where the indexes a and m mean "analytical" and "measured", respectively, and n is the number of draw lengths measured. Rewriting it into a matrix

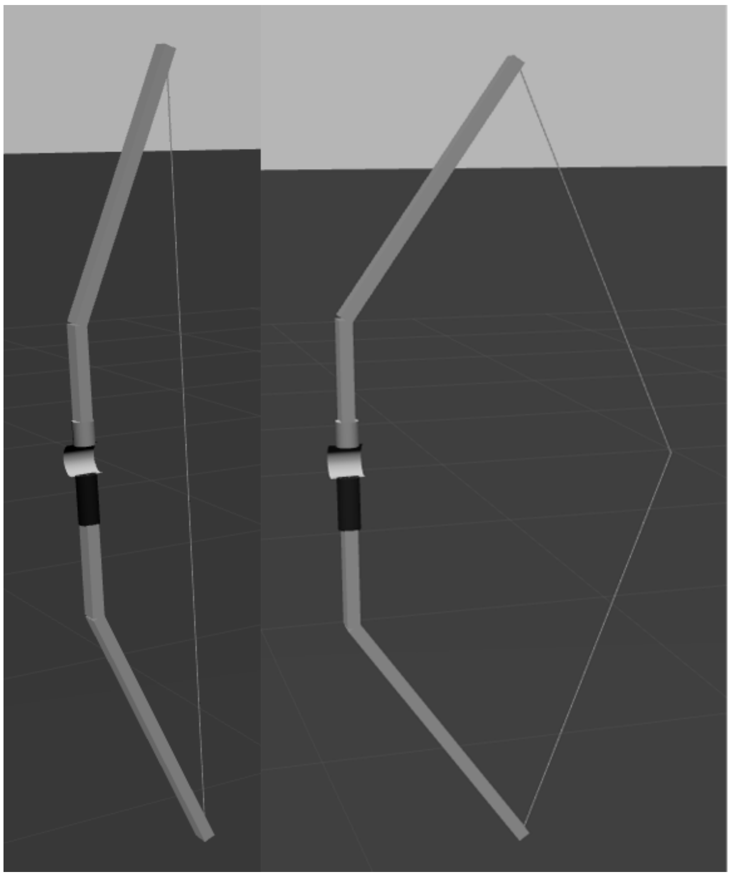

Figure 5.4: Bow Gazebo model undrawn and fully drawn

format yields:

$$\min_{C} \left\| \begin{bmatrix} F_a'(D_1) \\ F_a'(D_2) \\ \dots \\ F_a'(D_n) \end{bmatrix} C - \begin{bmatrix} F_m(D_1) \\ F_m(D_2) \\ \dots \\ F_m(D_n) \end{bmatrix} \right\|^2 . \tag{5.4}$$

Lets label

$$\boldsymbol{a}_F = \begin{bmatrix} F_a'(D_1) \\ F_a'(D_2) \\ \dots \\ F_a'(D_n) \end{bmatrix}, \quad \boldsymbol{b}_F = \begin{bmatrix} F_m(D_1) \\ F_m(D_2) \\ \dots \\ F_m(D_n) \end{bmatrix}, \tag{5.5}$$

then the minimizing C can be computed in closed form as

$$C = (\boldsymbol{a}_F^T \boldsymbol{a}_F)^{-1} \boldsymbol{a}_F^T \boldsymbol{b}_F. \tag{5.6}$$

By this computation, we obtained

$$C = 110.4458 \text{ N/rad}. \tag{5.7}$$

Another plugin was added to the bow model in Gazebo for testing purposes: it applied two opposite forces to the bow - one to the handle and the other

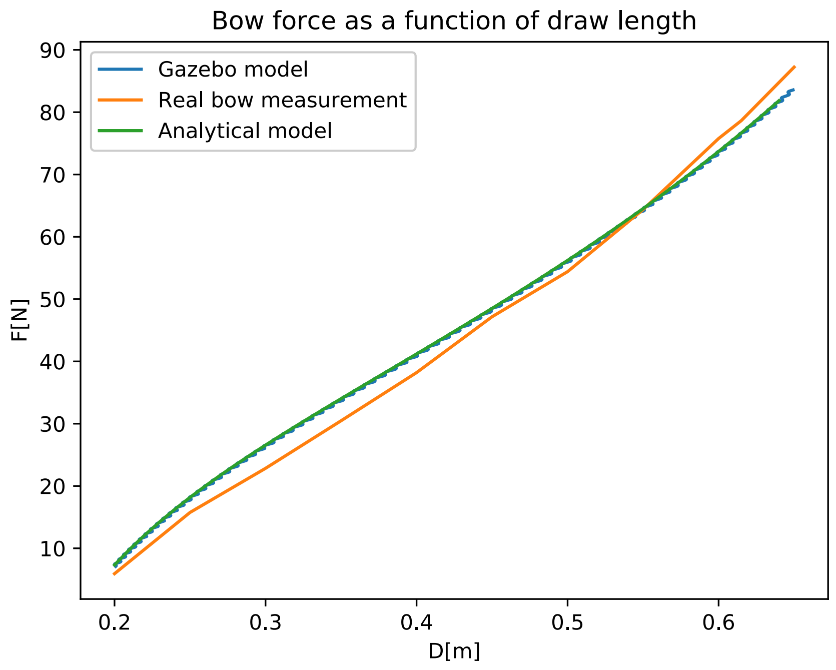

Figure 5.5: Bow force-draw curve

to the string mimicking draw of the bow. The forces were progressively increased and the draw length was being computed from the joint angles of the model. Since we were only interested in static steady state forces, the inertial properties of the links in the model were set to zero to avoid significant oscillations stemming from the bow model dynamics (essentially an undamped linear harmonic oscillator dynamics). The draw lengths and forces during this process were saved into a file.
The Figure 5.5 illustrates the force-draw curve obtained by this process together with an analytical force-draw curve obtained by applying the bow model equations in the previous chapter and finally, it contains a force-draw curve measured on the real bow. It shows that the performance of the Gazebo model coincides with the analytical model. It also shows the model is very close to the real bow force-draw curve.
Later, the SDF model was rewritten into URDF (Unified Robot Description Format) to be integrated into the TIAGo++ robot Gazebo model as an optional end-effector. Unfortunately, this was not fully successful, as URDF supports only the modelling of open kinematic chains (the bow is a closed kinematic chain) and all attempts to circumvent this limitation were so far met with frequent simulation crashes. Although this rendered the direct interaction between the robot model and the bow end-effector model impossible, the bow Gazebo model was still very useful. For example, the simulation did not crash when the bow end-effector model was stationary in the simulation, which enabled the search for optimal draw configurations

as the collision model of the bow end-effector loaded properly allowing the self-collision checking to account for it. There was no need to actually move the robot in the simulation during this process. The robot with the bow end-effector was moved in the simulation during the design of the bow drawing trajectories, but the string was removed from the bow model for this purpose, making its kinematic chain open (limb joint limits were introduced to prevent the limbs from undeflecting completely). This way, the planner could avoid collisions with the bow.
The folder `tiago_archer/bow_and_arrow_models/bow` contains the SDF and URDF models in files `bow.sdf` and `bow_ros.urdf.xacro`, respectively. The Gazebo plugin responsible for the bow properties is implemented in the folder `tiago_archer/src/bow_plugin/` in files `bow_ros_plugin.cpp` and `bow_ros_plugin.h`. The plugin applying force to the bow while testing is in `tiago_archer/src/bow_ros_testing.cpp`. Lastly, for the computation of the constant C and for the plotting of the force-draw curve, Python scripts were used located in the `tiago_archer/scripts` folder in files `compute_C_by_least_squares.py` and `force_draw_plotter.py`.

5.6 Search for optimal bow draw configurations

A ROS node in `tiago_archer/src/find_best_draw_config_nlopt.cpp` was mainly responsible for the search. The functions `torquesNormSum` and `isInfeasible` are implemented here in a straightforward way according to the Algorithms 1 and 2.
This implementation relies on classes like `moveit::core::RobotModel` [roba], `moveit::core::RobotState` [robb] or `planning_scene::PlanningScene` [pla] to construct the robot's model from its URDF description on the parameter server, hold, change and investigate robot's state and associated properties (like a Jacobian) and check for self-collision, respectively.
All necessary spatial transformations and general matrix or vector operations are performed using the Eigen library [GJ+10].
The function `findAllIKSolutions` had to be implemented as the class `moveit::core::RobotState` only provides the function `setFromIK`, which returns at most one possible solution of the inverse kinematics. The KDL kinematics plugin, which is a wrapper around the inverse kinematics solver provided by the Orocos Kinematics and Dynamics Library (KDL) [oro], is used for this internally. This solver is a numerical inverse kinematics solver, meaning it is trying to find a solution joint configuration $\boldsymbol{\theta}$ by finding the roots of the expression

$$T_d - T(\boldsymbol{\theta}), \tag{5.8}$$

where T_d is a transformation matrix of desired 3D pose of the robot end-effector and $T(\boldsymbol{\theta})$ is the transformation matrix of its pose given the joint configuration $\boldsymbol{\theta}$, i.e. the forward kinematics.
The `findAllIKSolutions` function was implemented using the `setFromIK`, which has a mechanism of rejecting invalid solutions and keep looking for

different ones. The validity of a solution is determined by an arbitrary user-defined function passed to `setFromIK` as an input argument. The `findAllIKSolutions` was thus implemented by running `setFromIK` in a loop until it was no longer able to find any solutions *dissimilar* enough from those already found before. The validation determination function therefore returns true when the input solution is not similar to any of the solutions in an array of already found solutions. As the measures of similarity between the solutions, the cosine similarity was used, which is computed for two joint configurations $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ as:

$$sim(\boldsymbol{\theta}_1,\boldsymbol{\theta}_2) = \frac{\boldsymbol{\theta}_1^T\boldsymbol{\theta}_2}{\|\boldsymbol{\theta}_1\|\,\|\boldsymbol{\theta}_2\|}. \tag{5.9}$$

This expression is close to 1 when the joint configurations are similar and close to 0 when they are not. An experimentally obtained threshold of 0.999 was used to distinguish similar and dissimilar solutions.
The `find_best_draw_config_nlopt` node listens to a `/link_inertias` topic for the inertial properties of all links constituting the robot arms - the mass and the position of the center of gravity are obtained this way for every link for use in torque computation. A Gazebo plugin for the robot model implemented in `tiago_archer/src/robot_link_inertias_publisher.cpp` is a publisher of these messages.
The search itself was done by the already introduced ISRES algorithm, specifically its implementation in the NLopt nonlinear optimization library [Joh]. The search was run 50 times to obtain multiple usable solutions for several specific α (pitch) angles of the bow. The result of this effort was 7 optimal draw configurations for $\alpha = 0°$, 4 configurations for $\alpha = 5°$ and $\alpha = -5°$, and 5 configurations for $\alpha = 10°$ and $\alpha = -10°$, so together 25 different draw configurations. A portion of these searches were actually performed with a slightly modified computation of the torques norm than previously stated. More specifically, the norm used in the Equation 4.11 was actually a maximum norm instead of the Euclidean. This sometimes leads to a better distribution of normalized torques across the motors.
The majority of the draw configurations found had the end-effectors at shoulder height or above, similar to how real archers draw a bow. Some configurations found were even very close to a standard bow draw posture as illustrated in Figure 5.6. This points to the optimality of the standard bow draw posture regarding muscle effort. A portion of the robot optimal draw configurations found had the end-effectors at stomach height - this would be unusable for a real archer since optical aiming in this posture is not possible. The Figure 5.6 also shows the "extended string" bow model. This was a part of the robot model during the search. Its purpose was to avoid configurations in which the right arm would be in the way of the shooting string, while permitting collisions near the end of the string trajectory during the shot. Therefore, the original undrawn string stayed in the model, but it was allowed to collide with the right arm, unlike the other two extra strings and the circle to fill part of the space between them.

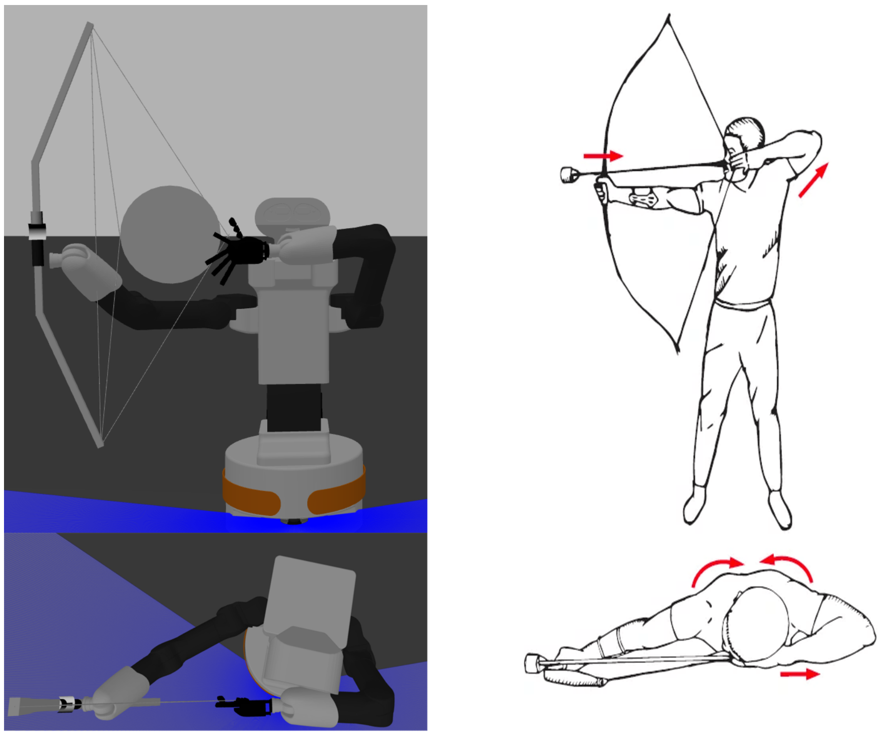

Figure 5.6: A robot bow draw configuration with the extended string bow model (left) and a standard posture for shooting a bow [Arr] (right)

The node implemented in `tiago_archer/src/draw_config_search.cpp` was used to map out the landscape of the optimization criterion, i.e., for a value of ψ (yaw angle) and a bow z position, it performed a grid search across the bow xy position and saved the values of `torquesNormSum` in a file. The general tendency observed from these searches is the function is decreasing in the negative y direction of the base link or the robot's right. This is illustrated in Figure 5.7 depicting a scatter 3D plot of the outputs of `torquesNormSum` as a function of bow xy position for bow $z = 1.0$ m and $\psi = -\frac{3}{8}\pi$.
Finally, a ROS node in `tiago_archer/src/draw_configs_demo.cpp` served to move the robot to one of the draw configurations saved in the `tiago_archer/optimal_draw_configs/` folder.

5.7 Design of bow drawing trajectories

A ROS node in `tiago_archer/src/draw_bow_find_traj.cpp` has been implemented for the design of the bow drawing trajectories. The procedure outlined in the previous chapter has been followed, but soon it became apparent that phase 1 - the search for the minimal draw on the line - was too restrictive and rarely yielded any usable trajectory. Although implemented, this phase was usually skipped.
The designed trajectories were saved to a file for later use. The YAML file

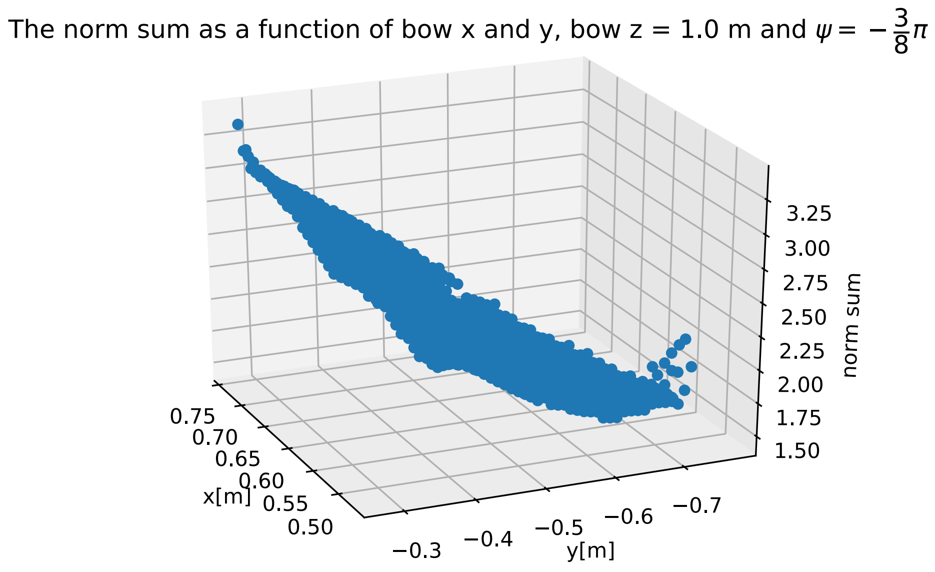

Figure 5.7: The output of `torqueNormSum` as a function of bow xy position

format was chosen for this purpose as it is well suited for the storage of complex structured data. Its interface to Python is extremely easy to use - the `yaml` Python library can store and load any Python object with two lines of code. All trajectories were therefore saved and loaded via ROS nodes implemented in Python scripts.

More specifically, the `draw_bow_find_traj` node sent the trajectories via the `/draw_trajectory` topic of type `tiago_archer/RobotTrajectoryArray` (defined as an example of a complex message in Section 5.2) to the `plan_saver.py` ROS node located in `tiago_archer/scripts`. This node then saved the trajectories to a file in the folder `tiago_archer/trajectories`.

These trajectories could be pieced together into two kinds of movements - a draw/undraw movement mainly for trajectory testing purposes and a draw/release movement. The whole movement from the start position to the final position - the start position again for draw/undraw and the draw configuration for the draw/release movement - could be played with minimal involvement of the planner. The planner was only used to correct small errors of the joint positions before playing a particular component trajectory of a movement as the robot motor controllers refuse to play a trajectory with the starting point differing in any joint by more than 0.01 radians from the current robot state.

To minimize the involvement of the planner in the whole process including getting into a start position of a different movement, in general, an "initial joint configuration" of the robot was introduced. This served as an intermediary

between different movements - for every draw configuration, three trajectories have been found and saved to connect to the initial configuration. These were more specifically, initial configuration to draw start configuration, draw start configuration to the initial configuration, and the final draw configuration to the initial configuration. Together, these enable all possible transitions between movements without the planner. These trajectories were designed using the ROS node in `tiago_archer/src/draw_bow_find_traj_to_init.cpp`. No path constraints were imposed during the design, it was enough for the trajectories to be collision-free and not unnecessarily complicated.
The complete movements including the appropriate transitions can be played with the ROS node in `tiago_archer/src/draw_bow_play_traj.cpp`. This node obtains the draw trajectories via the `get_robot_draw_trajectory` ROS service from the `draw_trajectory_server.py` node.

5.8 Target detector

Both detectors have been implemented as ROS Python nodes located in files in the `tiago_archer/scripts` folder - `target_detector.py` for the concentric circles detector and `target_detector_NN.py` for the neural network. They both subscribe to the `/xtion/rgb/image_rect_color` and `/xtion/rgb/camera_info` ROS topics providing the rectified (corrected for distortion) RGB image from the robot's camera, and the camera intrinsic parameters, respectively. The output of both detectors is a target position estimate advertised as a transformation in the ROS TF framework.

5.8.1 Concentric circles detector

The OpenCV - open-source computer vision library - has been used to implement the concentric circles detector [Its14]. The circles were found in the image by OpenCV's `HoughCircles` function. The foundation of this function is Hough Circle Transformation.
Hough transformations, in general, are algorithms designed to find simple geometric shapes in an image resistant to noise and missing points in the shape. They work with an image outputted by an edge detection algorithm and transform each edge point into the parameter space of the particular shape. This transformation for an edge point determines all the different parameters of the shape making the edge point part of the shape. Parameters for circle are three - center x and y coordinate and radius.
The standard Hough Circle Transformation algorithm iterates through different possible radii and for each one transforms all edge points in the image into the parameter space of only the center coordinates parameters. For a fixed radius, this transformation is:

$$\begin{aligned} a &= x' - r\cos\theta \\ b &= y' - r\sin\theta, \end{aligned} \tag{5.10}$$

where a and b are the x and y coordinates of the center, x' and y' are the edge point image coordinates and r is the radius. Letting $\theta \in < 0, 2\pi >$ makes this also a circle in the 2-parameter space. One edge point thus corresponds to a circle in the parameter space for a fixed r. Taking all edge points therefore yields multiple circles in the 2-parameter space for a fixed r. The points in these circles together with the current r are used to increment values in a 3D array called accumulator at the position $[a, b, r]$. This array is a discretization of the whole 3D parameter space, and all its cells are initialized to 0. Cells with value above a certain threshold are deemed to be the parameters of a circle in the image.
OpenCV uses a more advanced implementation - Hough Gradient Method. After running the input image through Canny edge detector, the gradient is computed for every edge pixel determining the normal direction to the edge. A 2D accumulator array is incremented in both directions of the gradient (positive and negative) from the edge point. This procedure finds candidates for circle centers. Subsequently, the best radius is found for each of these candidate centers by getting the distances from a particular center to all edge points and picking the distance shared by the most edge points (implemented as a 1D accumulator array) [The].
This implementation actually is not able to detect multiple circles with the same center, therefore it was necessary for the concentric circles detector to run `HoughCircles` several times on the same image for various values of minimal and maximal radius and put together all the circles found.
The color segmentation was implemented by the OpenCV `inRange` function, which picks the pixels of the target image with color in a certain box in HSV color space and returns a mask image. This image is a binary matrix with the dimensions of the target image storing "1" for the pixels in the color range and "0" otherwise. The ranges were chosen to contain the main regions where each color segment appeared in the HSV space. These box ranges are depicted in Figure 5.8, which plots all the pixels in a target image in the HSV color space. Two disjoint boxes define the blue segment.
This detector outputs two transformations - the position estimate of the last detected target as `/target_raw` and the median of these positions over 10 detection as `/target_averaged`.

5.8.2 Neural network detector

The neural network was implemented using the Detecto Python module [det]. During the training phase, a file storing the network weights has been created. The detector loads the weights during the initialization and uses them to predict the target bounding boxes for each input image. The box with the highest score is deemed to be the target. Its size and center are used for the target position estimation, being broadcast as `/target_NN` TF transformation.
A separate ROS workspace had to be created for the neural network detector as it requires Python 3 rather than Python 2, which is the version the rest of the system works with.

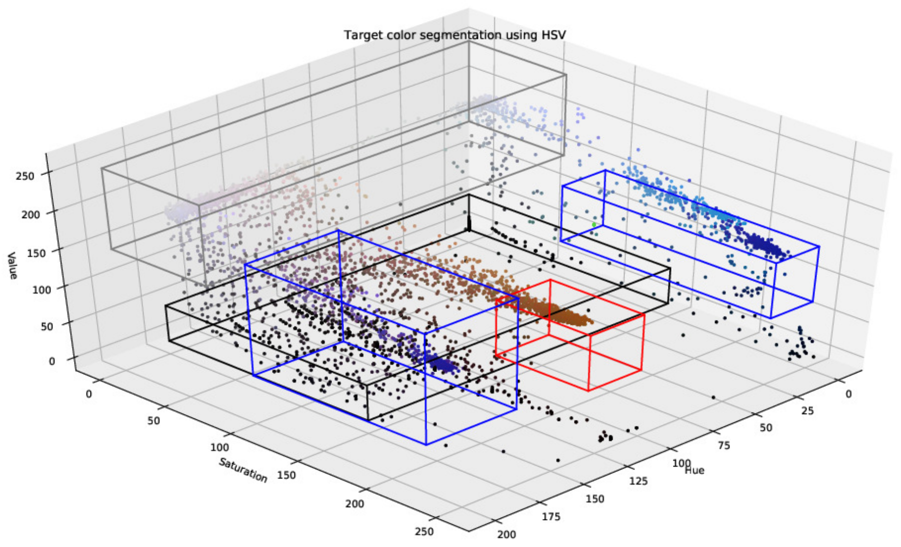

Figure 5.8: Colors of all pixels in a target image in HSV colorspace together with the boxes defining color segment ranges (white, black, blue and red)

5.9 Detect - Aim - Shoot loop

The Detect - Aim - Shoot loop is implemented in the `shoot_to_target.cpp` ROS node located in `tiago_archer/src`. This node is expected to be run simultaneously with both detectors, the `draw_trajectory_server.py` to have access to the drawing trajectories and with a publisher of a static transformation between the `arm_right_tool_link` and the `bow_planning_frame`, which corresponds to the bow frame. This is needed for determining the bow pose when running on the real robot as thanks to using the prerecorded trajectories for every arm movement, the bow model does not need to be incorporated to the real robot.
The implementation stays true to the shooting process and the computations outlined in the Approach chapter. There are, however, a few specifications worth mentioning here.
First, the localization phase uses, rather than a fixed waiting period for the robot to look at the target, a condition on the estimates. More specifically, they must be within 30 cm distance of each other. This signifies the neural network has caught up with the concentric circles (the neural network is lagging behind the concentric circles due to its significantly higher computation time).
For the robot to shoot accurately, a rotation by a precise angle Δrot needs to be implemented. A simple P controller with input saturation (minimal and maximal values for the input angular velocity are defined) can perform this task and orient the robot within 0.001 radians from the goal orientation. The current base link frame orientation in the map is provided by the `amcl` [amc] ROS node, which runs on the robot by default. AMCL stands for adaptive Monte Carlo localization - this node uses the mobile base laser scan data

together with the odometry to maintain a map of its surroundings and localize the robot in the map. The `amcl` node actually publishes the transformation between the `map` and the `odometry` frames. Obtaining the robot orientation with respect to the `map` frame instead of the `odometry` is resistant to errors caused by, e.g., slipping of a wheel. Unfortunately, the localization often (about 1 in 7 shots) underestimates the robot real orientation and as a result misses the goal orientation by up to a few degrees. This is enough to miss the target completely. To combat this, the robot's head is after localization fixed in a position pointing at where the localized target would be with respect to the robot base link after a perfect robot rotation. The rotation then proceeds as before, but after it, the drawing phase is not initiated unless the target center, as obtained by raw concentric circles, is less than 10 pixels away from the image plane center. This distance is obtained additionally from the `target_detector.py` node. The localization followed by rotation correction are performed in a loop until the aforementioned condition is satisfied.
Another correction being employed is the correction of the end-effector positions. The motor controllers on the real robot can position the joints accurately to the desired joint values with the precision of 0.01 radians. This can still introduce some imprecision as the resulting end-effector poses can differ by low units (1-3) of centimeters from the desired precise poses. A correction has been added to the end of the draw phase to account for this - the real bow pose is obtained and a small adjustment in the orientation and lift joint is performed.
Regarding the arrow trajectory computation, the potential energy of the draw is computed from the measured force-draw curve by a numerical integration using the trapezoid method. To obtain the velocity of the outgoing arrow from the Equation 4.5, the constant $\frac{\eta}{m}$ was roughly estimated by measuring the velocity of the arrow in three shots by a 180 fps iPhone 7 camera.

Chapter 6

Experimental evaluation

This chapter contains the description of the experiments aimed to evaluate the robotic archer. First, in the Section 6.1 the drawing trajectories are evaluated according to their design goal - to require reasonable torques from the motors. Next, Section 6.2 evaluates the accuracy of the target detection by both detectors. Finally, Section 6.3 evaluates the accuracy of the shooting itself.

6.1 Testing drawing trajectories

All 25 designed drawing trajectories have been tested on the real robot by measuring the motor currents during the particular draw/release movement with the bow and arrow mounted. A torque estimate has been produced from the currents as follows:

$$\tau = nKi, \tag{6.1}$$

where τ is a torque of a particular motor, i is the current through the motor, K is a motor torque constant and n is the motor gear ratio. The motor torque constants provided by PAL Robotics are used. More specifically, $K = 0.136$ Nm/A for the first and second joint motors, $K = 0.087$ Nm/A for the third and forth motors and $K = 0.0392$ Nm/A for the three remaining joints. The gear transmission ratios from the Table 3.1 are used.
The currents are obtained from the `/joint_states` ROS topic under "efforts". The ROS node `effort_average.py` located in `tiago_archer/scripts` subscribes to the `/joint_states` topic and produces an average of the currents for each motor every 20 ms. Data obtained like this for every drawing trajectory was saved to a file and later converted to torques according to the Equation 6.1, normalized by the respective nominal torques and plotted. The data from these experiments and the plots are located in `/tiago_archer/efforts_measurements`.
The only trajectory not meeting the requirement of all normalized torques in absolute values under 1.0 is the "pitch 10 number 1" trajectory (as previously mentioned, the trajectories are identified by their pitch angle in degrees and a number to distinguish trajectories with the same pitch angle). More specifically, joints 3 and 4 of the left arm are briefly outside this bound as depicted in Figure 6.1. This is probably caused by the left hand rings missing

the string during the draw and an excess of friction between the string and the rubber glove leading to large torques as the motors try to fight against it. This can be repaired by adjusting the ring positions accordingly. This load on the motors is tolerable for such a short time, but could cause motor overheating if left in the draw state for a few minutes. The right arm torques for this drawing trajectory are, on the other hand, very low with the maximal absolute value of a normalized torque below 0.55 as depicted also in Figure 6.1.

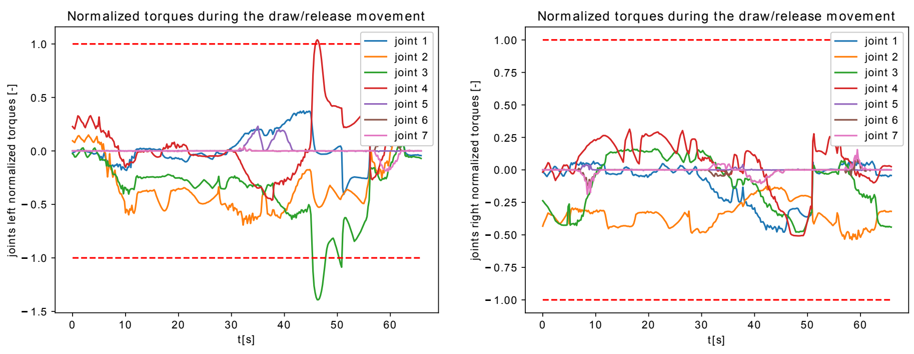

Figure 6.1: Normalized torques of the left and right arm motors for the drawing trajectory "pitch 10 number 1"

The rest of the trajectories do not exceed 0.9 normalized torque in any motor. The drawing trajectory "pitch 0 number 6" even has the normalized torques in absolute value below 0.75 for both arms as depicted in Figure 6.2.

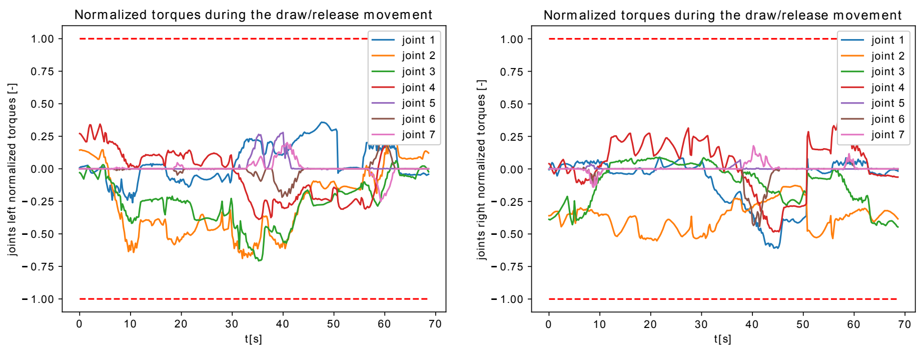

Figure 6.2: Normalized torques of the left and right arm motors for the drawing trajectory "pitch 0 number 6"

A maximum of absolute normalized torques during a particular trajectory in every joint for every drawing trajectory for both arms is provided in Tables 6.1 and 6.2.

	Left arm joints maximal absolute normalized torques [-]							
Trajectory	1	2	3	4	5	6	7	**max**
pitch 0 number 0	0.47	0.62	0.72	0.41	0.18	0.25	0.26	**0.72**
pitch 0 number 1	0.45	0.55	0.52	0.35	0.03	0.25	0.25	**0.55**
pitch 0 number 2	0.43	0.59	0.42	0.39	0.17	0.32	0.33	**0.59**
pitch 0 number 3	0.63	0.46	0.49	0.37	0.01	0.14	0.14	**0.63**
pitch 0 number 4	0.53	0.52	0.46	0.74	0.01	0.33	0.28	**0.74**
pitch 0 number 5	0.42	0.66	0.8	0.35	0.18	0.42	0.3	**0.8**
pitch 0 number 6	0.36	0.69	0.71	0.39	0.28	0.25	0.25	**0.71**
pitch -5 number 0	0.45	0.5	0.32	0.43	0.01	0.38	0.38	**0.5**
pitch -5 number 1	0.28	0.49	0.45	0.38	0.03	0.28	0.27	**0.49**
pitch -5 number 2	0.19	0.65	0.48	0.52	0.06	0.36	0.49	**0.65**
pitch -5 number 3	0.5	0.63	0.57	0.6	0.01	0.22	0.14	**0.63**
pitch 5 number 0	0.43	0.63	0.79	0.41	0.18	0.59	0.29	**0.79**
pitch 5 number 1	0.39	0.59	0.5	0.34	0.01	0.28	0.25	**0.59**
pitch 5 number 2	0.25	0.53	0.6	0.37	0.01	0.26	0.24	**0.6**
pitch 5 number 3	0.5	0.71	0.51	0.58	0.76	0.33	0.33	**0.76**
pitch -10 number 0	0.4	0.59	0.64	0.35	0.18	0.32	0.32	**0.64**
pitch -10 number 1	0.27	0.52	0.25	0.46	0.01	0.34	0.35	**0.52**
pitch -10 number 2	0.28	0.45	0.31	0.41	0.01	0.36	0.38	**0.45**
pitch -10 number 3	0.34	0.39	0.31	0.49	0.01	0.37	0.37	**0.49**
pitch -10 number 4	0.51	0.63	0.72	0.37	0.08	0.43	0.31	**0.72**
pitch 10 number 0	0.4	0.61	0.5	0.41	0.12	0.24	0.24	**0.61**
pitch 10 number 1	0.41	0.69	1.39	1.04	0.23	0.24	0.19	**1.39**
pitch 10 number 2	0.38	0.63	0.55	0.37	0.01	0.21	0.21	**0.63**
pitch 10 number 3	0.27	0.62	0.59	0.37	0.02	0.24	0.2	**0.62**
pitch 10 number 4	0.33	0.71	0.57	0.37	0.01	0.23	0.2	**0.71**

Table 6.1: Left arm joints maximal absolute normalized torques for every drawing trajectory

	Right arm joints maximal absolute normalized torques [-]							
Trajectory	1	2	3	4	5	6	7	**max**
pitch 0 number 0	0.77	0.58	0.49	0.42	0.04	0.28	0.14	**0.77**
pitch 0 number 1	0.88	0.59	0.44	0.48	0.2	0.15	0.14	**0.88**
pitch 0 number 2	0.84	0.59	0.46	0.36	0.14	0.1	0.11	**0.84**
pitch 0 number 3	0.58	0.46	0.65	0.39	0.03	0.16	0.16	**0.65**
pitch 0 number 4	0.51	0.53	0.69	0.37	0.08	0.13	0.13	**0.69**
pitch 0 number 5	0.63	0.57	0.51	0.6	0.05	0.36	0.12	**0.63**
pitch 0 number 6	0.61	0.55	0.45	0.48	0.07	0.44	0.18	**0.61**
pitch -5 number 0	0.57	0.71	0.83	0.4	0.23	0.09	0.09	**0.83**
pitch -5 number 1	0.71	0.58	0.44	0.49	0.21	0.14	0.14	**0.71**
pitch -5 number 2	0.15	0.58	0.53	0.4	0.04	0.14	0.2	**0.58**
pitch -5 number 3	0.64	0.54	0.66	0.32	0.23	0.13	0.17	**0.66**
pitch 5 number 0	0.74	0.57	0.54	0.38	0.43	0.23	0.23	**0.74**
pitch 5 number 1	0.74	0.57	0.49	0.44	0.41	0.19	0.19	**0.74**
pitch 5 number 2	0.65	0.6	0.46	0.39	0.02	0.19	0.16	**0.65**
pitch 5 number 3	0.52	0.6	0.54	0.48	0.16	0.21	0.21	**0.6**
pitch -10 number 0	0.71	0.65	0.5	0.41	0.33	0.18	0.12	**0.71**
pitch -10 number 1	0.56	0.5	0.69	0.44	0.1	0.1	0.1	**0.69**
pitch -10 number 2	0.58	0.57	0.69	0.41	0.09	0.12	0.11	**0.69**
pitch -10 number 3	0.63	0.83	0.77	0.39	0.13	0.12	0.12	**0.83**
pitch -10 number 4	0.84	0.6	0.72	0.63	0.05	0.16	0.16	**0.84**
pitch 10 number 0	0.72	0.58	0.63	0.37	0.35	0.19	0.19	**0.72**
pitch 10 number 1	0.48	0.54	0.48	0.51	0.04	0.13	0.19	**0.54**
pitch 10 number 2	0.75	0.59	0.43	0.48	0.19	0.09	0.09	**0.75**
pitch 10 number 3	0.7	0.61	0.42	0.38	0.06	0.16	0.13	**0.7**
pitch 10 number 4	0.64	0.63	0.6	0.41	0.02	0.14	0.16	**0.64**

Table 6.2: Right arm joints maximal absolute normalized torques for every drawing trajectory

6.2 Testing target detection

The accuracy of the target detection has been tested by comparing it with a very precise localization obtained from the Vicon Tracker [vic] system available inside the laboratory, where the archer has been developed. This localization involves attaching retroreflective markers to the localized objects - in this case the robot and the target - and tracking these markers by infrared cameras.
The target was during the testing placed on seven different positions in the field of view of the robot's camera. After reaching a new position, the detectors were let to "stabilize" for a while. After both the averaged concentric circles target position and the neural network target position already corresponded

to the new target location, both these positions were saved to a file together with multiple concentric circles raw detections and the ground-truth Vicon localization. This task was performed by `save_target_detection_data.py` node in `tiago_archer/scripts`. The resulting positions for each target location and the differences of each detector estimate from the ground truth are summarized in Table 6.3. These positions are measured with respect to the robot base link frame, the camera was in its default orientation with its z-axis aligned with the base link x-axis. A mean value and a corrected sample standard deviation are provided for raw concentric circles measurements. Its high variance justifies the use of averaging.
Best accuracy was achieved, as expected, for a closer target near the center of the image plane as indicated by the detection of the target in locations 2 and 3 in the Table 6.3. The localization phase of the detect-aim-shoot loop consequently features the robot's head constantly pointing at the last concentric circles raw detection position to center the image plane at the target.
The output of the localization phase of the detect-aim-shoot loop is the average of the concentric circles averaged target position and the neural network target position, which improves the accuracy as well. This is indicated by the mean of the absolute difference between Vicon measurements and the three position estimates - the concentric circles averaged, the neural network and their average - computed across the seven target locations:

$$\begin{aligned}\Delta_{abs,CCA} &= \begin{bmatrix}0.20 \pm 0.20\\ 0.057 \pm 0.027\\ 0.102 \pm 0.087\end{bmatrix},\\ \Delta_{abs,NN} &= \begin{bmatrix}0.20 \pm 0.13\\ 0.087 \pm 0.051\\ 0.038 \pm 0.049\end{bmatrix},\\ \Delta_{abs,av} &= \begin{bmatrix}0.123 \pm 0.079\\ 0.058 \pm 0.018\\ 0.053 \pm 0.045\end{bmatrix},\end{aligned} \tag{6.2}$$

where $\Delta_{abs,CCA}$, $\Delta_{abs,NN}$ and $\Delta_{abs,av}$ are the mean values of absolute differences between Vicon and concentric circles averaged estimates, neural network estimates and their average, respectively.

target location #	Vicon [m]	$\Delta_{CCR}[m]$	$\Delta_{CCA}[m]$	$\Delta_{NN}[m]$	$\Delta_{av}[m]$
1	5.270 -0.002 0.282	0.31 ± 0.36 0.081 ± 0.095 -0.15 ± 0.18	0.375 0.076 -0.166	0.181 0.087 -0.113	0.278 0.081 -0.139
2	3.691 0.031 0.279	-0.042 ± 0.027 0.056 ± 0.036 -0.063 ± 0.041	-0.038 0.056 -0.066	-0.167 0.052 -0.010	-0.103 0.054 -0.038
3	2.913 0.038 0.276	-0.090 ± 0.043 0.040 ± 0.095 -0.038 ± 0.018	-0.091 0.038 -0.037	0.002 0.048 0.011	-0.044 0.043 -0.013
4	3.286 0.681 0.278	-0.26 ± 0.23 -0.003 ± 0.002 0.026 ± 0.023	-0.144 0.041 -0.023	-0.105 0.037 -0.007	-0.124 0.039 -0.015
5	3.070 -0.844 0.274	0.58 ± 0.44 -0.101 ± 0.076 -0.27 ± 0.20	0.572 -0.096 -0.266	-0.412 0.179 0.106	0.080 0.042 -0.080
6	4.848 -1.019 0.276	-0.118 ± 0.086 0.112 ± 0.081 -0.033 ± 0.024	-0.061 0.016 -0.048	-0.268 0.132 -0.003	-0.164 0.074 -0.025
7	5.507 0.487 0.279	0.112 ± 0.094 0.102 ± 0.085 -0.107 ± 0.089	0.122 0.074 -0.110	-0.250 0.074 -0.018	-0.064 0.074 -0.064

Table 6.3: Position estimates (x, y and z coordinates) for 7 different positions with respect to the robot base link frame - ground-truth position in Vicon column, in other columns differences from Vicon of concentric circles raw (Δ_{CCR}), concentric circles averaged (Δ_{CCA}), neural network (Δ_{NN}) and an average between CCA and NN (Δ_{av})

6.3 Detect - Aim - Shoot loop testing

The detect-aim-shoot loop has been evaluated by letting the robot go through the whole process - from detection to shot - 15 times for two distances of the robot to the target, more specifically 4 and 4.5 m. The detect-aim-shoot loop for each measurement was started with the robot in an arbitrary orientation with respect to the target. The target stayed at the same location for all measurements and the position of the robot was also the same when measuring for a single distance.

The arrows often bounced out instead of sticking to the target, each shot thus has been recorded on a video and the approximate positions of the arrow impacts are depicted in Figure 6.3. The arrows sticking to the target are marked by green circles and the ones that bounced out are marked by yellow circles. A purple star marks the average position. The x and y deviations (x oriented to the right of the Figure 6.3 and y up) and the distance from the target center of each arrow hit are summarized in Table 6.4.
The figures demonstrate lower variance at closer distance, although the average position is further from the target center. Mainly, the low average height of the hits at 4 m distance is responsible for this. This is probably caused by the system using more draw configurations with downward pitch for the closer distance. The height accuracy of the shots from these draw configurations is more sensitive to the correct setting of the $\frac{\eta}{m}$ parameter, which has been only roughly estimated here.
At both distances, there is still a notable bias to the left caused probably by the rotation error due to the robot localization imprecision.
The arrows bounce out much more when further from the target as a larger portion of their energy has been dissipated (due to aerodynamic forces) and the orientation of the arrow with respect to the target on hit influences more whether the arrow sticks or not.

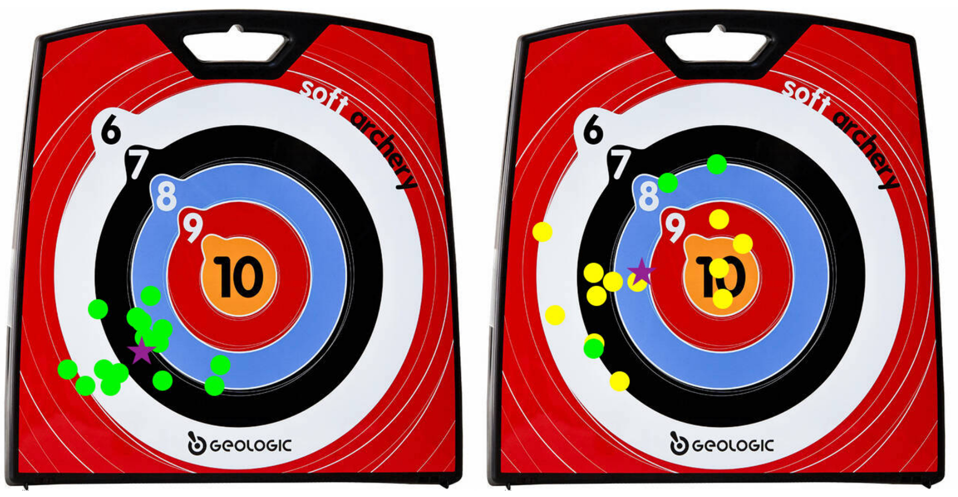

Figure 6.3: Target with arrow hits marked as green circles for sticked arrows and yellow circles for bounced arrows, distance 4 m (left) and 4.5 m (right) [bow]

	robot-target 4 m distance			robot-target 4.5 m distance		
shot #	x[m]	y[m]	distance[m]	x[m]	y[m]	distance[m]
1	-0.241	-0.131	0.274	-0.002	0.012	0.012
2	-0.214	-0.154	0.264	0.003	-0.031	0.032
3	-0.180	-0.154	0.237	0.032	0.046	0.056
4	-0.168	-0.135	0.216	-0.002	0.081	0.081
5	-0.188	-0.132	0.230	-0.005	0.158	0.158
6	-0.198	-0.046	0.203	-0.074	0.132	0.151
7	-0.106	-0.146	0.181	-0.118	-0.007	0.118
8	-0.033	-0.154	0.157	-0.152	-0.007	0.152
9	-0.025	-0.126	0.128	-0.175	-0.027	0.177
10	-0.143	-0.058	0.154	-0.179	0.005	0.179
11	-0.136	-0.065	0.151	-0.143	-0.146	0.205
12	-0.133	-0.087	0.159	-0.180	-0.101	0.207
13	-0.112	-0.095	0.147	-0.178	-0.093	0.201
14	-0.107	-0.073	0.129	-0.233	-0.054	0.240
15	-0.123	-0.026	0.126	-0.252	0.062	0.260
μ	**-0.141**	**-0.105**	**0.184**	**-0.111**	**0.002**	**0.149**
s	**0.061**	**0.044**	**0.050**	**0.095**	**0.085**	**0.075**

Table 6.4: x and y deviations and distances from the center for the arrow hits for both robot-target distances, mean μ of the values and their corrected sample standard deviation s

Chapter 7

Conclusion and Discussion

The whole process of shooting with a bow at a target is a complex task consisting of many subtasks, which would each deserve a separate thesis to do them justice. As a result, this thesis was more focused on certain lower level subtasks like the search for optimal draw configurations and the design of respective bow drawing trajectories for these configurations. For the other subtasks, like the target detector and detect-aim-shoot loop, an easy but notably suboptimal solution has been implemented.
The robot has fulfilled the main goal of this thesis of shooting at a detected target with a bow, but its accuracy, range and arrow cadence have still a lot of room for improvement.
The inaccuracy observed is a product of multiple factors. First, the bow model used for the computation of the vertical trajectory of the arrow is imprecise, the parameter $\frac{\eta}{m}$ needs to be tuned better as mentioned earlier. This could be done by measuring the velocity of the arrow leaving the bow for numerous shots and different pitch angles of the bow and estimating $\frac{\eta}{m}$ to fit the data best. The model of the shot also does not account for any aerodynamic forces. These are not so significant for close distance shooting like was performed here, but would become notable if the robot was made to shoot at a longer distance, especially the drag is probably high on the arrow due to its suction cup tip. Better grasping and releasing of the bowstring by the draw hand is also needed for the accuracy to improve as the string during release often slips from the rings to the last finger links before finally leaving the draw hand, because the hand has not opened fast enough. This can be repaired by shifting the rings closer to the fingertips and modifying the drawing trajectories accordingly. Some of the draw configurations also result in the string flicking the right arm a little in the last part of its trajectory during the shot. This, similarly to the previous problem, results in the string and by extension the arrow trajectory during the shot deviating from their optimal path - a straight line. The arrow then can bump to the bow as it is leaving - this leads to the arrow yawing away from the trajectory of its center of mass (which is not influenced by this bump) during its flight, decreasing its accuracy and increasing the chances of bouncing from the target. The imprecision introduced by the target localization has been already discussed. This can be potentially improved by using a higher resolution (but lower

Fps) camera mode on the robot, especially for bigger target distances. The biggest accuracy burden is, as mentioned before, the imprecision of rotation estimation by the AMCL. An aiming process focusing more on the optical alignment with the target could be used to rectify this. Such aligning is currently used only to correct large rotation estimation errors as it is time-consuming to perform, because it involves repeating the localization process multiple times.
This leads to the final area of improvement for the system: it is very slow. One complete detect-aim-shoot process can take up to 5 minutes due to the localization trying to reach consensus between the concentric circles detector and the neural network, which takes up to 9 s to process one image. An obvious improvement here is to run this detector on a GPU, which would decrease the computation time by several orders of magnitude.

Appendix A

Bibliography

[amc] *AMCL ROS wiki*, http://wiki.ros.org/amcl, [Online; accessed May 21, 2021].

[Arr] Arrowsoft Sports, https://arrowsoftsports.com/how-to-shoot-a-recurve-bow/, [Online; accessed May 21, 2021].

[Asa05] H. Harry Asada, *Introduction to robotics*, https://ocw.mit.edu/courses/mechanical-engineering/2-12-introduction-to-robotics-fall-2005/lecture-notes/chapter6.pdf, 2005, [Online; accessed May 21, 2021].

[ble] *Blender*, https://www.blender.org/, [Online; accessed May 21, 2021].

[BMK01] G. Sohl E. Wang F. C. Park B. Martin, J. E. Bobrow and J. Kim, *Optimal robot motions for physical criteria*, Journal of Robotic systems **18** (2001), no. 12, 785–795.

[bow] *Soft Archery Bow - Decathlon*, https://www.decathlon.cz/sada-softarchery-100-id_8505629.html, [Online; accessed May 21, 2021].

[CF17] Changrak Choi and Emilio Frazzoli, *Torque efficient motion through singularity*, 2017 IEEE International Conference on Robotics and Automation (ICRA), 2017, pp. 5012–5018.

[CNH47] Paul E. Klopsteg C. N. Hickman, Forrest Nagler, *Archery: The technical side*, https://www.archerylibrary.com/books/hickman/archery-the-technical-side/, 1947, [Online; accessed May 21, 2021].

[dat] *Neural network - diagram*, https://databricks.com/glossary/neural-network, [Online; accessed May 21, 2021].

[det] *Detecto documentation*, https://detecto.readthedocs.io/, [Online; accessed May 21, 2021].

[DL19] M. Cabatuan C. Llorente E. Dadios D. Ligutan, A. Abad, *FPGA implementation of archery target detection using color sequence recognition algorithm*, International Journal of Recent Technology and Engineering **8** (2019), 1391–1397.

[fls] *Front Line Solvers*, https://www.solver.com/nonsmooth-optimization, [Online; accessed May 21, 2021].

[gaz] *Gazebo simulator*, http://gazebosim.org/, [Online; accessed May 21, 2021].

[GDDM13] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik, *Rich feature hierarchies for accurate object detection and semantic segmentation*, CoRR **abs/1311.2524** (2013).

[Gir15] Ross B. Girshick, *Fast R-CNN*, CoRR **abs/1504.08083** (2015).

[GJ$^+$10] Gaël Guennebaud, Benoît Jacob, et al., *Eigen v3*, http://eigen.tuxfamily.org, 2010.

[Its14] Itseez, *The OpenCV Reference Manual*, 2.4.9.0 ed., April 2014, [Online; accessed May 21, 2021].

[Jal19] Marek Jalůvka, *Playing chess with KUKA robot using linguistic instructions*, Bachelor thesis, CTU in Prague, https://dspace.cvut.cz, 5 2019.

[JD20] Jian Li Jiarong Du, Ru Lai, *Vision-based automatic archery target reporting system*, Proc. of the 9th International Symposium on Computational Intelligence and Industrial Application (Beijing, China), ISCIIA2020, 2020.

[Joh] Steven G. Johnson, *The NLopt nonlinear-optimization package*, http://github.com/stevengj/nlopt, [Online; accessed May 21, 2021].

[mer] *Force sensor*, https://www.colosus.cz/merak-sily-natahu-luku-x151274, [Online; accessed May 21, 2021].

[mgi] *moveit Namespace Reference*, http://docs.ros.org/en/melodic/api/moveit_ros_planning_interface/html/namespacemoveit.html, [Online; accessed May 21, 2021].

[mova] *MoveIt*, https://moveit.ros.org/, [Online; accessed May 21, 2021].

[movb] *MoveIt concepts*, https://moveit.ros.org/documentation/concepts, [Online; accessed May 21, 2021].

[ope19] *OpenCV: Camera Calibration and 3D Reconstruction*, https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html, 2019, [Online; accessed May 21, 2021].

[oro] *Orocos kinematics and dynamics*, https://www.orocos.org/kdl.html, [Online; accessed May 21, 2021].

[pala] *PAL Robotics*, https://pal-robotics.com/robots/tiago, [Online; accessed May 21, 2021].

[PALb] PAL Robotics, *TIAGo++ Handbook*.

[PK10] Ryo Saegusa Giorgio Metta Petar Kormushev, Sylvain Calinon, *Learning the skill of archery by a humanoid robot iCub*, Proc. of the IEEE-RAS International Conference on Humanoid Robots (Nashville, TN, USA), 2010.

[pla] *planning_scene::PlanningScene Class Reference*, http://docs.ros.org/en/jade/api/moveit_core/html/classplanning__scene_1_1PlanningScene.html, [Online; accessed May 21, 2021].

[RHGS15] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun, *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*, CoRR **abs/1506.01497** (2015).

[roba] *moveit::core::RobotModel Class Reference*, http://docs.ros.org/en/indigo/api/moveit_core/html/classmoveit_1_1core_1_1RobotModel.html, [Online; accessed May 21, 2021].

[robb] *moveit::core::RobotState Class Reference*, http://docs.ros.org/en/jade/api/moveit_core/html/classmoveit_1_1core_1_1RobotState.html, [Online; accessed May 21, 2021].

[ros] *Robot Operating System Wiki*, http://wiki.ros.org, [Online; accessed May 21, 2021].

[Rub20] Hema Rubesh, *Interactive Color Image Segmentation using HSV Color Space*, Science Technology Journal (2020), 37–41.

[RY00] Thomas P. Runarsson and Xin Yao, *Stochastic ranking for constrained evolutionary optimization*, IEEE Trans. Evolutionary Computation **4** (2000), no. 3, 284–294.

[Sto] Rebecca Stone, *Image Segmentation Using Color Spaces in OpenCV + Python*, https://realpython.com/python-opencv-color-spaces/, [Online; accessed May 21, 2021].

[The] TheAILearner, *Hough Gradient Method*, https://theailearner.com/tag/hough-gradient-method/, [Online; accessed May 21, 2021].

[vic] *Vicon*, https://www.vicon.com/software/tracker/, [Online; accessed May 21, 2021].

[YR05] Xin Yao and Thomas P. Runarsson, *Search biases in constrained evolutionary optimization*, IEEE Trans. on Systems, Man, and Cybernetics Part C: Applications and Reviews **35** (2005), no. 2, 233–243.