Università degli Studi di Milano-Bicocca

Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di laurea Magistrale in Informatica

USI Università della Svizzera Italiana

Faculty of Informatics

Master in Informatics

Implementation of a Laser Turret with 2 Degrees of Freedom, and Applications for Human-Robot Interaction

Supervisor: Prof. Dr. Luca Maria Gambardella

Co-Supervisor: Dr. Alessandro Giusti

Co-Supervisor: Dr. Boris Gromov

Master Thesis of:

Gabriele Abbate
Student Number 761890

Academic Year 2017-2018

Contents

List of Figures

Listings

Introduction

This thesis describes the development and applications of a laser turret with pan and tilt control: this device can be used to project a laser dot on a given surface (wall and/or floor) and finely control its position by solving the system's inverse kinematics.
Once its functionality was validated, we used the turret for an human-robot interaction task. In particular, we considered an existing system in which an operator interacts with a drone using pointing gestures [1]; the system initially determines the relative localization between the two, then allows the operator to control the drone, which follows the indicated location in real time. The existing approach relied on a fast agile robot, and was unsuitable for implementation on slower or larger ground robots. In this thesis, we demonstrate how the turret can be used with this goal. Figure 1 gives an idea.
Finally, the turret was adopted to efficiently run experiments for fine tuning or validating different components of the system described above, such as the algorithms for relative localization and the algorithms for reconstruction of the pointed direction. To this end, we ran an experimental campaign involving ten users.

Motivations and Related Works

Many researches in the field of Human-Robot Interaction (HRI) involve the study of interactions between an operator deployed alongside a robot in an environment they share. That means that a direct-line of sight exists between the two. Many interfaces have been proposed, ranging from standard joysticks (e.g. for low-level control of UAVs) to hands-free gesture-based interfaces based on sensorized armbands [2], bracelets [3, 4], smartwatches [5] or voice commands [6].
The work developed in that thesis is collocated in that context, thus, an overview of related works is useful to understand motivations that stand behind that project.

Proximity HRI

Proximity interaction techniques can take advantage of *pointing gestures* to intuitively express locations or objects with minimal cognitive overhead; this modality has been often used in HRI research e.g. for pick-and-place tasks [7, 8, 9, 10], labeling and/or querying information about objects or locations [7, 11, 12], selecting a robot within a group [13, 14], and providing navigational goals [15, 16, 2, 17, 6, 18, 4].

Pointing Based HRI Interface

First, a differentiation can be made on sensing with respect to the type of sensors involved and their locations with respect to the human: it can be either *external* or based on *wearable* devices. External sensing is usually based on vision systems, like regular RGB

Figure 1: System Deployed to Drive a Ground Robot

cameras [11, 19], stereo cameras [20], structured light depth cameras [10] and time-of-flight (ToF) cameras [8]. The wearable sensing is typically realized either with inertial [2, 21] or magnetic measurements [22, 20].
Our work is an example of wearable sensing, as we exploit a wearable Inertial Measurement Unit (IMU) device. We are interested in such type of sensing because, even if vision-based methods are able to capture rich information about the objects, such as velocity, shape, size and color, they are susceptible to poor lighting conditions, low spatial resolution and may demand high computational resources. Thus, for proximity interaction, wearable sensing is a very interesting possibility.
In particular, the use of pointing gestures constitutes a very practical and profitable solution, being based on mechanics which are natural for humans. This is why pointing gestures solutions are largely deployed to solve HRI problems and significant research efforts have been devoted to this topic. In fact, using pointing gestures as an input interface dates back to 1980s, when Bolt presented his now famous work "Put-that-there" [22]. In the HRI research, pointing gestures are often used for pick-and-place tasks [7, 8, 9, 10], labeling and/or querying information about objects or locations [7, 11, 12], selecting a robot within a group [13, 14], and providing navigational goals [15, 16, 2, 17, 6, 18].
Providing navigational goals is exactly the field in which our system finds one of its main applications and this is why we are now going to mention more works related to that task. First, however, it is worth to say that one important issue to be solved in natural human-robot interaction that involves pointing is a perception of the user's gestures. This can be a responsibility of a robot, i.e. the recipient of the message, as well as of a group of cooperatively-sensing robots [23, 14]; of the environment [24]; or, as in our case, of a device worn by the user [21, 2, 6]. The first approach is the most popular in HRI research. However, it presents important challenges to solve the perception problem, and requires the robot to consistently monitor the user. Relying on sensors placed in the environment relaxes the requirements on the robots, but limits the applicability of

the system to properly instrumented areas; in both cases, the positions being pointed at need to be inferred by external observation, which is typically performed with cameras or RGB-D sensors.

Providing Navigational Goals

Van Den Bergh et al. [15] used pointed directions to help a ground robot to explore its environment. The robot continuously looks for a human in its vicinity and once detected begins the interaction. Using an RGB-D sensor (Microsoft Kinect) the system detects human's hand and wrist. A vector connecting the center of the hand and the wrist is then projected on the ground, giving a principal exploration direction. Finally, the next exploration goal is automatically selected from a set of possible goals with respect to an instantaneous occupancy grid acquired by the robot.
Similarly to the previous work, Abidi et al. [16] use a Kinect sensor to extract pointed directions. Navigation goals are continuously sent to the ground robot, which reactively plans its motion and thus allows the user to correct his input on the fly. The main drawback, however, is that the robot has to "keep an eye" on the user in order to reach the final target. To estimate pointed locations authors suggest two approaches: (1) a vector originating from the elbow and passing the hand/finger, and (2) a vector originating from the eyes and also passing the hand/finger.
Jevtic at al. [17] experimentally compared several interaction modalities in the user study of 24 participants. A ground robot equipped with a Kinect and other sensors was used. The study compares three interaction modalities: Direct Physical Interaction (DPI), person following, and pointing control in area- and waypoint-guidance tasks. The DPI modality requires the user to push the robot by hands, the torques generated at motors are measured via electrical current and then are fed to a friction-compensation controller that drives the robot in the appropriate direction. The person following modality makes the robot to follow the user at a safe distance, the user can stop the robot at any time by raising her left hand above the left elbow and thus can control the robot's precise location. The pointing modality allows the user to command the robot's position with a pointing gesture, where the target location is calculated from the intersection of the ground plane with a line passing through the right elbow and the right hand of the user. The authors measured task completion times, accuracy, and workload (with NASA-TLX questionnaire). Reported results show that the DPI modality is systematically better than the other modalities for all the metrics, while the pointing control shows the worst results.
Such a low performance of the pointing interface can be explained by a lack of appropriate feedback and a time-sparse nature of the implemented gesture control: the user issues a single command to drive the robot to a goal and see where the system "thinks" he was pointing at only when the robot reaches the target, therefore, the user is unable to efficiently correct the robot's position. This problem is nicely solved by our system, as it provides a real-time feedback to user pointing with a laser pointer, making him able to understand where the system thinks he is pointing and also correct any misalignment.
Problems reported in the study by Jevtic at al. [17] are further aggravated by an inherently limited precision of a chosen pointing model (elbow-hand). As reported by many other works (see [16, 25, 8]), including those from the psychology research (see [26, 27]), a more appropriate model would be a line that passes through the head and the fingertip. This is exactly the model we use for that thesis.

Wearable Sensors

An alternative approach to a perception of pointing gestures is wearable sensors, as we mentioned before.
Sugiyama et al. [21] developed a wearable visuo-inertial interface for on-site robot teaching that uses a combination of monocular camera and IMU to capture hand gestures in the egocentric view of the user. The camera is also used for a monocular Simultaneous Localization and Mapping (SLAM), which allows to localize the user with respect to a common with the robot coordinate frame.
Wolf et al. [2] suggest a gesture-based interface for a robot control, that is based on a device they call BioSleeve, a wearable device placed on the user's forearm and comprised of a set of dry-contact surface electro-myography sensors (EMGs) and an IMU. Optionally, the authors suggest to strap an additional IMU sensor on the upper arm to be able to perform a model-based arm pose reconstruction for pointing gestures. However, no information is given on how a user would localize herself with respect to the robot in order to control its position.
A work by Villan et al. [5] suggests to use a single smartwatch device to control a drone. The system provides two interfaces: high-level commands and velocity commands.
Cacace et al. [3] demonstrate a multi-modal human-robot interface used for interaction in search and rescue missions. The operator is equipped with a Myo armband used for gestures, headset for voice commands, and a tablet with a touch screen.

Document Structure

The rest of the document is structured in chapter as follows:

- **Chapter 1 - Models Specification** All the geometric models and formulas on which the system is based are explained. This includes turret model, human pointing model and relative localization procedure;
- **Chapter 2 - Hardware Implementation** We introduce all the hardware components involved in the development of the presented system: servo motors and interface board for the turret, arm IMU devices and the ground robot used for demonstrations;
- **Chapter 3 - Software Implementation** We give an overview of the main libraries used and then describe the entire project software implementation;
- **Chapter 4 - Experiments and Applications** We describe the experiments the turret system allowed us to perform and show possible application scenarios developed;
- **Chapter 5 - Conclusion and Future Work** We make a brief recap of the thesis, draw some conclusions and report eventual further improvements or possible works;
- **Appendix A - Interesting Stuff** Here we collect a couple of things that did not find a place in the main document, but can be interesting to mention;
- **Appendix B - Code Listing** Relevant code listings.

Chapter 1

Models Specification

In this chapter we describe the abstract models used to shape all the three parts of the system from a geometric perspective. That means:

- the turret model;
- the human pointing model;
- the relative localization procedure.

1.1 Turret Model

Degree of Freedom (DoF) is the number of independent parameters that define the configuration of a mechanical system. In that thesis, we present a two DoF Pan & Tilt turret. That means that our parameters are two angles. In a 3D reference system, **pan** is the horizontal angle about the upright Z axis, **tilt** is the vertical angle about the rotated Y axis, as in figure 1.1. Our final goal is to be able to define the direction of a laser ray mounted on top of the turret, so that we can control the position of the projected laser dot on a given surface, by solving the system's inverse kinematics.
For those purposes, we have built two different turrets. Since the model of the first one is slightly simpler than the second, we will start describing the former, which turns out to be helpful to understand the latter. We will focus on the case in which the laser must be projected on the ground.

1.1.1 First Model

First, figures 1.2 helps to understand how the model is shaped to match the physical structure of the turret. We have three reference frames. The **base_frame** is fixed and is the one in which we define the coordinates of the projected point. **pan_frame** and **tilt_frame** are the frames used to represent our revolute joints. **H** is the height of the turret, which is known. Note that the convention used for the frame is the following:

- red is the x axis;
- green is the y axis;
- blue is the z axis.

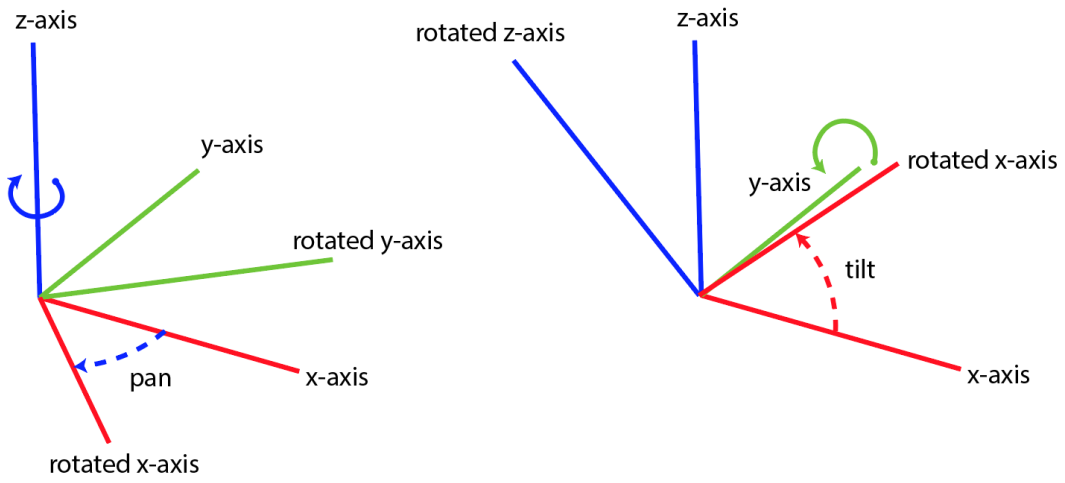

Figure 1.1: *left*: Pan Angle Explained; *right*: Tilt Angle Explained

Thus, we consider the laser ray to be a prolongation of the x axis of the **tilt_frame**. Figure 1.3 shows what we want to be able to do: given the x, y and z of the laser point we want to set **pan** and **tilt** angles accordingly. In order to do so, firstly we will solve the forward kinematics, then the inverse will be easily derived.

Forward Kinematics

The forward kinematics should take as input our parameters (i.e. **pan** and **tilt**) and then return the coordinates of the laser projected point into **base_frame** reference. Note that, since we are controlling the direction of an infinite ray, in order to obtain a unique (x, y, z) triple, we must intersect such ray with a plane defined by the triple (0, 0, z). In the forward kinematics equations this can be obtained by assuming that we know the z of the point we want. Another option could be to assume that z is zero, since we are considering the laser projection on the floor (i.e. on the **base_frame**).
As well as what is already defined in figure 1.3, we must add:

- **L** as the distance from the **pan_frame** origin to the projection of the laser point on the **base_frame**;
- **D** as the distance from the **tilt_frame** origin to the laser point.

First, note that the **pan** angle does not depends on the z coordinate, so, starting from:

$$D = \frac{H - z}{\cos(tilt)} \tag{1.1}$$

$$L = \sqrt{(H - z)^2 + D^2} \tag{1.2}$$

We can easily obtain laser point coordinates:

$$x = L\cos(pan) \tag{1.3}$$

$$y = L\sin(pan) \tag{1.4}$$

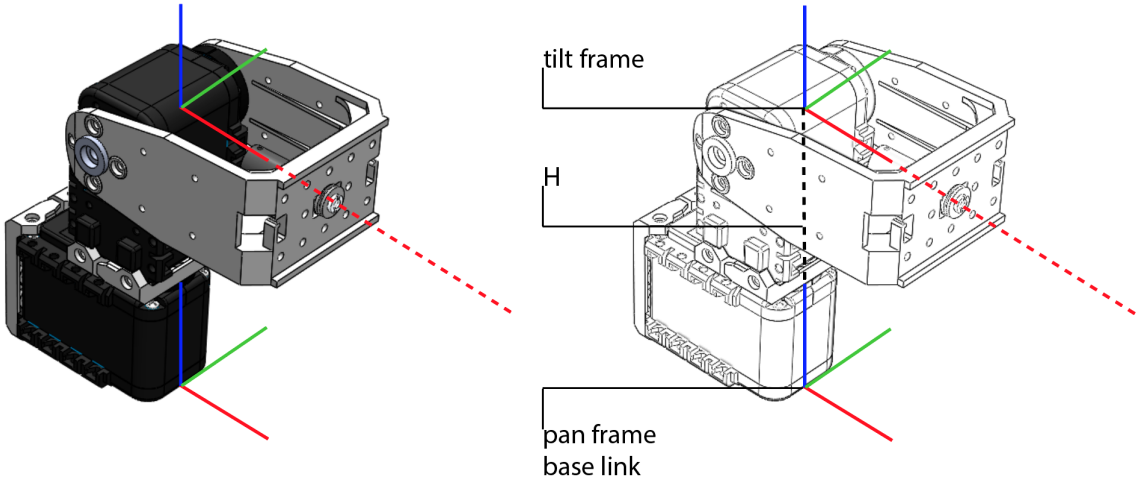

Figure 1.2: First Model, Reference Frames Explained

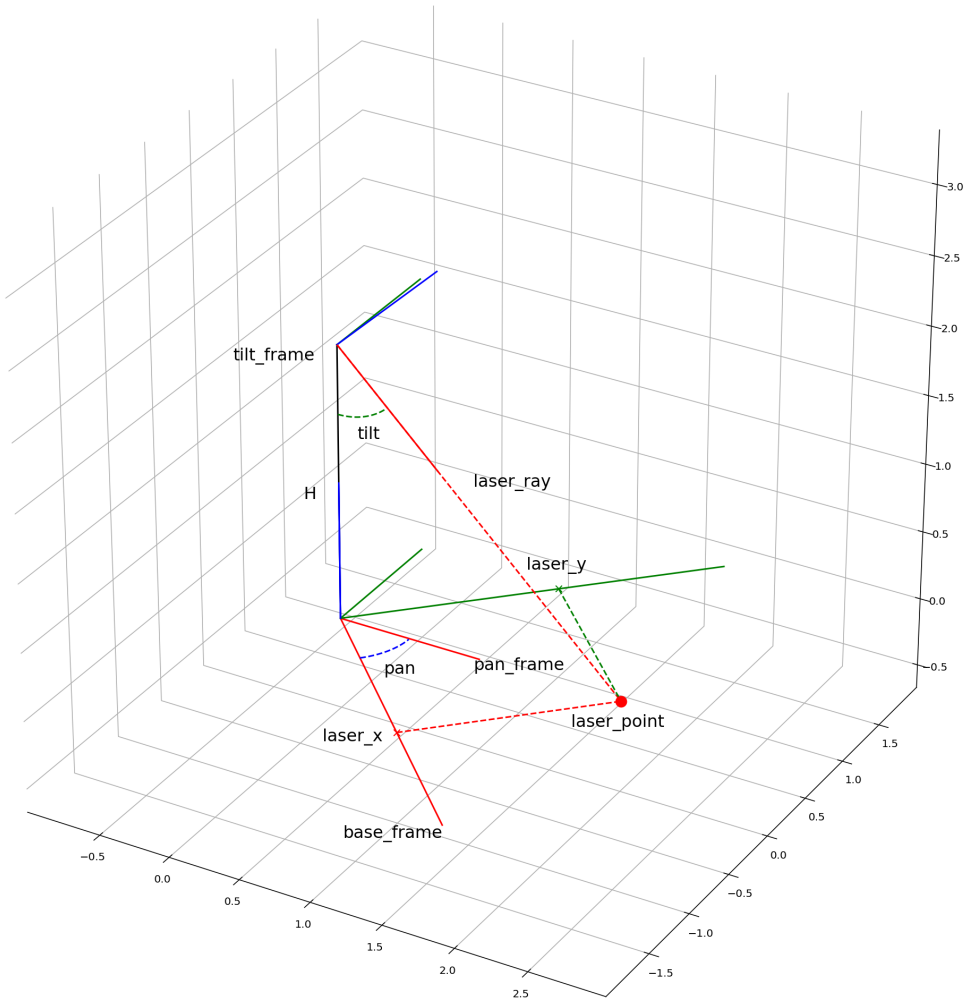

Figure 1.3: First Model, Drawing to Explain the Kinematics

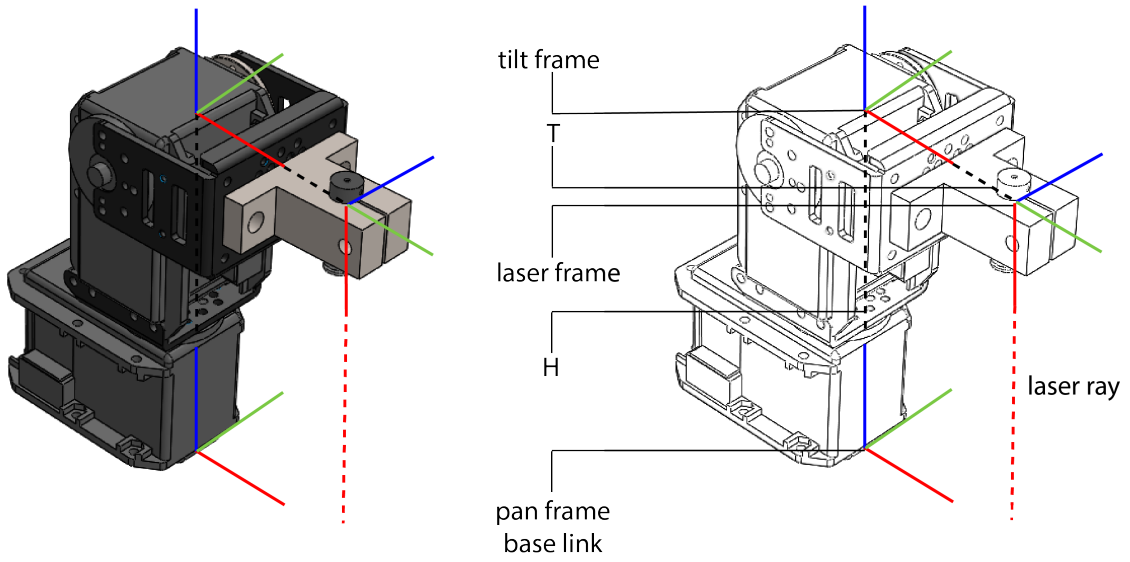

Figure 1.4: Second Model, Reference Frames Explained

Inverse Kinematics

The inverse kinematics takes as input the triple $(x,\ y,\ z)$ of the laser point and returns the corresponding **pan** and **tilt** angles.
In addition to 1.1, we can say that:

$$H - z = D\cos(tilt) \tag{1.5}$$
$$L = D\sin(tilt) \tag{1.6}$$
$$L = \sqrt{x^2 + y^2} \tag{1.7}$$

Thus, we have that:

$$tilt = \arctan\left(\frac{L}{H - z}\right) \tag{1.8}$$
$$\tag{1.9}$$

Finally, thanks to equations 1.3 and 1.4 we can immediately obtain:

$$pan = \arctan\left(\frac{y}{x}\right) \tag{1.10}$$

1.1.2 Second Model

The second model is slightly different from the first, as we can see in figure 1.4. In that case, we have the laser ray which is perpendicular to the x axis of the **tilt_frame**. One could think that to solve the inverse kinematics, adding 90 degrees to the tilt angle could be enough. However, since the ray origin does not coincide with the origin of the frame, this is wrong. Changing the tilt angle will not change only the direction of the ray, but also the position of its origin. This makes the kinematics a bit more complicated for that model.
Here we report only the inverse kinematics as it is the most interesting to understand. Note that the assumption made for the z of the laser point in section 1.1.1 still holds.

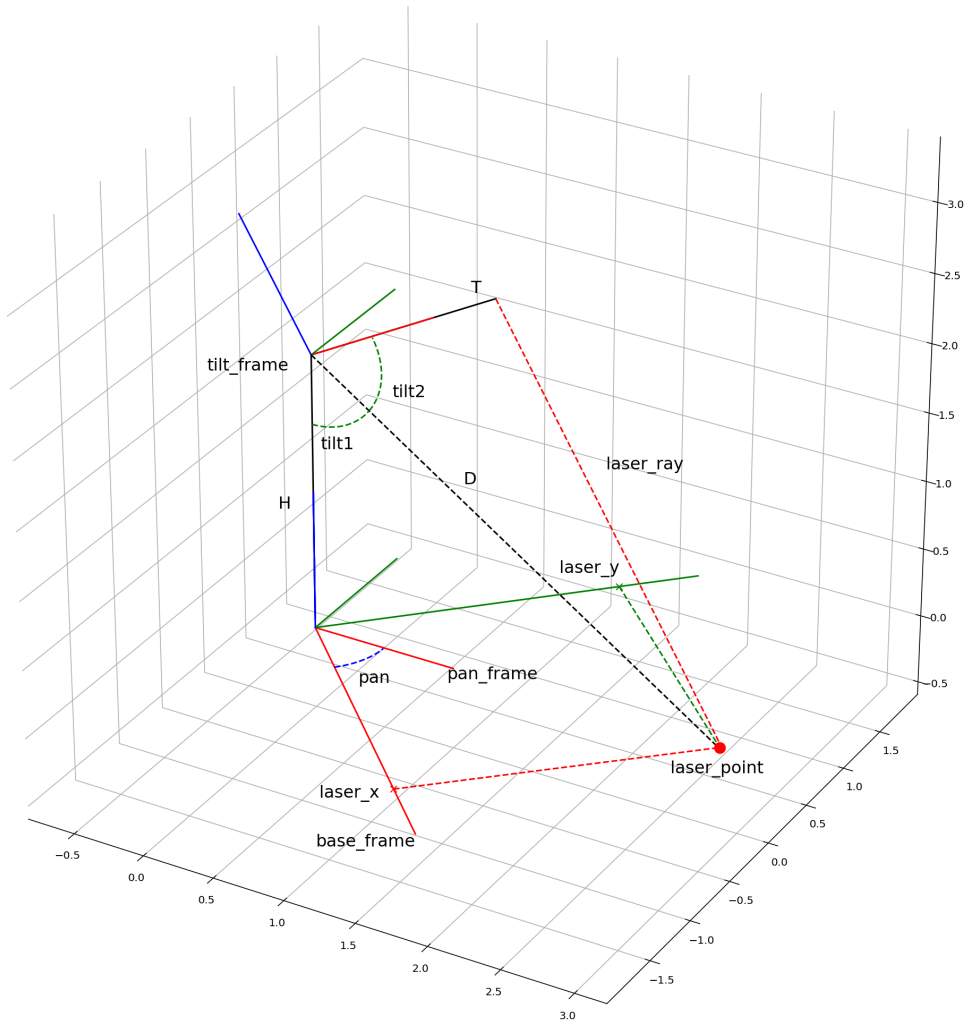

Figure 1.5: Second Model, Drawing to Explain the Kinematics

Inverse Kinematics

First, we recall **L**, **D** and we introduce **T**:

- **L** as the distance from the **pan_frame** origin to the projection of the laser point on the **base_frame**;
- **D** as the distance from the **tilt_frame** origin to the laser point;
- **T** as the known distance from the **tilt_frame** origin to the laser ray origin.

As we can see in figure 1.5, we have to decompose **tilt** in two parts, this is why we explicitly draw **D** that time. On the contrary, **pan** is obtained in the same way we did for the first model, thus:

$$pan = \arctan\left(\frac{y}{x}\right) \tag{1.11}$$

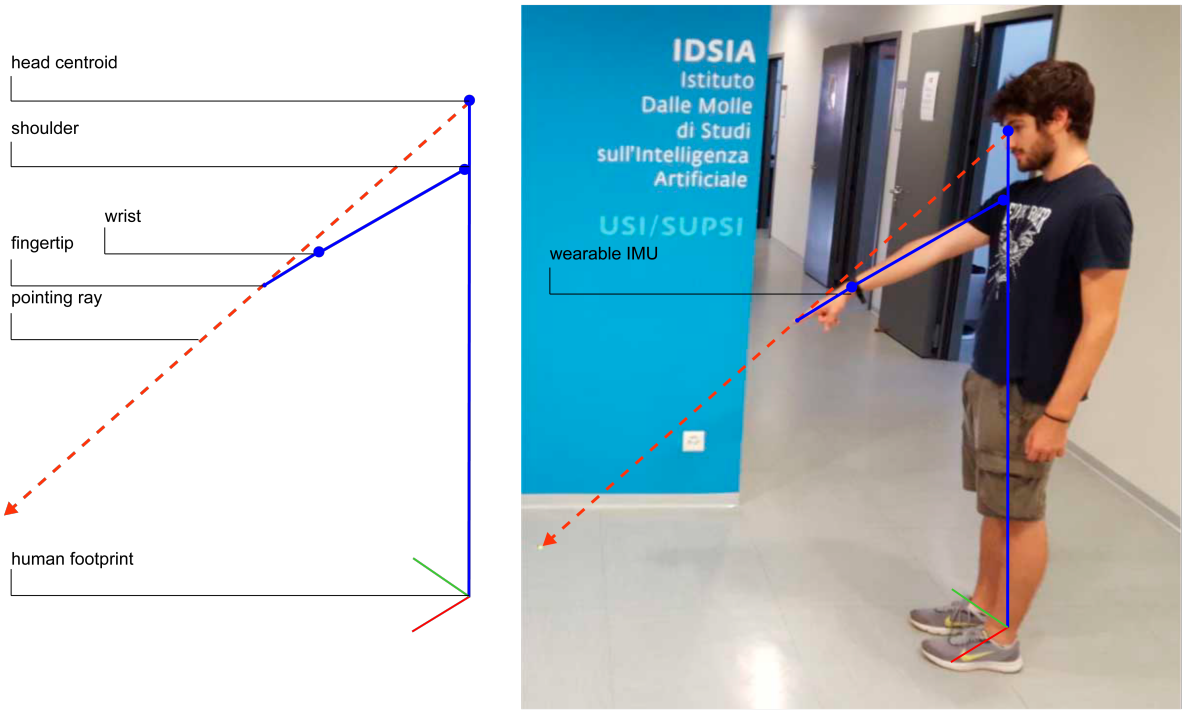

Figure 1.6: Human Pointing Model

Then we compute **L** and **D**:

$$L = \sqrt{x^2 + y^2} \tag{1.12}$$

$$D = \sqrt{(H - z)^2 + L^2} \tag{1.13}$$

Finally,

$$tilt1 = \arctan\left(\frac{L}{H - z}\right) \tag{1.14}$$

$$tilt2 = \arctan\left(\frac{D}{T}\right) \tag{1.15}$$

$$tilt = tilt1 + tilt2 \tag{1.16}$$

1.2 Human Pointing Model

The approach we follow to model human pointing is the one used in [1].
The *head-finger* model defines human pointing rays as originating at a centroid of the head and passing through the index fingertip. Figure 1.6 shows both the model and an explanation photo.
The reference frame **human_footprint** is located at the human feet with the x-axis pointing forward, y-axis to the left, and z-axis pointing up. The pointing ray is a 3D half-line on which the point that the human intends to indicate lies. So, our goal is to intersect that line with the floor (i.e. the xy-plane of **human_footprint** frame) in order to get the point indicated by the user. To do so, we get pointing rays using orientation readings from a wearable IMU. Once we have that, we can compute the intersection of those rays with the ground as a simple *line-plane* intersection, as explained in [28].

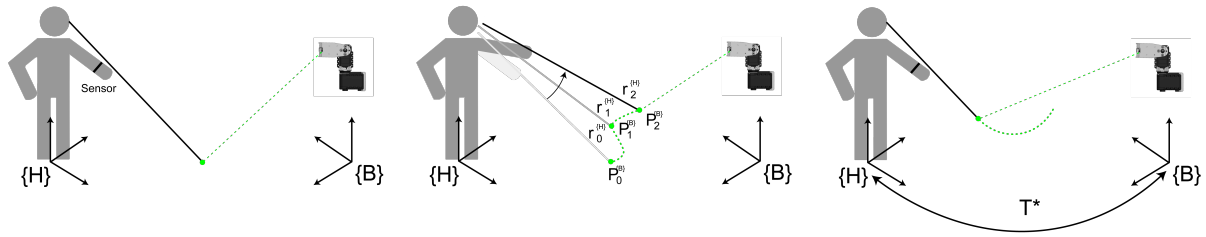

Figure 1.7: *Left*, Phase 1: the operator points at the laser he wants to interact with; an explicit (button press) event triggers the beginning of the interaction. *Center*, Phase 2: the laser dot starts moving along a paths, and the human keeps pointing at it; the system acquires a set of pairs each composed by: a pointing ray r in the operator's frame of reference $\{H\}$; a point P in the turret's fixed base frame $\{B\}$. *Right*: after a few seconds the system has reconstructed the transformation T^* linking $\{H\}$ and $\{B\}$: pointing rays can be known in the turret frame, and the interaction can continue in an application-dependent way.

Consider a line $\mathbf{L}$ given by its parametric equation:

$$P(s) = P_0 + s(P_1 - P_0) = P_0 + s\mathbf{u} \tag{1.17}$$

where the parameter s is a real number and $u = P_1 - P_0$ is a line direction vector. Then, given a plane $\mathbf{P}$ expressed by a point V_0 on it and a normal vector $\mathbf{n} = (a, b, c)$, we must first check if $\mathbf{L}$ is parallel to $\mathbf{P}$ by testing if $\mathbf{b} \cdot \mathbf{u} = 0$, which means that $\mathbf{u}$ is perpendicular to $\mathbf{n}$. In that case, $\mathbf{L}$ and $\mathbf{P}$ are parallel and thus either never intersect or else $\mathbf{L}$ lies completely on plane $\mathbf{P}$.
If the line and the plane are not parallel, then we can compute their unique point intersection. Considering $\mathbf{w} = P_0 - V_0$, at the intersect point, the vector $P(s) - V_0 = \mathbf{w} + s\mathbf{u}$ is perpendicular to $\mathbf{n}$. This is equivalent to say that the dot product:

$$\mathbf{n} \cdot (\mathbf{w} + s\mathbf{u}) = 0 \tag{1.18}$$

Solving 1.18 we obtain the intersect point $P(s_I)$:

$$s_I = \frac{\text{-}\mathbf{n} \cdot \mathbf{w}}{\mathbf{n} \cdot \mathbf{u}} = \frac{\mathbf{n} \cdot (V_0 - P_0)}{\mathbf{n} \cdot (P_1 - P_0)} = \frac{-(ax_0 + by_0 + cz_0 + d)}{\mathbf{n} \cdot \mathbf{u}} \tag{1.19}$$

1.3 Relative Localization

The relative localization procedure goal is to estimate the pose (position and orientation) of the turret's frame in the reference frame of the human. The approach we use is the one proposed in [1].
First, we define the human frame $\{H\}$ the same way we did in section 1.2. The reference frame $\{B\}$ is an arbitrary fixed reference frame in which the turret reports the position of the laser projected dot (in section 1.1 this is the **base_frame**).
The input of the procedure is a finite set of pairs C:

$$C = \{(r_1^{\{H\}}, P_1^{\{B\}}), \ldots, (r_N^{\{H\}}, P_N^{\{B\}})\}$$

where $r_i^{\{H\}}$ is a pointing ray in the reference frame $\{H\}$ and $P_i^{\{B\}}$ is the corresponding laser dot position in the frame $\{B\}$. Those pairs are built collecting user's rays while he is following the moving laser with a pointing gesture for a short period of time τ.

The output of the procedure is a coordinate transformation T^* between the two frames, expressed as a composition of translation and rotation:

$$\rho = [t_x, t_y, t_z, \gamma_z]$$

where $t = [t_x, t_y, t_z]$ represent the translation, γ_z the rotation around the z-axis. Note that we simplify the model ignoring rotations around x(roll) and y-axis (pitch) since the z-axes of the user and the turret are parallel being both on the same plane (i.e. the floor). Finding ρ will allow us to collocate the turret's frame pose (position and orientation) into the human's frame.
This coordinate transformation is obtained through an optimization procedure: given an estimate T of the transformation, we can convert laser positions $P_i^{\{B\}}$ defined in the turret frame into the operator frame:

$$P_i^{\{H\}} = T P_i^{\{B\}}$$

Using these points we define a new ray $q_i^{\{H\}}$ that shares the origin with $r_i^{\{H\}}$, but passes through the point $P_i^{\{H\}}$. In that way we can define an error function θ for a set of pairs C:

$$\theta(T, C) = \frac{1}{N} \sum_{i=1}^{N} \angle(r_i^{\{H\}}, q_i^{\{H\}}) \tag{1.20}$$

where $\angle(\dots)$ represents the unsigned angle between the directions of two rays. So, the goal of the optimization procedure is to find the coordinate frame transformation T^* between the operator frame $\{H\}$ and the laser frame $\{B\}$ that minimizes that error function:

$$T^* = \arg\min_T \theta(T, C) \tag{1.21}$$

Chapter 2

Hardware Implementation

In this chapter we describe the hardware implementation of the system. In particular, in section 2.1 and 2.2 we will focus on the two different laser turrets built, pointing out all the issues and differences that make the second one better. We also mention the arm IMU device and the groud robot used to reconstruct human pointing and build our demos respectively.

2.1 First Turret Model

Figures 2.1 shows the first turret. As we can better see in figure 2.2 and 2.3, it follows the model already seen in chapter 1. So, at the base we have one servo motor in charge to control the *pan* angle. On its flange, a second motor is mounted with a plastic bracket on its own flange. The laser diode is then mounted in the middle of that bracket, in such a way that its direction passes through the middle of the servo shaft. So, the second servo directly controls the *tilt* angle. The *Arduino Uno Board* and the *Bioloid Bus Serial Interface* conclude our hardware list.
In the next sections we analyze each parts and their uses in details.

2.1.1 Dynamixel AX-12+

As reported on the official website [29]:
"*the DYNAMIXEL is a smart actuator system developed to be the exclusive connecting joints on a robot or mechanical structure. DYNAMIXELS' are designed to be modular and daisy chained on any robot or mechanical design for powerful and flexible robotic movements. The DYNAMIXEL is a high performance actuator with a fully integrated DC (Direct Current) Motor + Reduction Gearhead + Controller + Driver + Network, all in one servo module actuator. Programmable and networkable, actuator status can be read and monitored through a data packet stream.*"
The first turret is composed of two **Dynamixel AX-12+** motors.

2.1.2 Motor Specification

The full datasheet can be found here [30]. We are mostly interested in two specifications:

- **Resolution**: 0.29°;
- **Communication speed**: 7343 bps $\sim$ 1 Mbps.

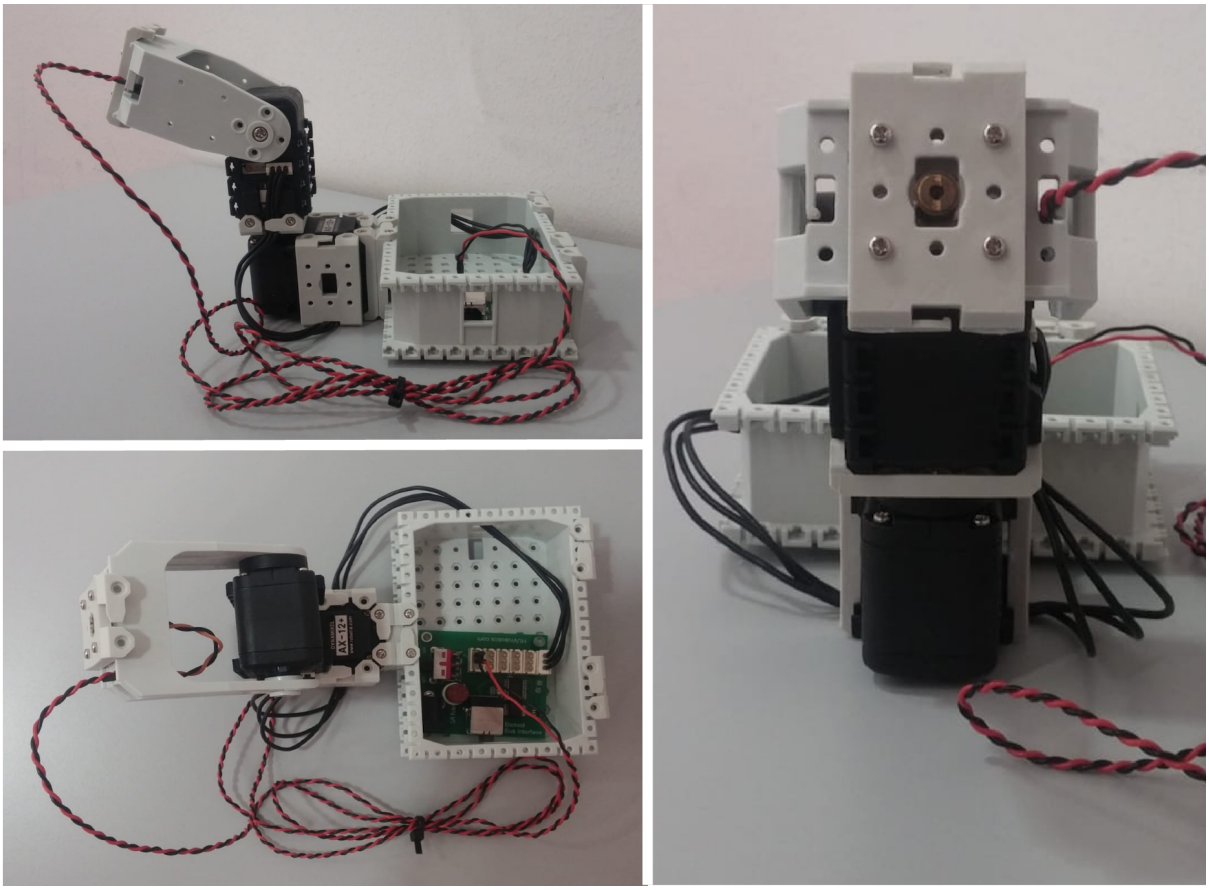

Figure 2.1: First Turret Model, Actual Photos

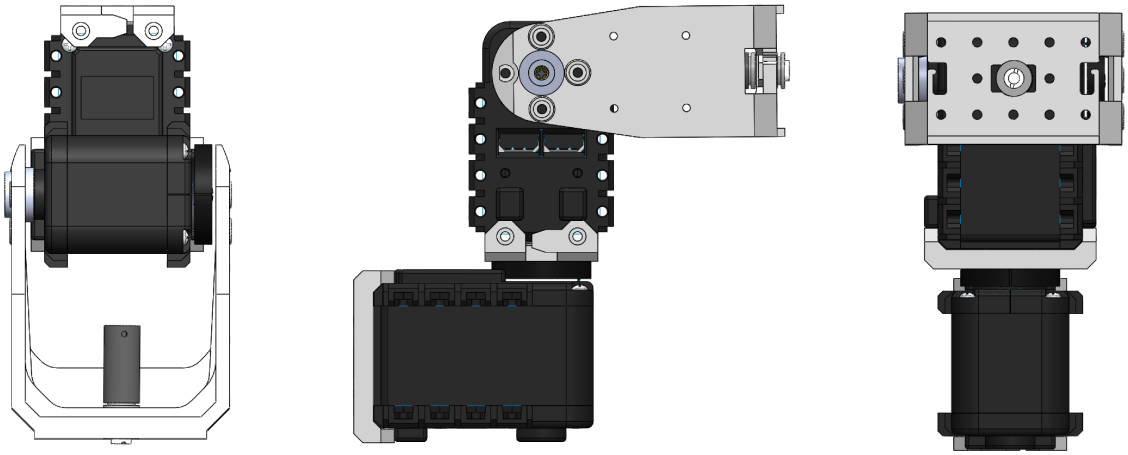

Figure 2.2: First Turret Model, 3D from Different Views

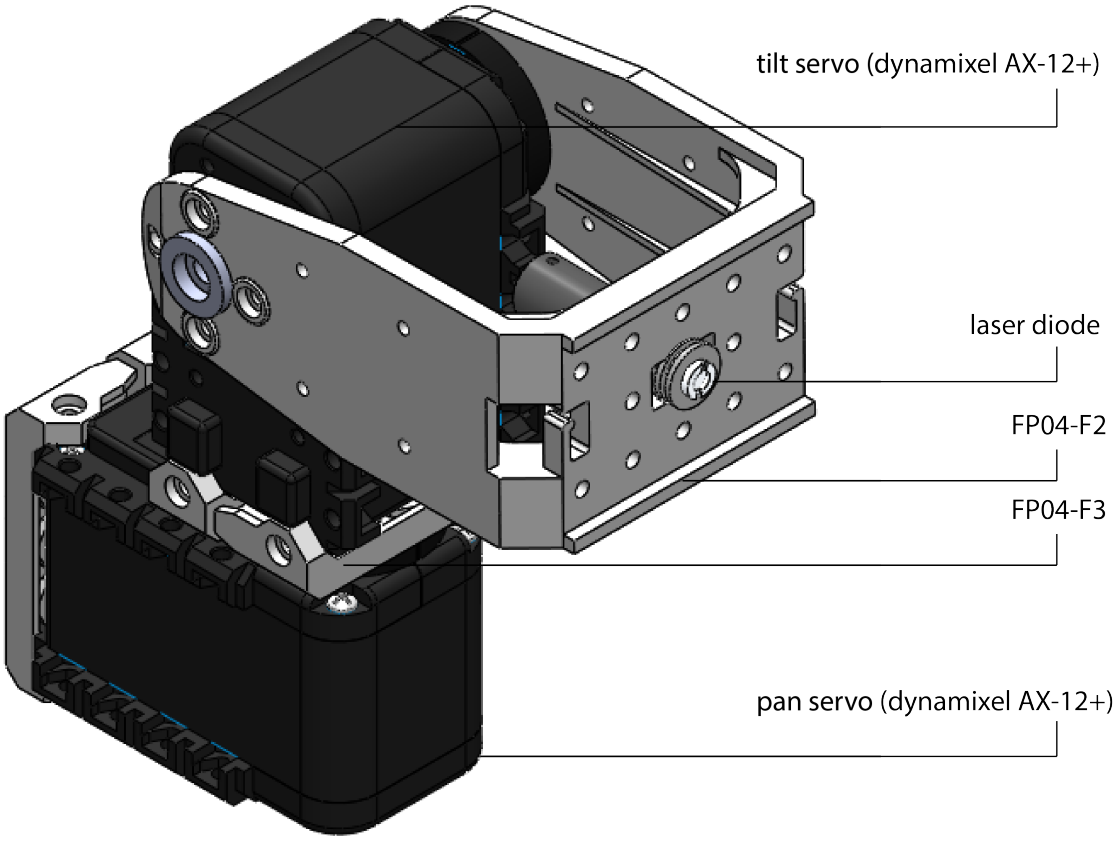

Figure 2.3: First Turret Model, 3D with Parts Names

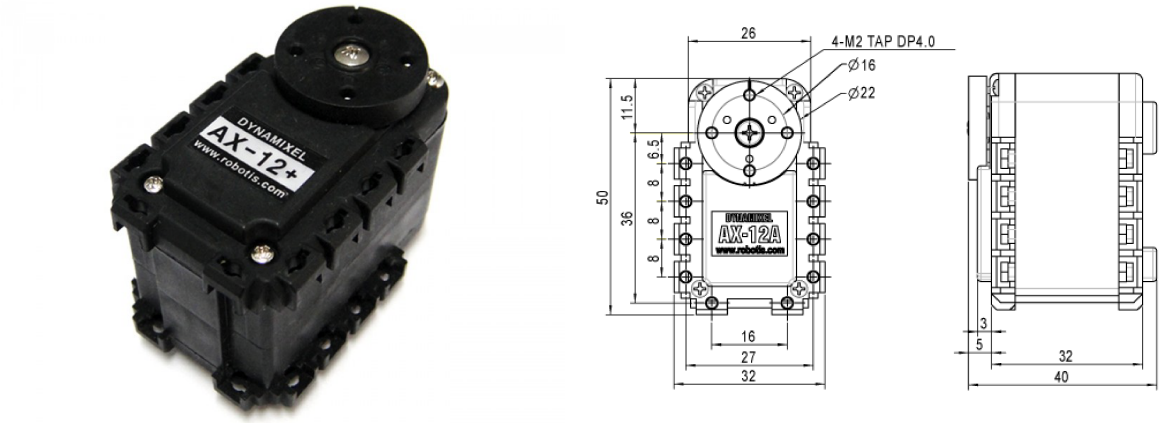

Figure 2.4: Dynamixel AX-12+

Those are the values that set the limitations for that turret, causing the issues we discuss in the next section.

2.1.3 Issues

Here we will only report the issues raised by the motors. The final solution is explained in next chapter. All other attempts are mentioned the appendix A.

Servo Resolution and Slow Speed Limit

Even though a resolution of 0.29° could seem pretty good, that value heavily limits the performance of the turret. As a matter of fact, in our system, the joint will be often asked to do very small movement and, thus, draw tiny angles. Moreover, the motor is not able to move with too much low speed values. That issue becomes even bigger when the laser dot moves far from the turret: the angles become always more smaller. The result is that the performances of the turret can be good enough only on a small sized space around the turret, but that is not certainly enough for our system.

Trajectory Smoothness

The performances degradation heavily impacts smoothness of trajectory drawn with the laser dot. Playing with servos internal parameters we could try to find the right trade-off between precision and smoothness, but since the relative localization is based on the laser positions and the user following those positions, then we can not sacrifice too much either precision or smoothness.

Slow Communication Protocol

This is the main issue. It is only partially related with the communication speed of the motors. Of course, with a higher speed, we could afford to use a slower protocol. Anyway, solving that issue allows us to obtain the best from that turret, hitting its limits, but obtaining a system which could work well enough for our purposes. Since the solution is part of the software implementation, it will be presented in the next chapter.

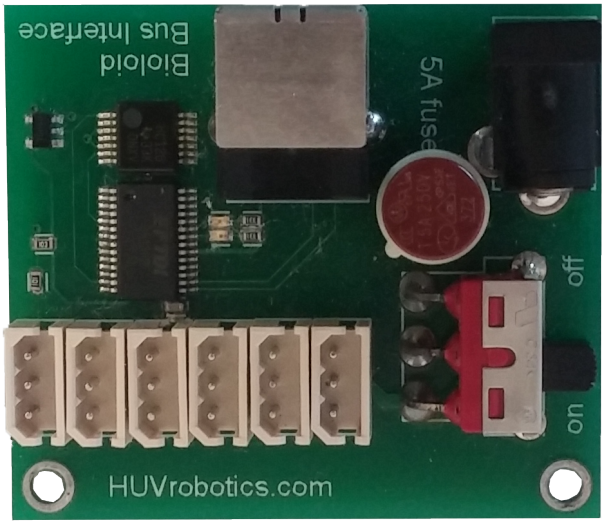

Figure 2.5: Bioloid Bus Interface

Figure 2.6: Arduino Uno Board

Figure 2.7: Laser Diodes

2.1.4 Bioloid Bus Interface

This board allows PC to communicate with Bioloid bus devices (e.g. Dynamixel AX-12) using a USB cable at speeds of up to 1.0 Mbps. In other words, this simple device provides serial communication between the PC and the motors. It is also needed to provide voltage to the motors. We can not use it to power up the laser as the output voltage is too high and there is no voltage regulator.

2.1.5 Laser Diode and Arduino Uno Board

We use a simple 5V red laser diode powered directly from an *Arduino Uno Board*. We use that board because it provides a 5V output and thus is a simple and fast solution to power up the laser.

2.1.6 Structural Parts

Figures 2.8 shows the plastic frames used to assemble the turret. Mounting instructions can be found in appendix A.

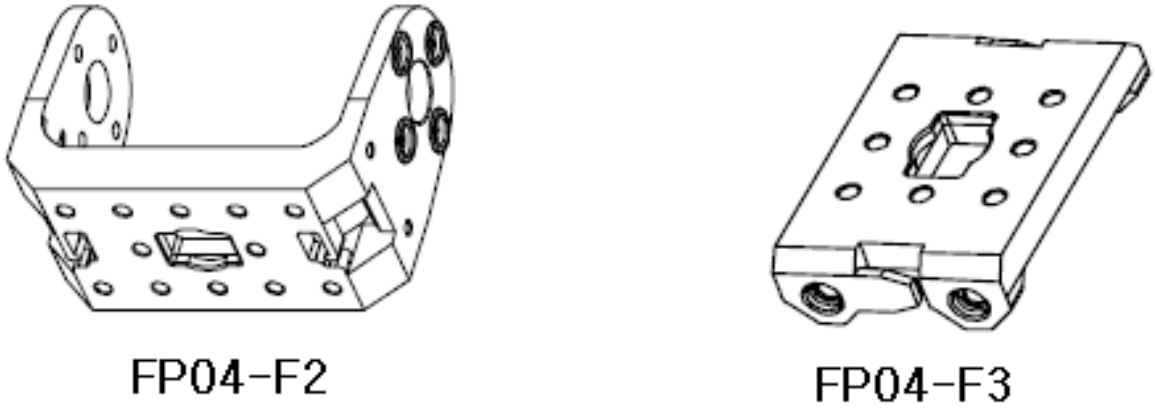

Figure 2.8: Plastic Mounting Frames

2.2 Second Turret Model

As already mentioned, we are going to present a second and more powerful turret. Looking at figure 2.9, 2.10 and 2.11 we can immediately see that there are some structural differences from the previous model, but those are already discussed in 1.1.2. Here, we want to discuss the new hardware, in particular motors and structural parts, which come from the ScorpionX MX-64 Robot Turret Kit by Interbotix [31].

2.2.1 Dynamixel MX-64T

Quoting from [32]:
"*The MX-64T Dynamixel Robot Servo Actuator is the newest generation of Robotis Dynamixel actuator; equipped with an onboard 32bit 72mhz Cortex M3, a contact-less magnetic encoder with 4x the resolution over the AX/RX series, and up to 3mpbs using the new TTL 2.0 bus. Each servo has the ability to track its speed, temperature, shaft position, voltage, and load. As if this weren't enough, the newly implemented PID control algorithm used to maintain shaft position can be adjusted individually for each servo, allowing you to control the speed and strength of the motor's response. All MX Series servos use 12v nominal voltage, so MX Dynamixels can be mixed without having to worry about separate power supplies. All of the sensor management and position control is handled by the servo's built-in microcontroller. This distributed approach leaves your main controller free to perform other functions.*"
So, we already know that those motors are more powerful than the ones used for the first model. Moreover, they are also compatible. The result is that we can reuse the same software interface written for the first turret without any issues, obtaining a more reliable model in term of accuracy, precision and trajectory smoothness. That confirms that things are done right also with the first turret, but we are facing its physical/hardware limitations. The results with the new turret are considerably better even without directly exploiting and tuning the more sophisticated functions offered by the new motors, such as built-in PID controllers or faster communication.

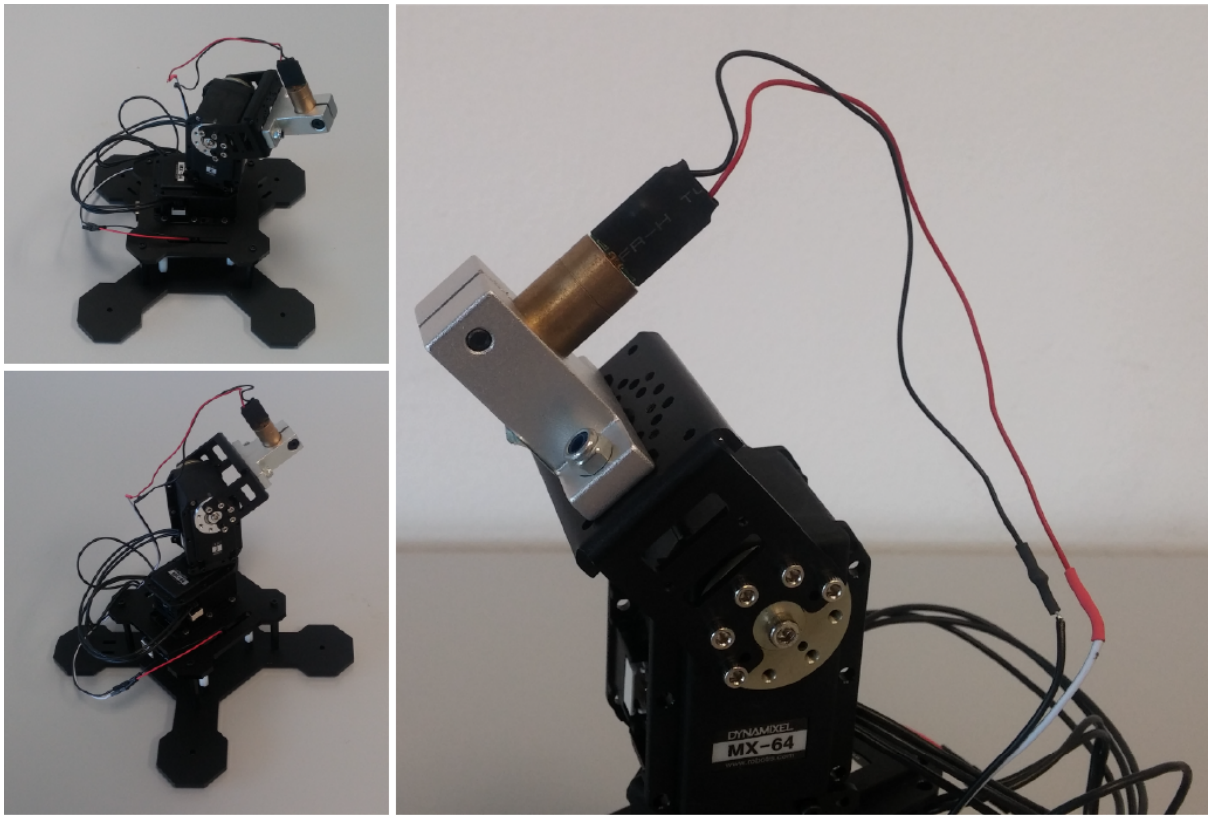

Figure 2.9: Second Turret Model, Actual Photos

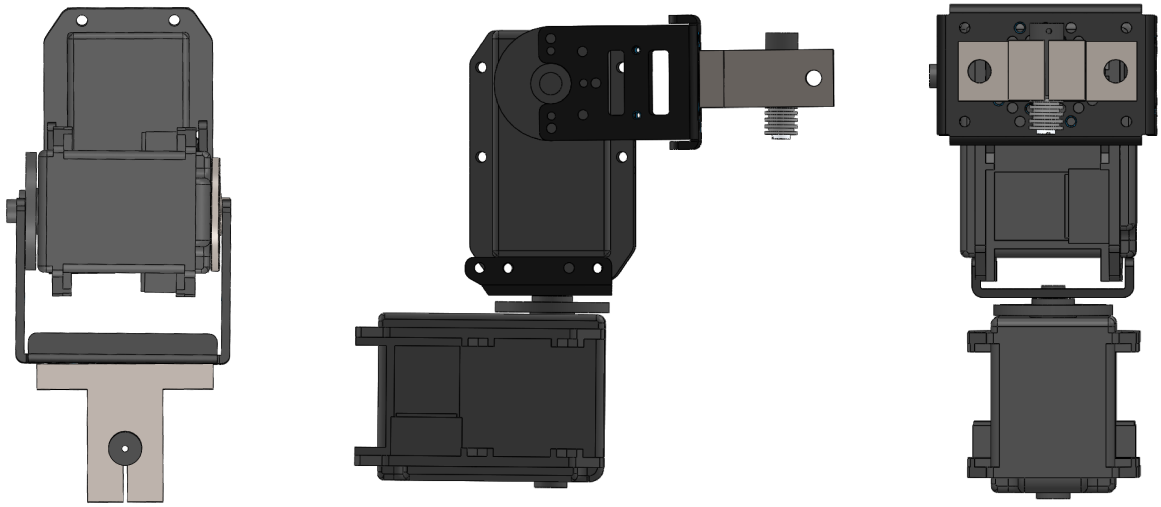

Figure 2.10: Second Turret Model, 3D from Different Views

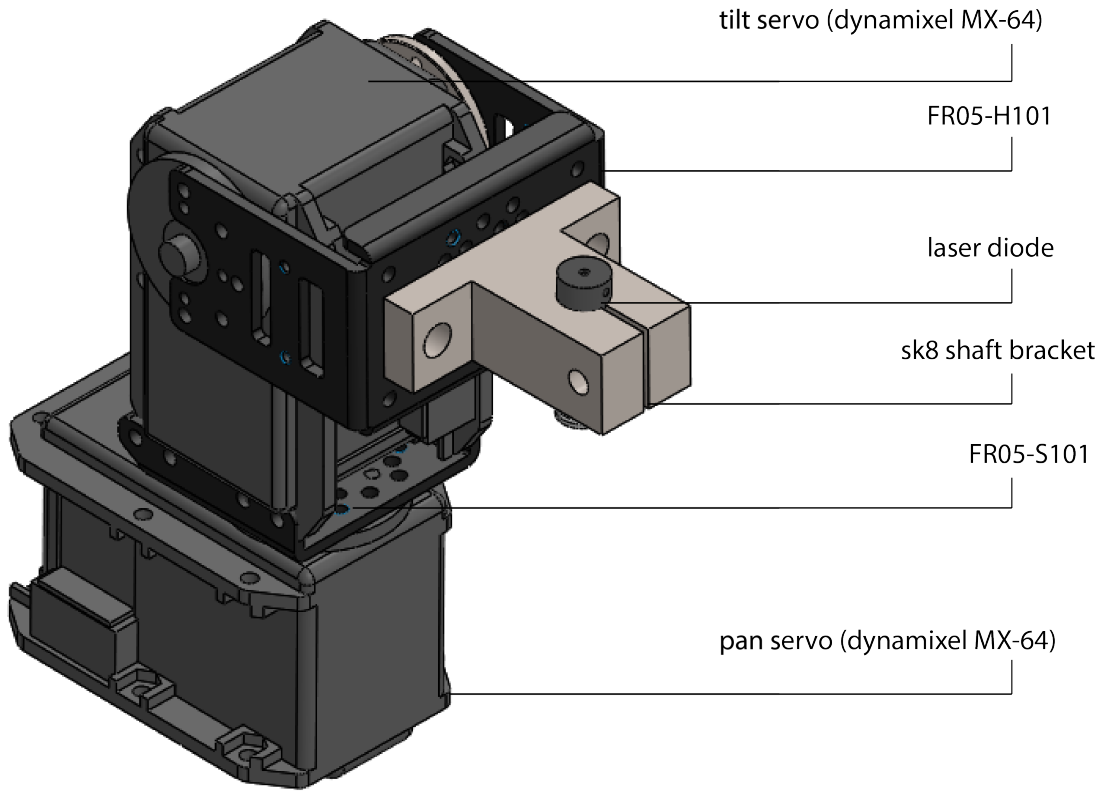

Figure 2.11: Second Turret Model, 3D with Parts Names

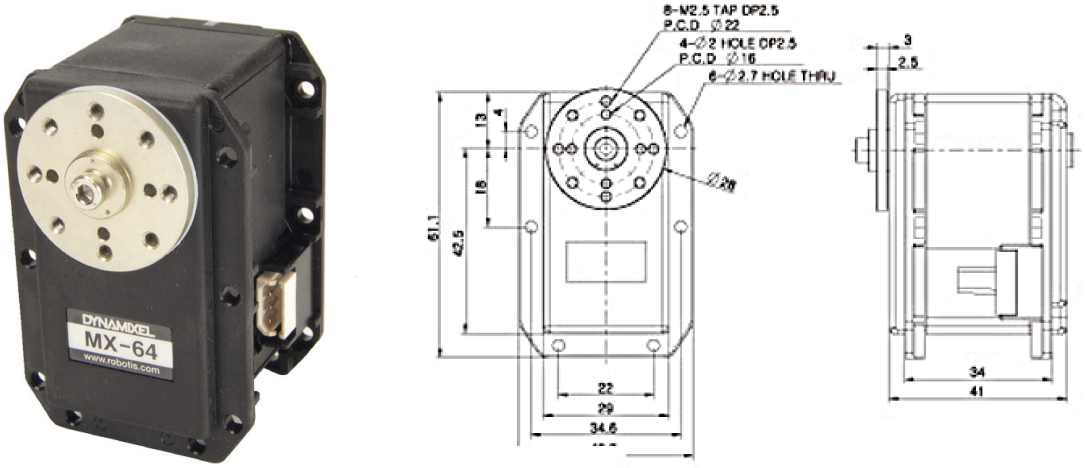

Figure 2.12: Dynamixel MX-64

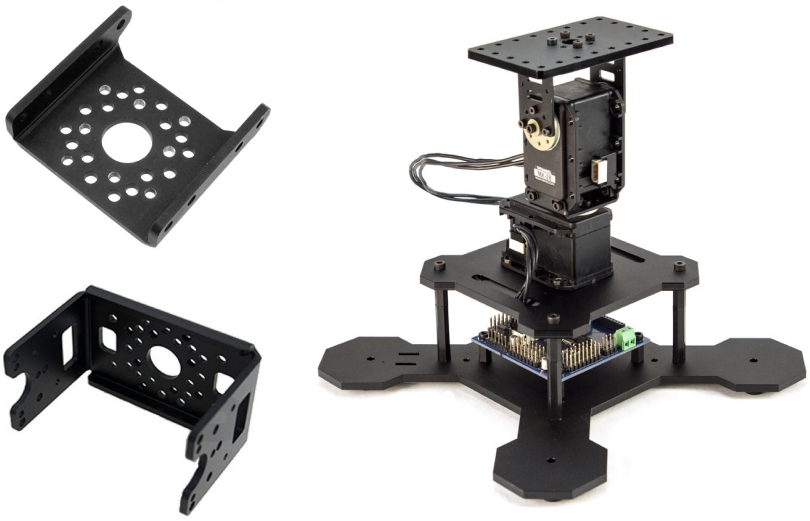

Figure 2.13: *Top-Left*: FR05-S101 Side Frame. *Bottom-Left*: FR05-H101 Hinge Frame. *Right*: ScorpionX MX-64 Robot Turret Kit.

2.2.2 Motor Specification

From the full datasheet [33] we report only two important specifications:

- **Resolution**: 0.088°;
- **Communication Speed**: 8000 bps $\sim$ 4.5 Mbps.

As we can see, the resolution of those servos is roughly 3 times better than the Dynamixel AX-12. This is enough to make the new turret perform perfectly for our needs using exactly the same approach we use to solve the issues explained in 2.1.3

2.2.3 Other Components

Though we use a 3V green laser diode on that turret, the other components are the same: *Bioloid Bus Interface* for communication and *Arduino Uno Board* for powering the laser (that time from the 3.3V output).
The structural parts are provided together with the ScorpionX MX-64 Robot Turret Kit. Figure 2.13 shows the components used from that kit: the turret base and the metal frames. Since the bracket of the tilt servo is too short, we had to mount the laser diode perpendicularly, obtaining the different structure already discussed.

2.3 Arm IMU

As explained in 1.2, we need an arm IMU device to reconstruct the direction of human pointing. First, we present the Myo Armband, "*a gesture recognition device worn on the forearm and manufactured by Thalmic Labs. The Myo enables the user to control technology wirelessly using various hand motions. It uses a set of electromyographic (EMG) sensors that sense electrical activity in the forearm muscles, combined with a gyroscope, accelerometer and magnetometer to recognize gestures.*". Obviously, we are interested only into the functions provided by gyroscope, accelerometer and magnetometer. We

Figure 2.14: IMU Devices: *Left*: Myo Armband. *Right:* MetaMotionR Wrist Band.

used such an expensive and mostly unnecessary device because it was what we already had.
However, our later work is based on a cheaper and more suitable device, the MetaMotionR board by Mbientlab, which can be mounted on a wrist band and provides a 9-axis IMU with Sensor Fusion. This is more convenient for our purposes, as it is easier to setup (it has a bluetooth driver for linux), more comfortable to wear (it is just like a watch) and also provides a LED for colour feedback and a small button (both are useful for our experiments and demos).

2.4 Kobuki

"*iClebo Kobuki is a low-cost mobile research base designed for education and research on state of art robotics. With continuous operation in mind, Kobuki provides power supplies for an external computer as well as additional sensors and actuators. Its highly accurate odometry, amended by our factory calibrated gyroscope, enables precise navigation.*" [34].
As already stated, one of the main application of our system is ground robot navigation. We developed a couple of demo using the *Kobuki*, a groud robot suitable for our purposes. Note that we actually use a *TurtleBot2*, but only its structure (i.e. *Kobuki* base plus mounting platform). For example, we do not exploit the *kinect* camera provided with the *Turtlebot2*. Its structure is perfectly suitable to mount our turret on top of it (figure 2.15) and thus demonstrate our system.

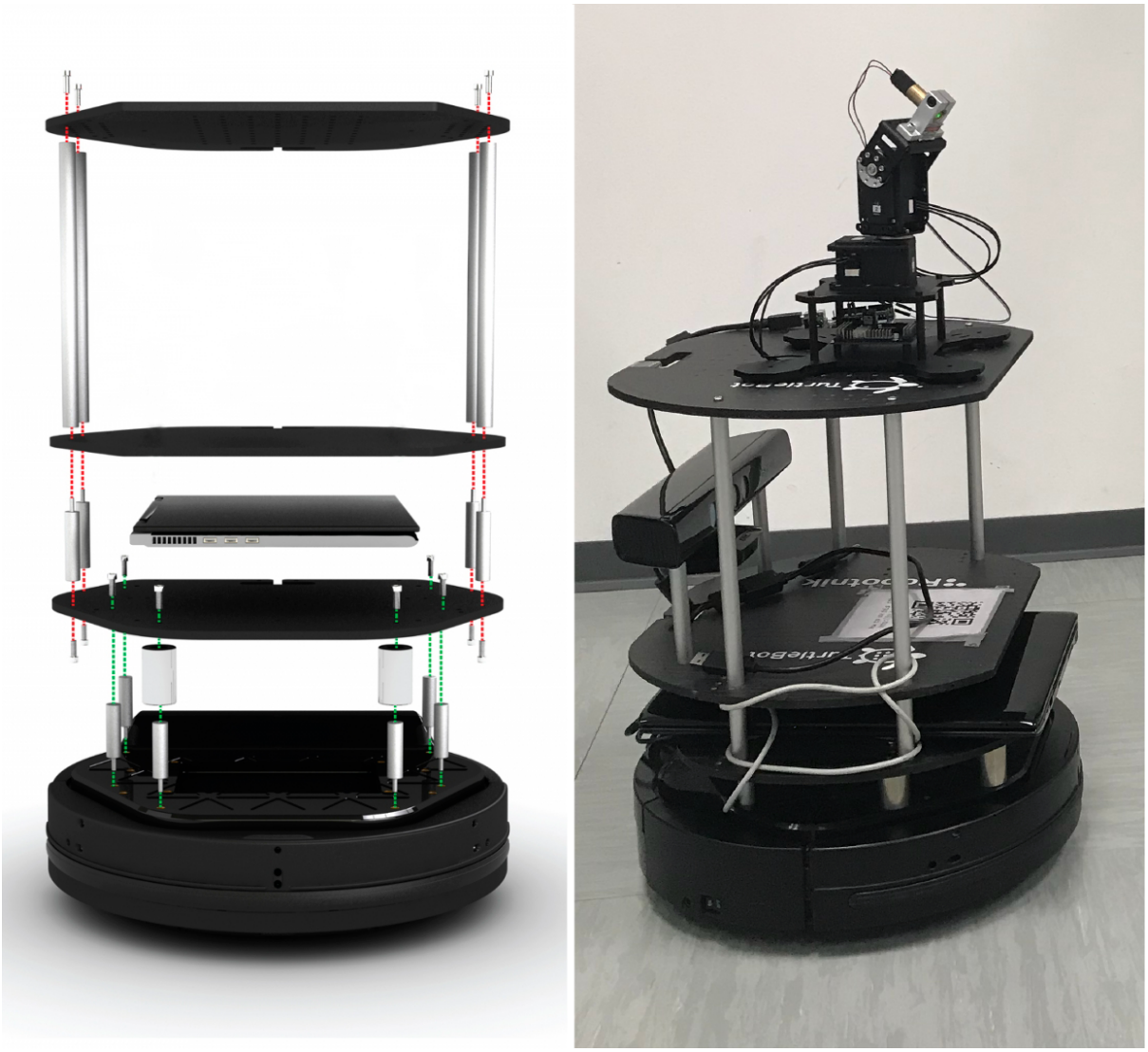

Figure 2.15: Turtlebot2 (with Kobuki base) and Our Turret

Chapter 3

Software Implementation

Now that we have all the theoretical and practical elements of the system, here we report all the work done to implement it on a software side. That includes code written from scratch and also code used to interface with already existing parts. This chapter can be divided in 4 parts:

1. introduction of all the basics libraries used;
2. description of the code implemented to control the turret and the solution adopted to overcome the issues mentioned in 2.1.3;
3. explanation on how we interface with the arm pointing system and the relative localization procedure;
4. demos implementation.

Experiment setups and procedures are left for chapter 4.

3.1 Libraries and Frameworks

The system developed is composed by many modules handling different functionalities. It is worth to mention a couples of libraries and frameworks which are widely used throughout the project, as some of them are also primary to understand the implementation. In the next section we will give an overview of:

- ROS Kinetic

  `rospy` 1.12.14

  `rosbag` 1.12.14

  `tf` 1.11.9

  `rviz` 1.12.16

- `Dynamixel Workbench` and `Dynamixel SDK`
- `NumPy` 1.14.4
- `Matplotlib` 1.5.1
- `pandas` 0.23.3

The programming language used for development is Python 2.7.12.

3.1.1 Robot Operating System (ROS)

ROS is a open-source[1] meta-operating system for robots [35]. ROS is the industry and research standard framework for robotics and it is aimed to help software developers to create robotic applications.

Its primary goal is to support code reuse in robotics research and development. ROS is a distributed framework of processes (aka Nodes) that enables executables to be individually designed and loosely coupled at runtime. These processes can be grouped into Packages and Stacks, which can be easily shared and distributed. ROS also supports a federated system of code Repositories that enable collaboration to be distributed as well. This design, from the filesystem level to the community level, enables independent decisions about development and implementation, but all can be brought together with ROS infrastructure tools.
At the lowest level, ROS offers a message passing interface that provides inter-process communication and is commonly referred to as a middleware. The ROS middleware provides these facilities [36]:

- publish/subscribe anonymous message passing
- recording and playback of messages
- request/response remote procedure calls
- distributed parameter system

In addition to the core middleware components, ROS provides common robot-specific libraries and tools to get a robot up and running quickly. Here are just a few of the robot-specific capabilities that ROS provides:

- Standard Message Definitions for Robots
- Robot Geometry Library
- Robot Description Language
- Pose Estimation
- Localization
- Mapping
- Navigation

in particular, the *Robot Geometry Library* is widely used for that project.

[1]Main client libraries and tools are under the terms of the BSD license (`https://en.wikipedia.org/wiki/BSD_licenses`).

Robot Geometry Library: tf

A common challenge in many robotics projects is keeping track of where different parts of the robot are with respect to each other. For example, if one wants to combine data from a camera with data from a laser, he needs to know where each sensor is, in some common frame of reference. This issue is especially important for humanoid robots with many moving parts. This problem is addressed in ROS with the `tf` (transform) library, which will keep track of where everything is in the robot system.
Designed with efficiency in mind, the `tf` library has been used to manage coordinate transform data for robots with more than one hundred degrees of freedom and update rates of hundreds of Hertz. The `tf` library allows one to define both static transforms, such as a camera that is fixed to a mobile base, and dynamic transforms, such as a joint in a robot arm. One can transform sensor data between any pair of coordinate frames in the system. The `tf` library handles the fact that the producers and consumers of this information may be distributed across the network, and the fact that the information is updated at varying rates.
`tf` lets the user keep track of multiple coordinate frames over time. `tf` maintains the relationship between coordinate frames in a tree structure buffered in time, and lets the user transform points, vectors, etc between any two coordinate frames at any desired point in time.

Nodes

ROS nodes are executable files that use ROS to communicate with other nodes. Nodes are usually fine grained processes that perform precise computations. To communicate, nodes use a ROS client library to publish or subscribe to *topics* or *services*. Different Nodes can run on different hardware while being in the same ROS system.

Bags

Another function of ROS, are `rosbag` files. Bags are a format for saving and playing back ROS message data. Bags are an important mechanism for storing data, such as sensor data, that can be difficult to collect but is necessary for developing and testing algorithms.

ROS Visualization: rviz

`rviz` is a 3D visualizer for displaying sensor data and state information from ROS. Using rviz, one can visualize virtual model and configuration of the robot configuration. One can also display live representations of sensor values coming over ROS Topics including camera data, infrared distance measurements, sonar data, and more.
That tool is helpful as it allows to simulate the system without having to move the real hardware. In that way one can asses the correctness of algorithms etc . Of course, that does not guarantee that in the real world thing are going to work, but this is what robotic is all about!

3.1.2 Dynamixel SKD

The `ROBOTIS Dynamixel SDK` is a software development kit that provides Dynamixel control functions using packet communication. The API of Dynamixel SDK is designed

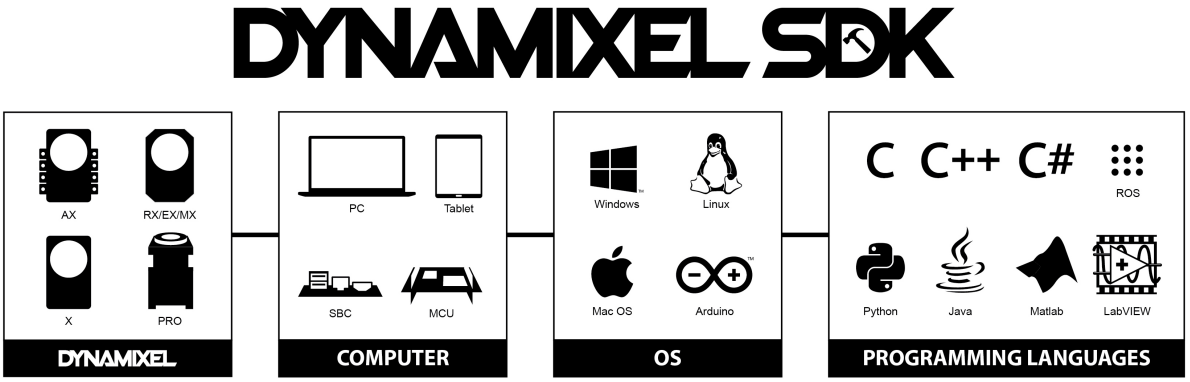

Figure 3.1: Dynamixel SDK

for Dynamixel actuators and Dynamixel-based platforms. It is based on C/C++ programming. The Dynamixel SDK supports all Dynamixel series developed by `ROBOTIS`. For example, all series such as AX, RX, EX, MX, XL, XM, XH, PRO-L, PRO-M and PRO-H are supported by packet communication.

3.1.3 Dynamixel Workbench

Dynamixel-Workbench is dynamixel solution for ROS. This metapackage allows one to easily change the ID, baudrate and operating mode of the Dynamixel. Furthermore, it supports various controllers based on operating mode and Dynamixel SDK. These controllers are commanded by operators.

3.1.4 Numpy

NumPy is the fundamental package for scientific computing with Python [37]. It contains among other things:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multidimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

3.1.5 Matplotlib

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms [38]. Matplotlib can be used in Python scripts, the Python and IPython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits.

Matplotlib makes plotting high quality graphs accessible to anyone. With Matplotlib it is possible to generate a wide variety of types of plots from histograms to scatter plots with just a few lines of code.

Matplotlib tries to make easy things easy and hard things possible. One can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc., with just a few lines of code. Matplotlib uses a MATLAB-like interface. Power user have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions similar to MATLAB one.

3.1.6 pandas

pandas[2] is an open source community supported, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for Python. pandas is still in active development and since it has seen a growth in adoption, on the official documentation multiple guides are present to help shifting from another environment or language like R.

3.2 Turret Software Implementation

First, we must recall what are our goals. We want to be able to drive a laser dot on a given surface (i.e. the floor) by setting its x and y coordinates (the z would be implicit on the floor). This must be done with a good precision and also with high frequency, as we want to draw nice and smooth trajectories with the laser. To do so, we can only control two Dynamixel servos, setting their angles values accordingly to the inverse kinematics solved in 1.1.
So, we will describe how we implement the turret model and then the interface written to control the motors and achieve our goals.

3.2.1 Turret Model Implementation

The two turret models are implemented as ROS node containing the inverse kinematic equation and the code to publish the `tf tree` of the turret. Those models, in fact, are simply implemented as a chain of `tf frames`. In that way, we can easily keep track of the position/transformation of our frame, corresponding to our *pan* and *tilt* angle. This is an immediate implementation of the logic already depicted in figures 1.2 and 1.4.
So, those are the nodes in charge to compute and keep track of the values of our two DoF.

3.2.2 Motors Controller

The drivers for both the AX-12+ and the MX-64 Dynamixels are provided by the `dynamixel sdk`. Moreover, the `dynamixel workbench` offers ROS interfaces to work with the official sdk. That is one of the most crucial part of software, because it is directly related to the issues mentioned in section 2.1.3. As a matter of fact, it is where

[2]The name is written in lowercase letters.

we intervene to solve those issues, which are mostly due to the slow communication protocol. In fact, the motors are not able to process all the points composing the trajectory that we need to send at 50Hz and they simply drop them. That happens because each time a command is sent through the bus interface, each motors has to reply with an acknowledgment. So, we disable all kind of response from the servos, assuming that once a command is sent, it will be properly executed. Moreover, we hack the drivers to be sure that the PC will not wait for motors replies. That is a bit tricky as, even though the low level function to write the servos registries without wait for reply exists, we have to find and expose it through the workbench libraries.
Given that, we are able to draw our smooth trajectories in the simplest way possible: sending points at a fixed rate with the servos in joint mode. *Joint mode* means *position control*. In other word, we can directly specify the value of the position/angle we want the servo to reach to the servo itself. Its internal controller will move the motor accordingly. That works fine with the first turret model, but also with the second one. Even better, the *MX-64* based turret works perfectly thanks to its better servo resolution.

Even if the two turret model could use the same interface, the software project contains two different ones as, for the first turret, we use an unofficial low level driver that is tightly coupled with the architecture of the Dynamixel *AX-12+* itself and thus is not reusable. On the contrary, for the second turret, the code is rewritten to directly exploit the generic ROS service interface offered by the `dynamixel workbench`. That code would also work with the first model.
Finally, we have a useful module to publish the points composing the desired trajectory.

3.2.3 Laser Turret Complete Picture

Now that all the software parts composing the turret have been introduced, we can put everything together to understand exactly how the code is modulated and which module is in charge of what. Here we will consider the module related to the *MX-64* turret, since they would be suitable to interface also with the *AX-12+* one. Figure 3.2 give us an idea.

tf_broadcaster is the module containing and simulating the geometrical model of the turret. It also computes the inverse kinematic of the laser dot. So, this is the piece of software in charge of computing the joints position (i.e. *pan* and *tilt* values) corresponding to the laser dot goal position (expressed in the reference frame of the turret). It only takes a 3D point as input, but serves all our purposes: it can be used to draw a trajectory, as long as ones send a stream of point as input; it can be used to follow the human pointing, after the relative localization is done; it can, of course, mark a single spot on the floor (as we need it for the experiments). It will publish those values and update its internal representation of the turret accordingly, but it will not directly send the commands to move the motors. This is very convenient, as it allows us to simulate the system even without a physical turret and decouples the logical model from the physical controller.

turret module contains that physical controller. Again, that code is written with the idea of making the system modular. In fact, this class leverages on the `dynamixel workbench` to send commands to any Dynamixel motors. In that way, as long as we are working with a device which needs to set only two angles (e.g. *pan* and *tilt* unit), we can

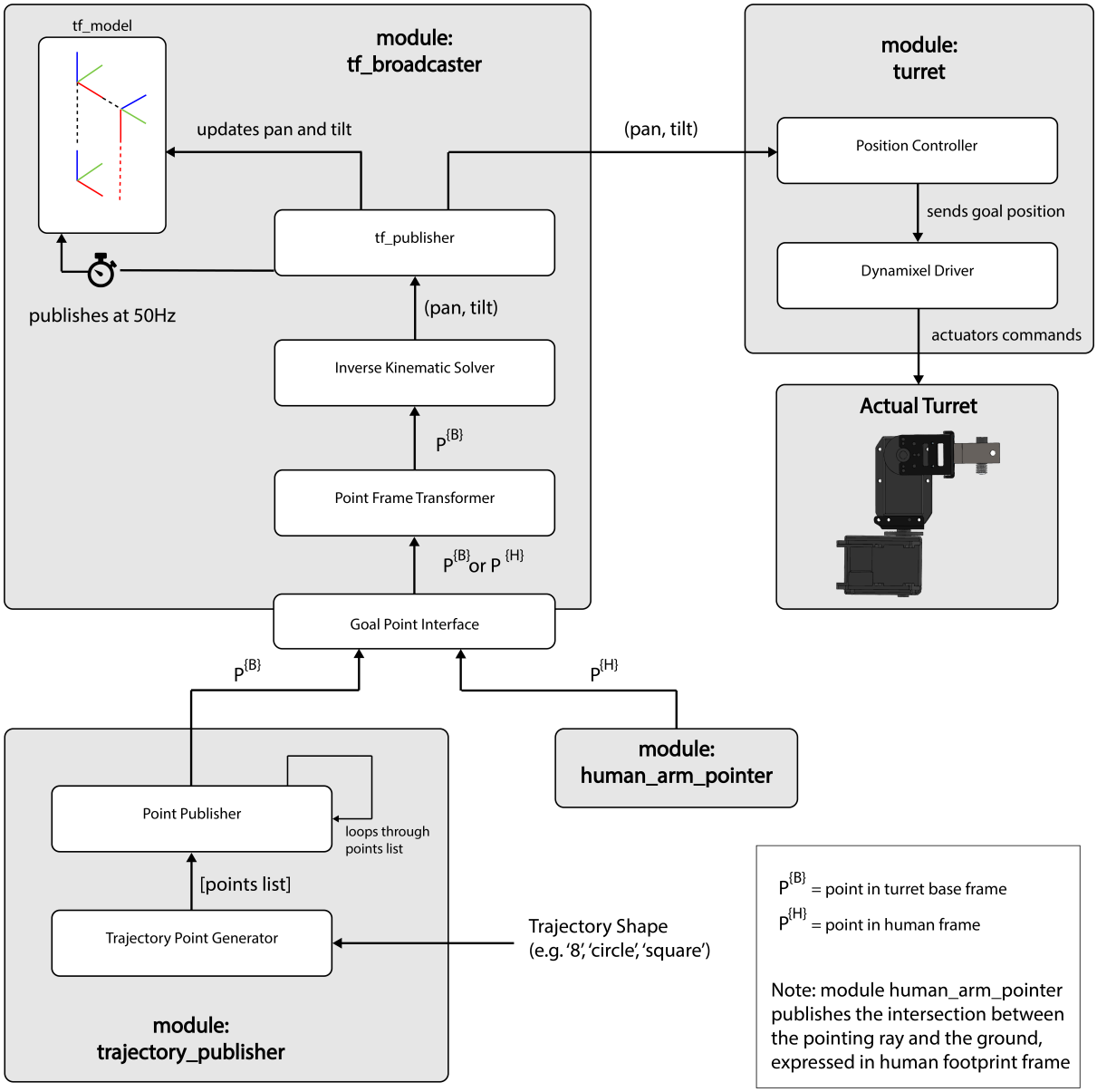

Figure 3.2: Laser Turret Implementation, the Complete Picture

plug into the system any turret built with any Dynamixel motor model. Needless to say, that was helpful when we had to deal with two different turret models.

trajectory_publisher is needed to decide the shape of the trajectory to draw. Then, it will publish each point at a fixed rate to the interface provided by `tf_broadcaster`. In that way, the laser dot will follow the desired trajectory. We can draw many different shapes, but in our experiments and demos we use the "∞" shape.

3.3 Pointing Software Implementation

As explained in section 1.2, we are using the human pointing model suggested and implemented for [1]. The geometry already presented in that section is very convenient as it can be implemented directly into code, as in the **arm_node** module.

Thus, is very simple to understand how that part works: we receive as input the quaternions (i.e. the rotation) from the wearable IMU. In that way we can determine the direction of the pointing ray that we know has its origin between the eyes and passes through the fingertip of the human, exactly as previously shown in figure 1.6. That requires also that certain measures of users' bodies are known. In particular, for each user we take the height which goes from the ground to the centroid of the head, from the shoulder to the wrist and from the wrist to the fingertip
We will not explain that code in details since it was not developed for that thesis, but we will mention one interesting fact related to pointing on the floor and the differences occurred when implementing the pointing on the wall (that part was developed for the thesis).

3.3.1 Pointing on the Floor

Keeping in mind that the pointing ray is an infinite line, we can easily see that there is a problem: when the human is pointing above the horizon (i.e. is not pointing on the floor), the ray will be still intersecting the plane behind the human. Luckily, for how things have been defined in 1.19, we can simply check whether $s_I \geq 1$ to be sure that the intersection is ahead of the human. If not (also if the ray is parallel to the floor), we do not have a valid intersection, thus we return a pose for the pointer made with `NaN` values.

3.3.2 Floor vs Wall

As already stated, for that thesis we implemented also a demo with the user pointing on the wall. The implementation for the pointing itself was pretty straightforward, as it is similar to the one on the floor, but there are a couple of details that are implicit when pointing on the floor and that must be taken into account for the wall case. In fact, assuming that the human is standing on a flat plane perpendicular to him, which is a fair assumption for our system, we already know the distance from the floor (i.e. 0) and its orientation. So, we know that if the human is not pointing above the horizon, we have an intersection. However, in the wall case, we have to explicitly know and set the distance of the human from the wall and its orientation to intersect the pointing ray. We could always determine them from the turret with the relative localization, but, again, that would require that the distance and the orientation at least of the turret from the wall are known. That makes the demo for our system on the wall limited by those initial conditions that, on the contrary, can be implicitly taken for granted in the floor case. To be precise, if we place the turret on a table, we would need to take its height in any case, but that would be the only thing needed for the floor case.

3.4 Relative Localization Software Implementation

As for the the pointing, the relative localization (relloc) implementation is the same proposed in [1]. However, that code was written to work with a drone, so a flying robot with an odometry which is slightly different from a ground robot (and a laser point). For that thesis, only a couple of modifications were needed to interface the turret and then replace the drone with the laser dot. The result can be seen in **motion_relloc** module. In that section we will give an overview of how that procedure is implemented in terms of input and output.

3.4.1 Relloc Input

As already explained in 1.3, the relloc procedure takes as input a set of pairs composed by the position of the laser dot in the turret frame and the *corresponding* pointing ray in human frame. *Corresponding* means that we want to get the data of the ray generated by the human while the turret was marking that particular point with the laser. This is crucial, because it means that data must be synchronized precisely. This also explains why we wanted the turret to be as precise, fast and smooth as possible. Thus, we are sending trajectory point to the turret at $50Hz$ and collecting data from the arm IMU at the same rate.
To build our input set we specify the duration of the interaction in seconds (usually is $5s$ in our demos) and the number of pairs to sample (usually 250). Since in 5 seconds at $50Hz$ we collect exactly 250 pairs, it means that we usually sample the whole set. We use those values as 5 second are enough to draw an ∞ shaped trajectory with a speed that allows the human to follow the laser easily.

3.4.2 Relloc Output

Recalling what was written in section 1.3, the output of the relloc is the transformation:

$$\rho = [t_x, t_y, t_z, \gamma_z]$$

which allows us to collocate the turret's frame pose (position and orientation) into the human's frame.
To obtain that transformation we iteratively call the optimization procedure contained already explained. We start with an arbitrary initial guess of:

$$\rho = [0, 0, 0, 0]$$

and then try to reduce the error as defined in equation 1.20, by sampling each pairs in a random order and updating ρ accordingly.
In the end, we publish the transform of the turret into the human frame for a fixed time window (usually $60s$), leveraging on the functions provided by `tf`.

3.5 System Complete Pipeline

Now we finally have all the elements to understand the entire system pipeline, from the turret to the relloc. Figure 3.3 shows a schematic with all the involved modules. We can see that the turret draws a trajectory, the human follows that trajectory with pointing gestures. The system then puts together each laser point with each pointing ray and computes the relative localization. Once the relloc is estimated, it can be used for different applications, as we show with demos and experiments.

3.6 Demos Implementation

We have implemented three different demos to show the behaviour of the system and also demonstrate possible use cases. In that section we will discuss only aspects related to the software implementation of those demos. They are presented in depth in chapter 4.

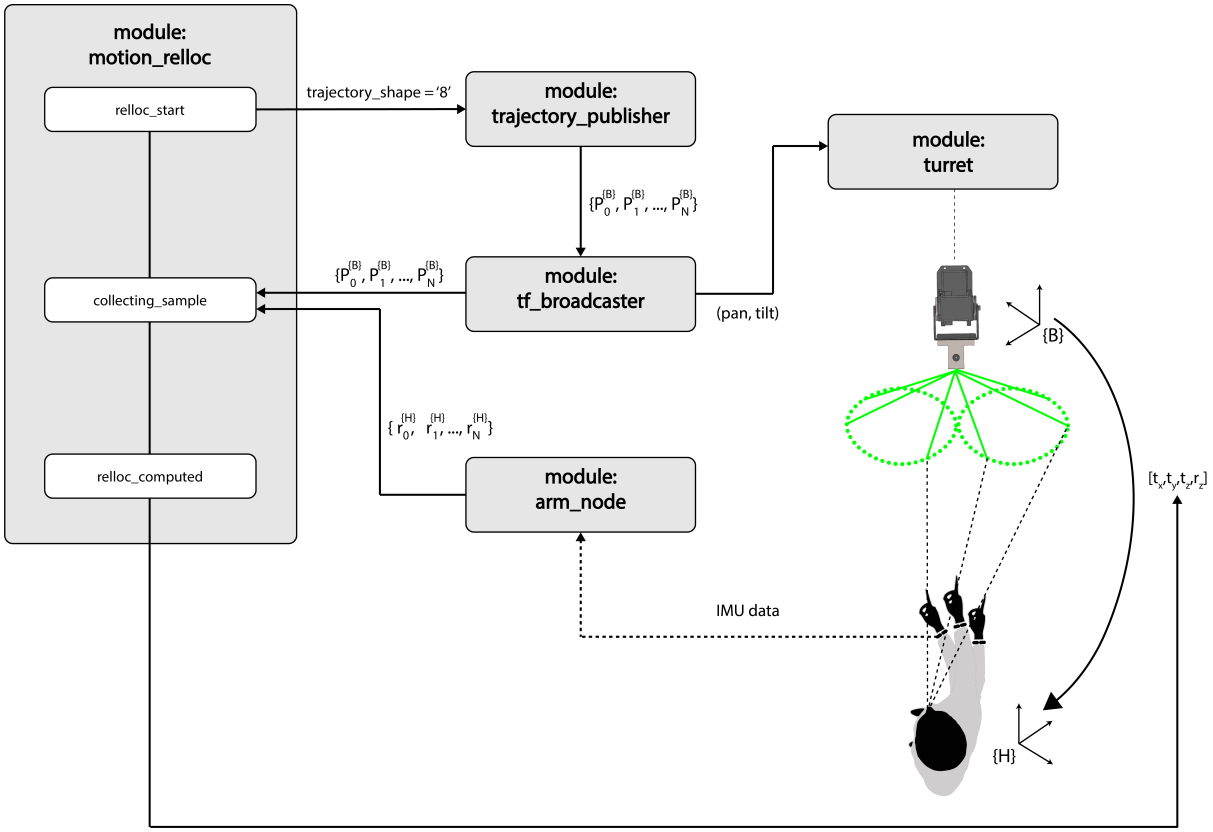

Figure 3.3: System Complete Pipeline

3.6.1 Relloc Demos

Obviously, those demos are needed to give an example of the relloc. There are 2 different version of it. In both cases, first we compute the relloc, then we have two different behaviour: in one case, we use the estimated position to mark the point where the user is standing with the laser; in the other case, we allow the user to directly control the laser dot position on the floor (we implemented also a wall version) with pointing gestures (figure 3.4).
In both cases, all we have to do is compute transformation between human and turret frame: in one case just to obtain the human position in the turret frame, in the other to obtain the point the human is pointing in the turret frame. All these transformations needed are easily obtained, as usual, thanks to the `tf` library for ROS, which played a major role for those demos. As a matter of fact, in figure 3.4 we can see a user driving the laser around pointing at the ground. Since the relative location is known, goal points expressed with pointing rays in frame $\{H\}$ can be transformed into $\{B\}$ reference and thus be used to drive the turret.

3.6.2 Kobuki Go to Goal Demo

That demo exploits the kobuki platform to show an interesting use case for the system: mobile robot navigation. In that case, we do the relloc and then tell the kobuki to reach a certain position by pointing at it for three seconds.
In addition to the relloc system, we implemented a basic PID controller to move the kobuki. The library to interface with the robot is available online [39]. That simple

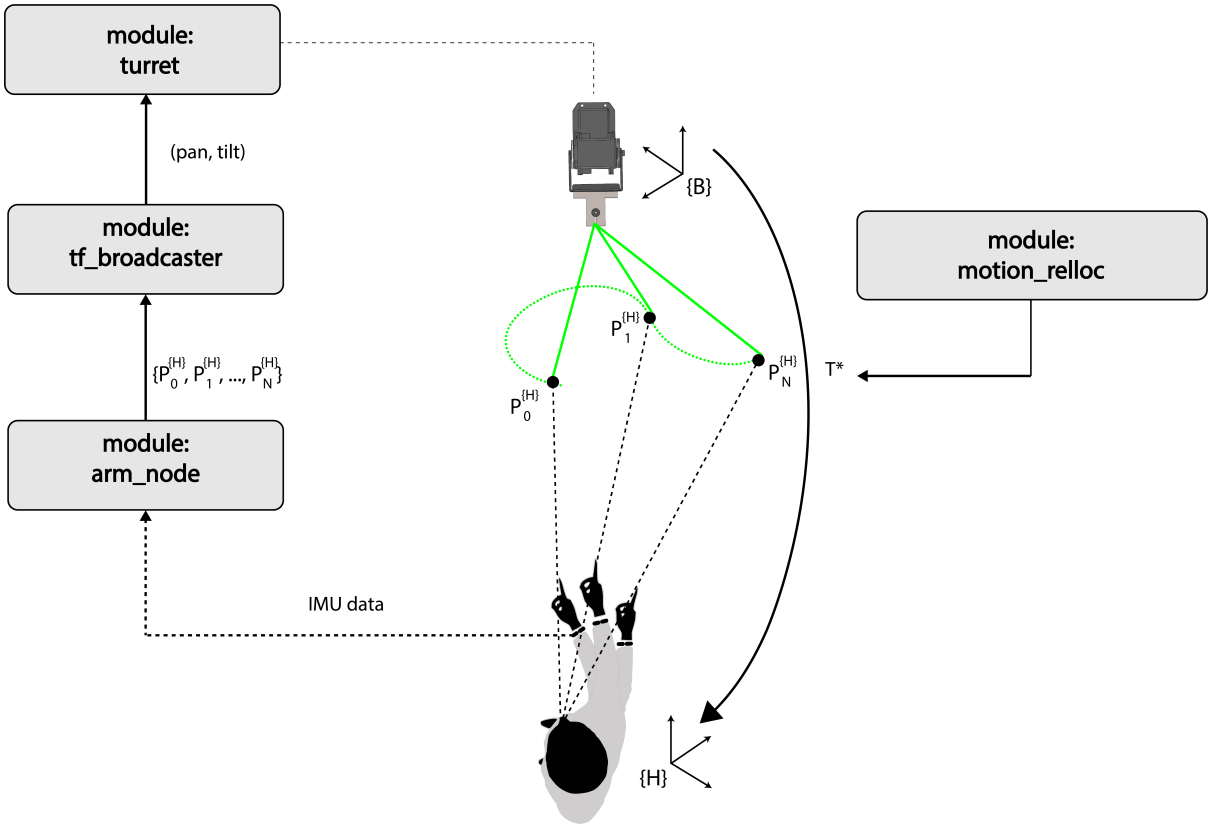

Figure 3.4: Relloc Demo

controller, contained in the **kobuki_go_to_goal** module, is very important as, while it computes the velocity command to move the kobuki, it also computes the new goal for the turret, in order to keep pointing the destination while the robot is moving, leveraging on its own odometry. Figure 3.5 tries to explain that.
To detect the fact that the user is pointing the same point for three seconds, we implemented a nice function with a fixed size queue where we store the coordinates of the last 150 laser points (which means points in the last 3 seconds at $50Hz$). We check if the average distance of each point from the first point in the queue is within a given threshold: in that case we detect the 3 seconds pointing. All that additional code is contained in the **kobuki_go_to_goal** module, which interfaces flawlessly with our system.

3.6.3 Kobuki Follow Trajectory Demo

In that case we have the same ingredients of the *Go to Goal Demo*, but the last part is different. Now the kobuki will not simply go to a goal point, but rather it will follow a trajectory drawn by the user.
To mark the begin of the trajectory, the user points the start point for three seconds. After an audio feedback, he draws the trajectory with pointing gestures and finally marks the end of it again with another three seconds pointing.
The three second pointing detection code is the same explained in 3.6.2. Moreover, we added code to sample the trajectory point list based on distance: that means that the actual trajectory followed by the kobuki is composed of points which are at a threshold distance from their predecessor. With a threshold of 0, we use all points.

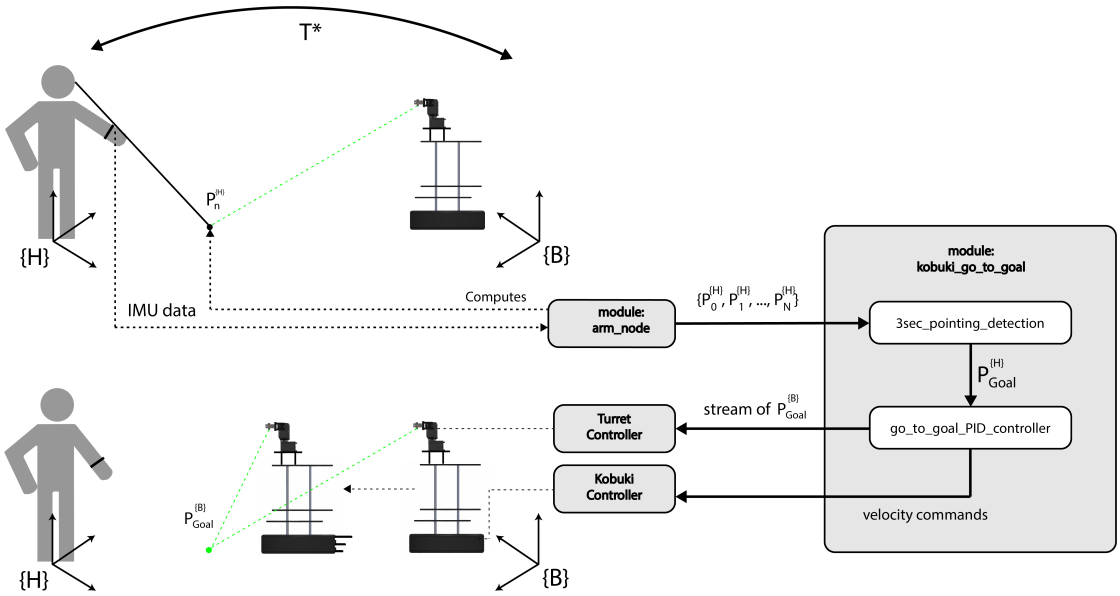

Figure 3.5: Kobuki Go to Goal Demo

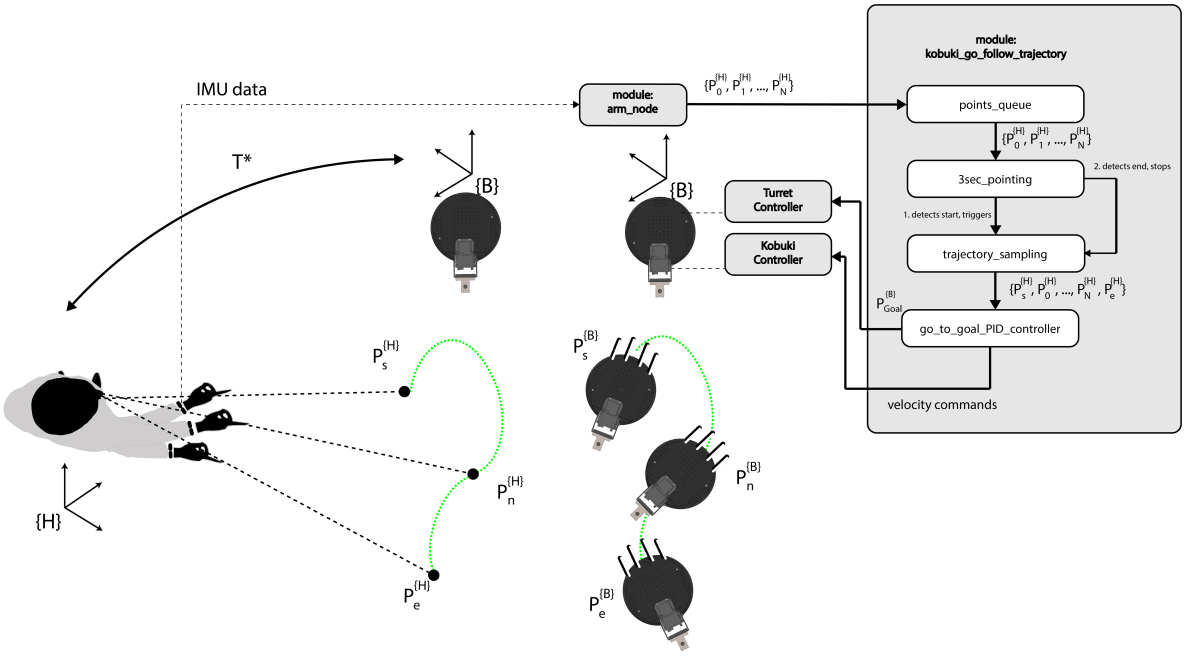

Figure 3.6: Kobuki Follow Trajecotry Demo

Chapter 4

Experiments and Applications

In this chapter describe the two main experiments we built to collect data and analyze our system. Ten people were involved in the data collection. Each experiment will be presented in terms of goals, procedures and results. Moreover, we will show a couple of applications for the system involving also a ground robot. Those applications are not only discussed, but also demonstrated with a kobuki, allowing us to draw qualitative results.

4.1 Pointing Feedback Experiment

4.1.1 Setup

We place the turret and three ground targets in known positions. The user is placed in front of the second target, oriented perpendicularly to the line connecting the targets. Figure 4.1 shows that setup. The relative localization between the human and the turret is known a priori, so we are not performing any relloc. The user is wearing the IMU and we are collecting his pointing data.
The user has to point each target first without any additional visual feedback, so he has to rely solely on his perception. Second, he has to point again with the visual feedback provided by the laser dot. Those are the steps composing each iteration of the experiment (one for each target):

- after a countdown, the user points the target without feedback;
- the user keeps pointing for 5 seconds;
- after a sound feedback, the user can rest his arm;
- after a countdown, the user points the target with the laser feedback;
- the user keeps pointing for 5 seconds;
- after a sound feedback, the user can rest his arm.

Each user has to perform that loop three time for each target, for a total of nine iterations.

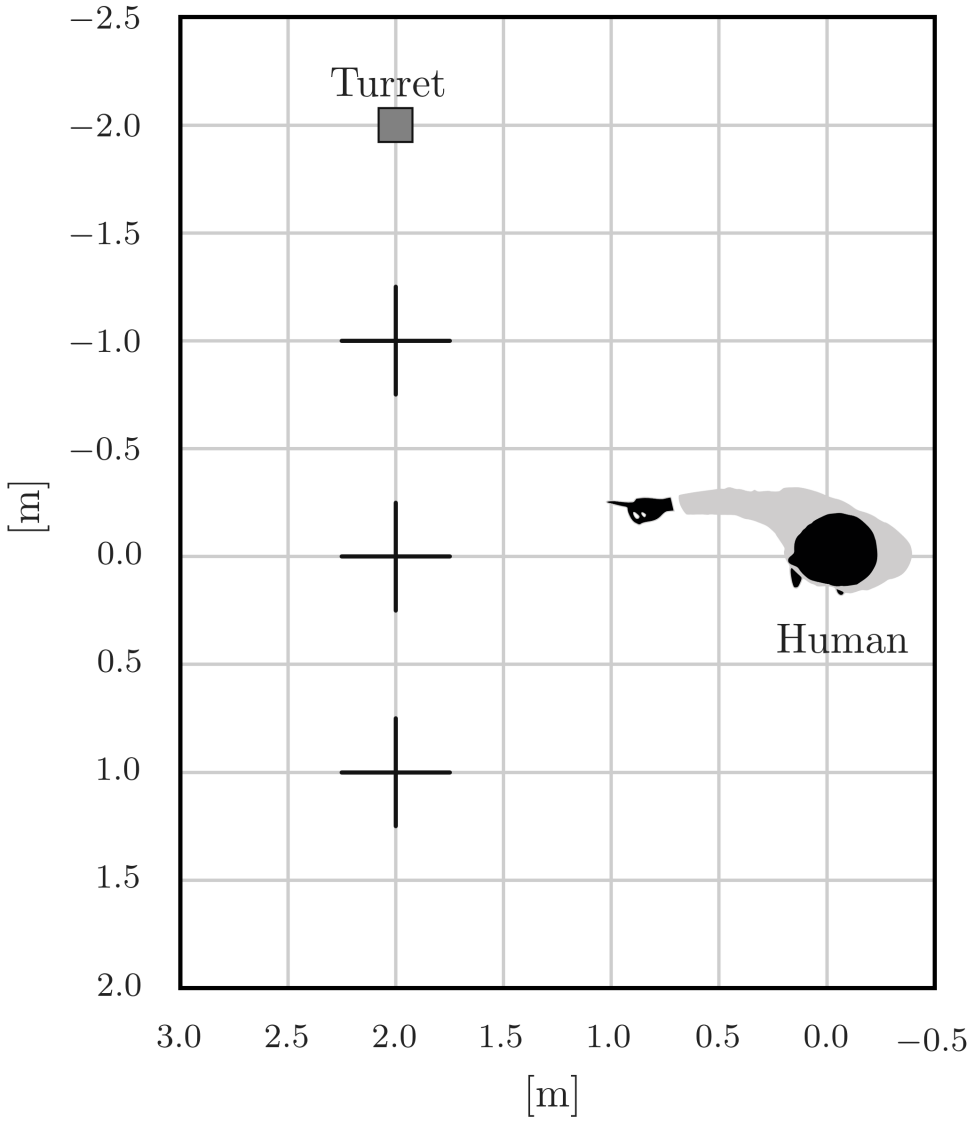

Figure 4.1: Pointing Experiment Setup (Black Crosses Are Ground Targets)

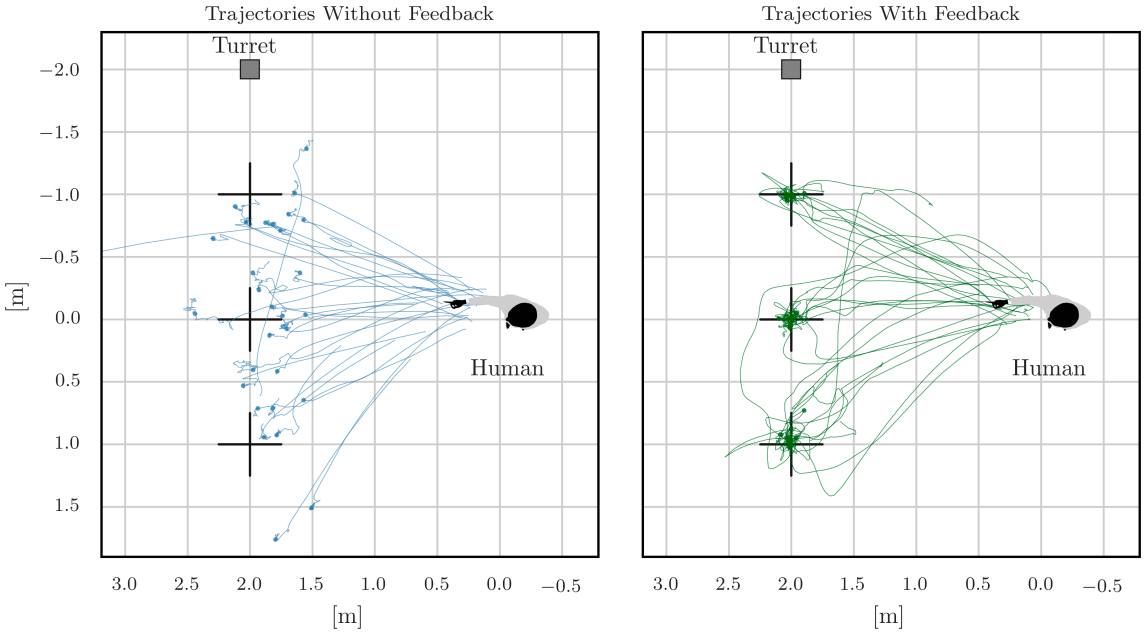

Figure 4.2: Users' Trajectories With and Without Feedback

4.1.2 Goals

The performance of the interface crucially relies on operator's perception. Due to simplifications in the pointing model that we use and various sensory errors, the estimated frame transformations and the pointing are expected to be imprecise.
With that experiment we want to understand if availability of visual feedback is useful and thus improves the pointing accuracy. This is why we ask users to drive the laser dot to the target, where the laser dot represents the location where the system thinks the user is pointing. Comparing results with and without feedback, we can understand if providing that feedback helps users to timely adapt to any misalignment.

4.1.3 Results

First, we can look at the users' trajectories with and without feedback in figure 4.2. We can immediately see that without feedback users go straight to their goal point, but that point, for the system, does not correspond to the target. With feedback, on the contrary, users are able to tell the system to point at the target. This intuition can be see in figure 4.3 and is further confirmed by figure 4.4: without feedback, users quickly reach an average distance from the target of 0.5 m but do not improve any further. When the feedback is provided, distance decreases to almost 0 within 5 seconds. This is expected as the system has intrinsic inaccuracies (for example in the reconstruction of pointing rays) which the user is unable to see and correct.

This demonstrates that real-time feedback is a key component for our system. This justifies all the work done to build the turret, as, while with fast moving robots (e.g. a drone), the robot position itself can be the feedback, for slow moving machine (e.g. a ground robot) the laser dot represents a valid possibility, as also our demos with the kobuki show.

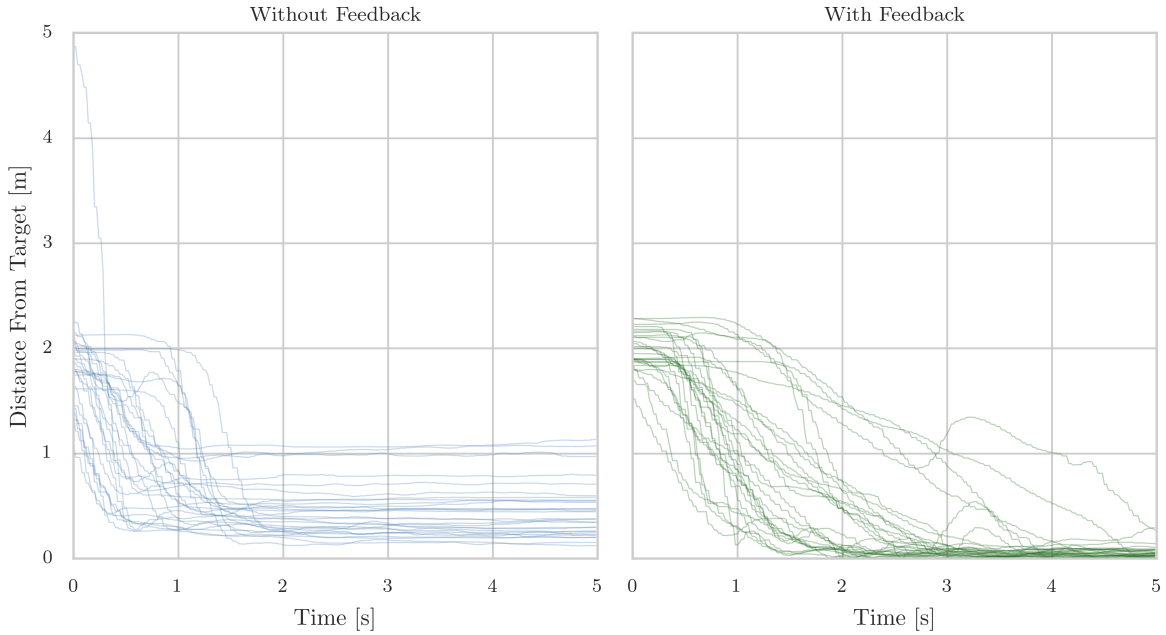

Figure 4.3: Comparison Of Distances Over Time

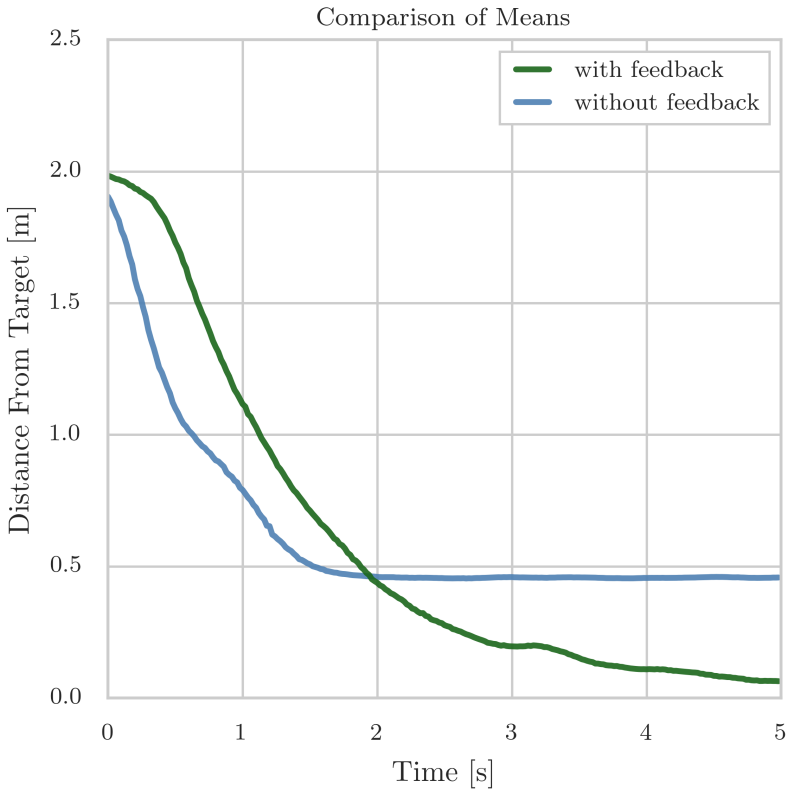

Figure 4.4: Comparison Of Average Distances Over Time

4.2 Relloc Experiment

For that experiment we will describe the setup and the goal, but will provide only results obtained with the first turret model, which heavily influenced the results with its limited performances. We have also experimented a bit with the second turret obtaining better qualitative result, but we did not have time to perform an entire data collection with users, which we plan to do in the future.

4.2.1 Setup

We perform the relloc 9 times from 9 different known positions on the floor. The order of the positions is randomly shuffled for each user. So, the experiment runs the following loop 9 time for each user:

- the turret points the position where the user must stand and starts a countdown;
- at the end of the countdown, the relloc starts: the turret draws the ∞ trajectory and the user follow it with pointing gestures;
- once the relloc is done, the turret points to the estimated location of the user.

In that way we can have an immediate visible the result: the nearer the turret is pointing to the user, the better the relloc is working. We can obtain that measure directly confronting the known initial location with the estimated one.

4.2.2 Goals

The main goals of that experiment of course is to obtain a direct estimate of the relloc precision. Moreover, it is useful to see if also inexperienced users can interface with the system easily.

4.2.3 Results

As already said, we collected data only with the first turret model. We will also run the experiment with the second turret, but from the trials we already did, we have qualitative evidence that the second should obtain better results than the first.
However, we report the average error obtained with the data collected with the first turret:

$$\begin{aligned} \text{mean\_distance\_error} &= 48.74 \text{ cm} \\ \text{mean\_x\_error} &= 36.71 \text{ cm} \\ \text{mean\_y\_error} &= 24.60 \text{ cm} \end{aligned}$$

where **mean_distance_error** is the mean of the distances between the position where the user is supposed to be when starting the relloc and the position estimated by our system. **mean_x/y_error** are the means of the differences between each coordinates.

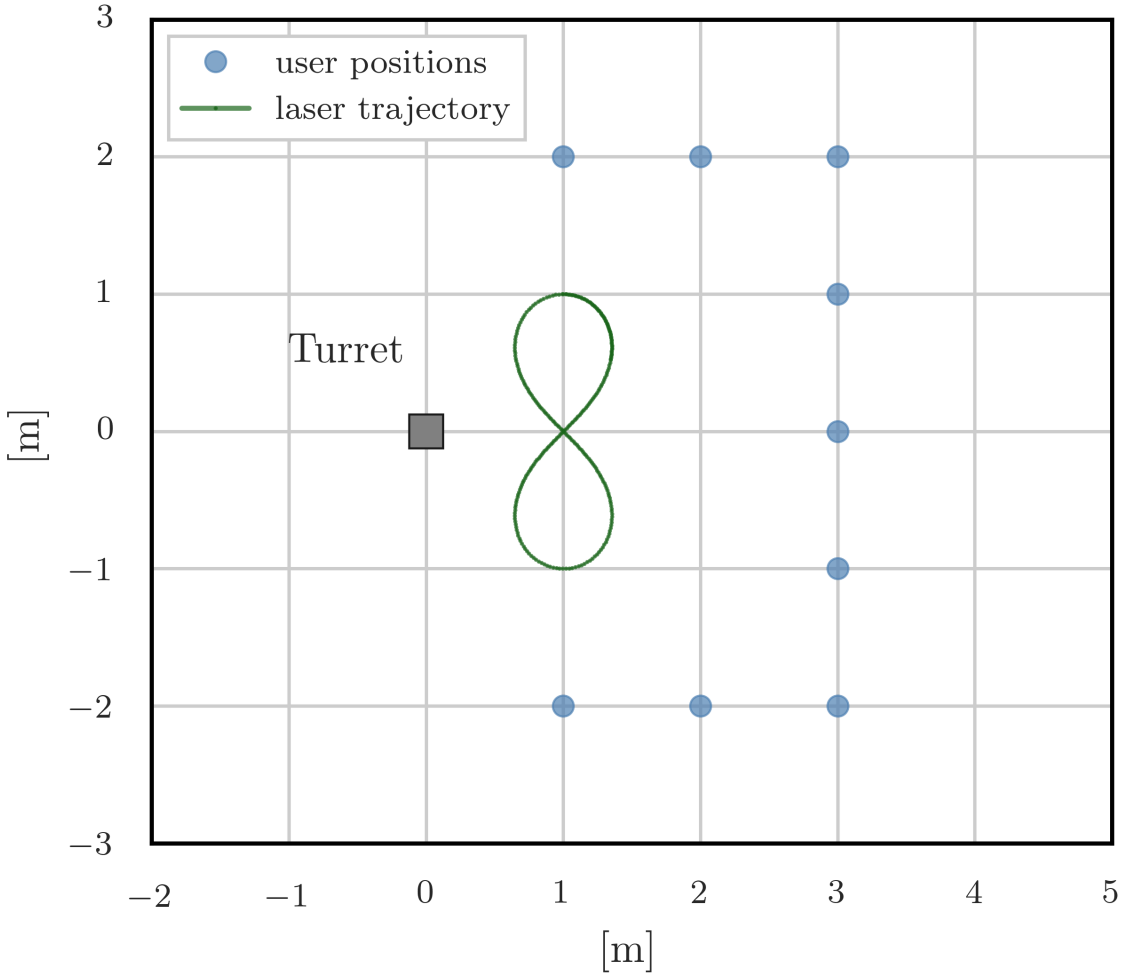

Figure 4.5: Relloc Experiment Setup

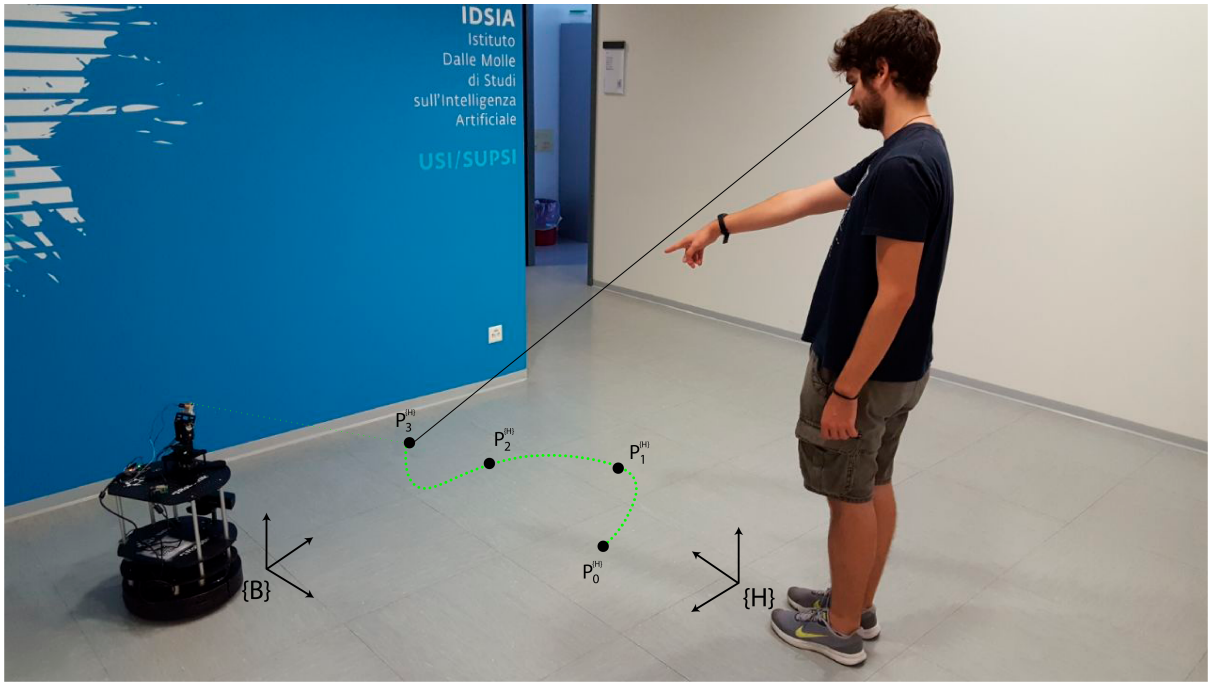

Figure 4.6: User Driving the Laser Dot After the Relloc

4.3 Applications

As already mentioned, we explored a couple of applications for our system. One consists exactly in what was done for the experiments already presented. In fact, that system is useful to collect data and study pointing models. It also serves as a tool to asses the performances and the precision of the relloc procedure.

However, in that section will we present two entire human-robot interaction scenarios involving a ground robot (the kobuki) as well as the turret.
A video containing both of them is available online [1].

4.3.1 Kobuki Go to Goal Application

The original relloc system was applied to control a flying robot [1]. In that context, the feedback user had while controlling the drone with pointing consisted of the drone itself. However, that approach is not suitable for a slow moving ground robot, as the kobuki. Here the turret system developed for this thesis comes in play: mounting it on top of the kobuki, we can face the mobile robot navigation task in a convenient way.
Top-right figure of 4.7 shows the setup of the demo: we have the user wearing the IMU device and the kobuki equipped with our turret. The steps composing the demo are the following (consider figure 4.7):

- the user does the relloc (bottom-left figure);
- an audio feedback alerts the user that the relloc is done and he can now move the laser with pointing;
- the user points the place he wants the kobuki to reach for three second (top-right figure);

[1] `https://youtu.be/hyh_5A4RXZY?t=85`

- the kobuki moves to that point, while the laser keeps pointing at it. (bottom-right figure)

That last figure gives also a qualitative feedback: the user is able to drive the kobuki through the cones easily and precisely.

4.3.2 Kobuki Follow Trajectory Application

Here we add another step to the previous go to goal application: now, by pointing for three seconds, users mark the begin of a trajectory. Then, they can draw an entire trajectory (figure 4.8) and finally, pointing again the same point for three seconds, set the end point. After that, the kobuki will follow the trajectory. Note how bad that trajectory can look again in figure 4.8. In that case, the user was drawing very slowly and with scarce accuracy. Anyway, by sampling that trajectory discarding points too close to each others, the kobuki was able to move around the two cones flawlessly (figure 4.9).
Just to sum up, here all all the step composing the application:

- the user does the relloc (as in figure 4.7);
- an audio feedback alerts the user that the relloc is done and he can now move the laser with pointing;
- the user points for three seconds the place he wants the kobuki to start the trajectory;
- after an audio feedback, the user can draw the trajectory. (figure 4.8);
- the user points for three seconds the place he wants the kobuki to end the trajectory;
- after an audio feedback which confirms the acquisition, the kobuki moves following that trajectory (figure 4.9).

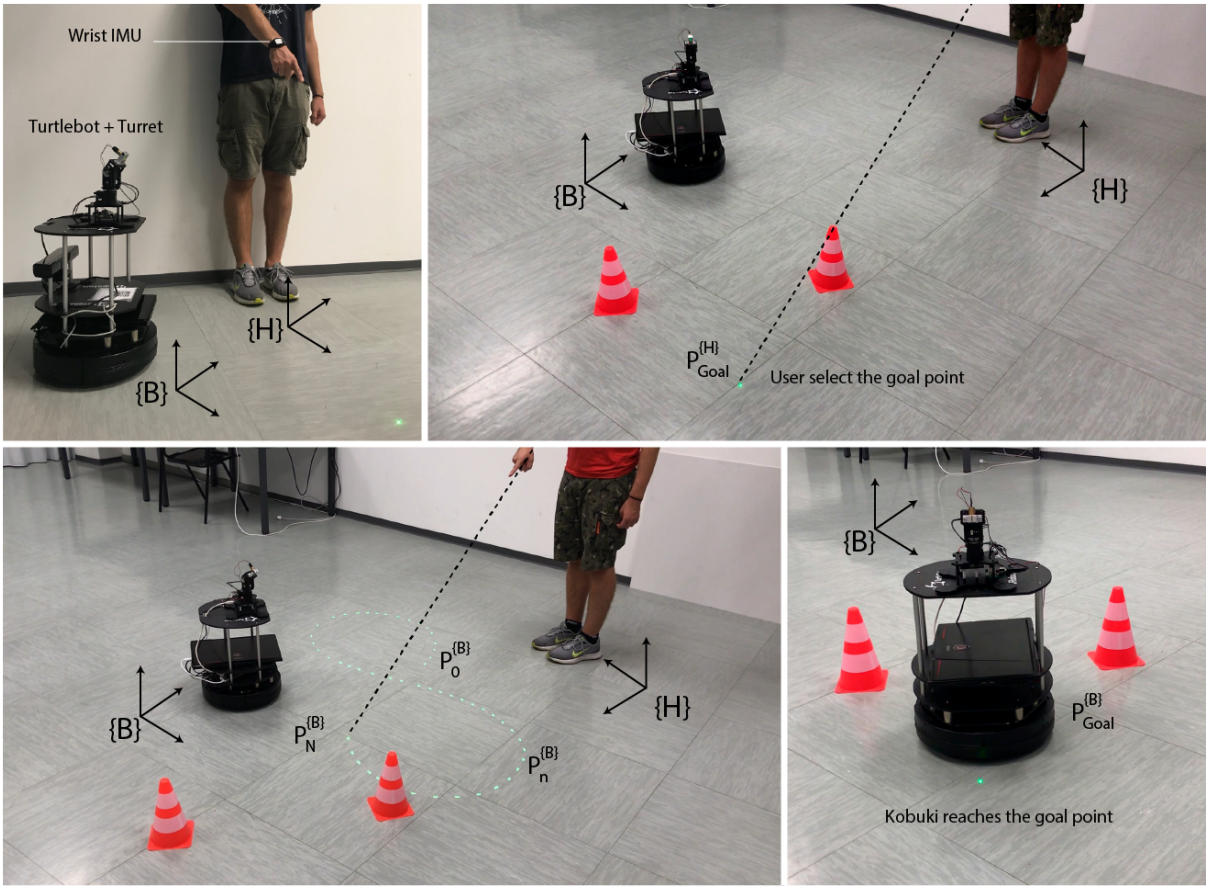

Figure 4.7: Kobuki Go to Goal. *top-left*: Setup; *bottom-left*: User Performing the Relloc; *top-right*: User Pointing Goal Point for 3 Seconds; *bottom-right*: Kobuki Reaching the Goal Point.

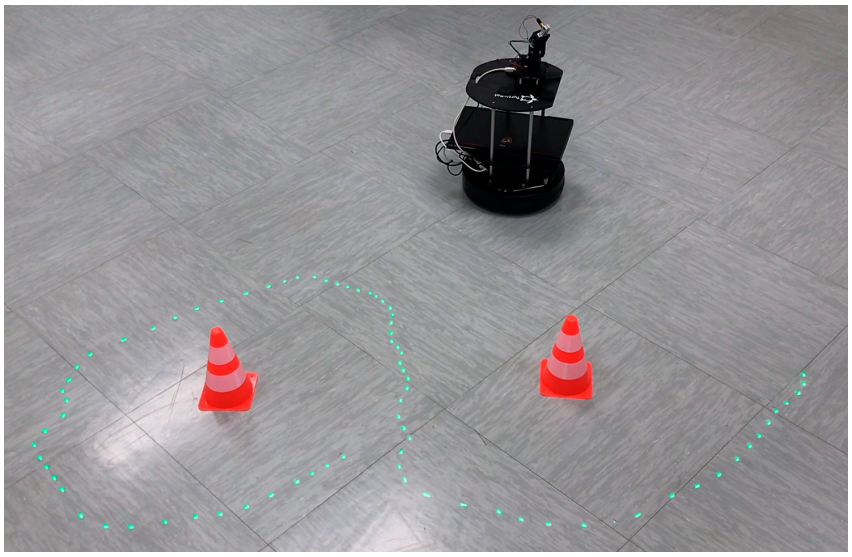

Figure 4.8: Kobuki Follow Trajectory: User's Trajectory

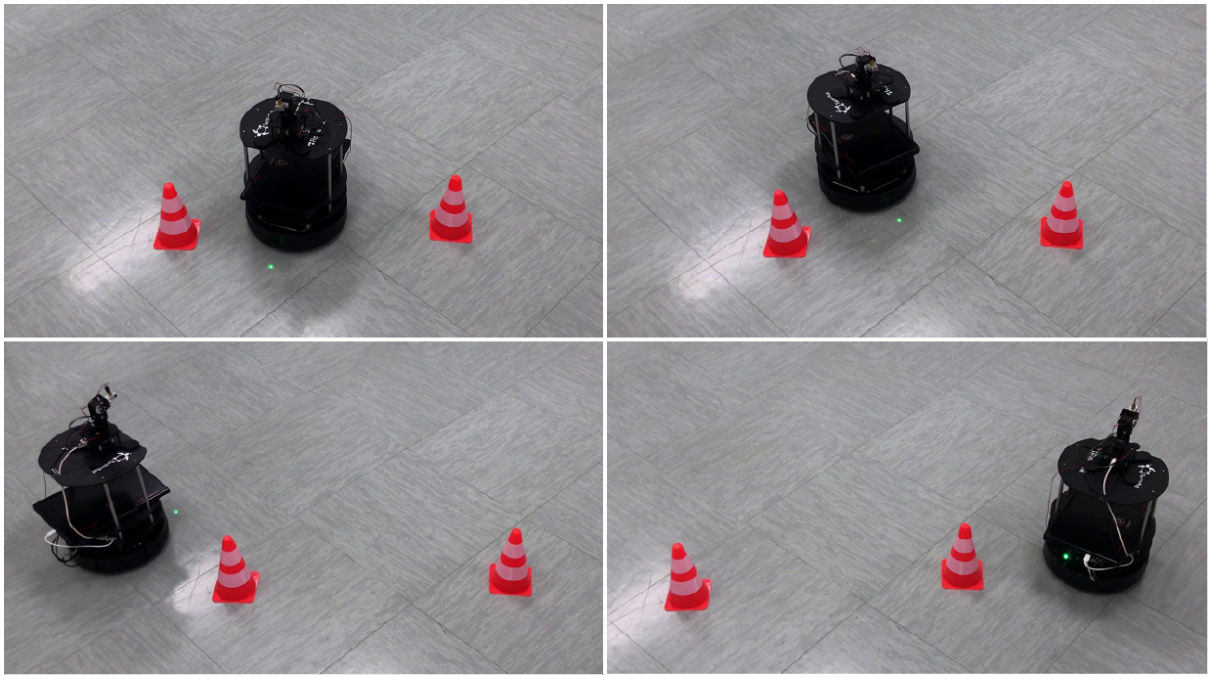

Figure 4.9: Kobuki Follow Trajectory: Kobuki Following Trajectory

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The first step of this project is to develop a pan and tilt turret able to project a laser dot on a given surface and finely control its position by solving the inverse kinematics. That goal is achieved with two different models, that we have described in that thesis both in terms of geometrical model and hardware/software implementations. The result is that, with our first turret, we are able to fulfill all our goals in a proper way. However, the second model proves to be better in performances, since it allows us to control the laser with greater precision and smoothness, being built with more expensive and powerful servos.

With the functionality of the turret validated, we are able to deploy our system for a HRI task. In fact, we integrate that work with an existing system in which an operator interacts with a drone using pointing gestures [1]. That interaction approach is suitable for fast agile robot, able to follow indicated locations in real time, but does not fit for slower or larger ground robots. Our goal is to provide a usable interface for those latter cases with our turret. So, by mounting the turret on a kobuki, we are able to solve the robot navigation task efficiently, applying the relative localization approach to the aforementioned slow ground robot and driving it around exploiting the laser pointing. Applications developed show that our approach works well in practice, as users are able to drive the robot to a goal point or through a trajectory very precisely.

Finally, the turret gives us the possibility to efficiently run experiments to validate our system components. In particular, we report two experiments: pointing experiment and relloc experiment.

The first helps us to understand whether our over-simplified human pointing model is good enough to work well. The answer is yes, as results show. Moreover, it also points out the fact that users, with a feedback for their pointing, are able to compensate any intrinsic error or misalignment of the system, giving a strong confirmation to our work. This is not surprising, because is exactly one of the motivations that we highlights analyzing related works in *Motivations and Related Works* section.

The second is useful to asses the precision of the relative localization procedure. Moreover, that experiment was run with a group of unskilled people, so it is also an indicator of how easy the procedure is to be followed. Results are quite good, considering the fact that we had time to run the data collection campaign only with the first turret, which is less accurate. Demos we developed later, which deployed the relloc with the new turret, give us qualitative evidences that this model is more precise and reliable. We even found out

that performing the relloc sometimes is more precise than manually setting the position of the user. This happens because measuring manually is of course prone to human error and, most important thing, does not take into account any inaccuracy of the system. The relloc thing, on the contrary, takes place entirely within the system environment, so, when it is precise, is able to collocate the human within the system accurately.

5.2 Future Work

5.2.1 Relloc and Pointing Experiments

As already explained many times, the relloc experiment was run only with the first turret model. We plan to conduct a data collection campaign with the second model. Being more precise, we expect to obtain better results and a more reliable indicator of the relloc accuracy. If that intuition will prove to be true, we could also rethink the pointing experiment in such a way that we do not set the position of the human by hand, but we make him run the relloc before starting with the data collection.

5.2.2 Human Body Measures

To allow users to use the system (e.g. to run experiments) we had to collect their physical data in order to build the human system properly. Since that procedure is time consuming and error prone, we would like to make it automatic. For example, we could determine approximate measures based on user's height and sex. Moreover, we could exploit the IMU sensor to determine the height. That procedure would make interfacing with the system easier and faster for new users.

5.2.3 Point to Wall (in Arbitrary Places)

As already explained in section 3.3.2, we implemented the pipeline also to drive the laser on the wall, but there are some limitations. In fact, we have to explicitly set the distance of the turret from the wall and his relative orientation. A possible further development could help to remove that requirement. For example, we could equip the turret with a coaxial laser rangefinder with pointer. In that way, we could be able to determine those initial condition autonomously and thus the wall would not need to be "calibrated" manually. That would allow the system to be deployed in arbitrary places.

Appendices

Appendix A

Interesting Stuff

In that appendix we discuss a couple of things that for different reasons are only mentioned in the main document, but can still be interesting.

A.1 Motor Issues: Solution Attempts

As we discussed in 2.1.3, we had to face many issues with the Dynamixel AX-12+ servos. Many solutions were implemented before the right one, which is explained in the thesis. Here we give an overview of all those attempts as, even if none of them completely solved our issues, developing them and understand why they were not working was a huge part of that thesis work, so they are worth to be mentioned. Here comes a brief list:

ROS Library: changing the library for ROS, using both official and unofficial. It did not help because as, how it turned out later, we had a driver problem;

Wheel Mode vs Joint Mode: those servos have the possibility to directly control their velocity (wheel mode) and not their position (joint mode, default behaviour). Thus, we implemented an open loop controller in wheel mode: that means that we compute for how long the motor must move at a certain speed to reach a given position. That was possible thanks to an external library, but it turned out that Dynamixels AX-12+ does not have a true speed controller, but just a PWM controller. That makes even harder to control the motors, as one would have to be able to compute how much power is needed to move the motor at a certain speed under certain conditions (e.g. load);

PID Controller: again using wheel mode, we tried to build a PID controller, but since the communication with the turret was very slow, that did not improve things;

PID, Joint Mode: we even tried a PID controller in joint mode, though it did not make much sense;

Compliance Slope and Margin: those servos have two internal parameters. *Compliance* is to set the control flexibility of the motor. Diagram in figure A.1 shows the relationship between output torque and position of the motor. *Compliance Margin* exists in each direction of CW/CCW and means the error between goal position and present position. The greater the value, the more difference occurs. *Compliance Slope* sets the

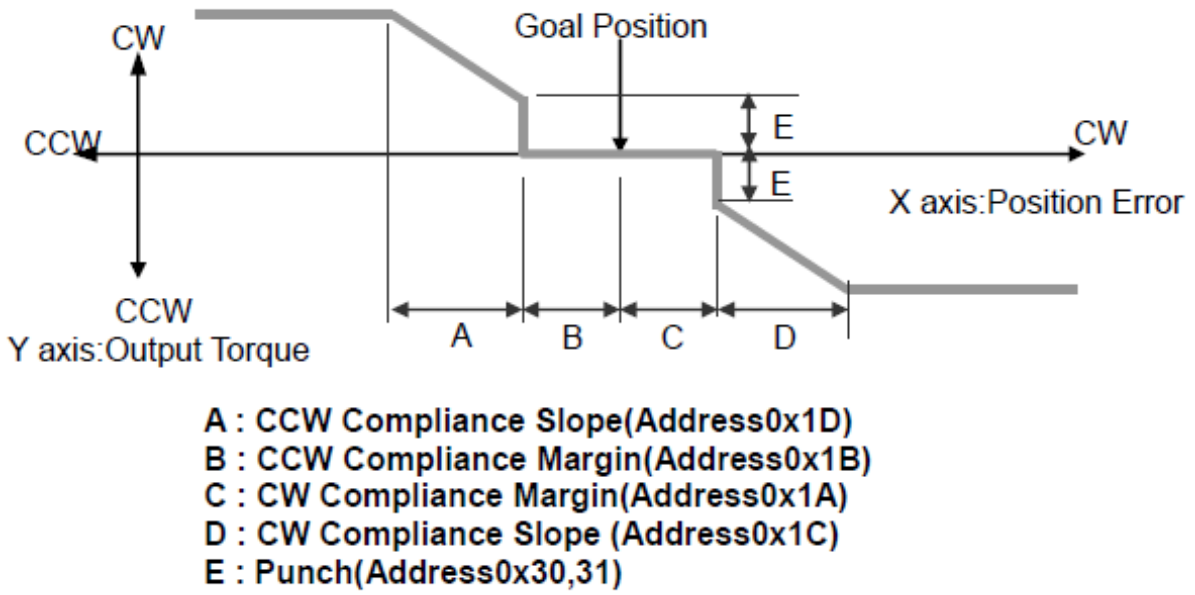

Figure A.1: Compliance Slope and Margin

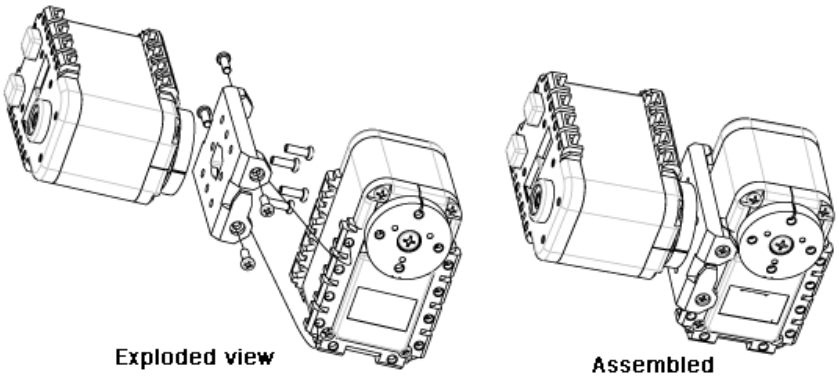

Figure A.2: Mounting F3 Frame

level of Torque near the goal position, the higher the value, the more flexibility is obtained. Changing those values allowed us to obtain smoother trajectories, but sacrificing precision which, at the earlier stage was not important and not visible, but it could not be tolerate when implementing the relative localization system, as being imprecise with the laser pointer directly affects system performances.

A.2 Motor Frames Mounting Instruction

For sake of completeness, we include the instruction followed to mount the motors together.
Figure A.2 shows how the `F3` is used to attach two dynamixels AX-12 together. In that way, we implement our pan component, since the bottom motor (the one on the left in the picture) can rotate the base of the upper one. Figure A.3 shows how the same thing can be obtained with two MX-64 servos and a `SR05-H101` frame.
Figure A.4 shows how to mount the `F2` frame on top of a AX-12 servo. This is the physical representation of our tilt angle, as we mount the laser diode exactly in the middle of that

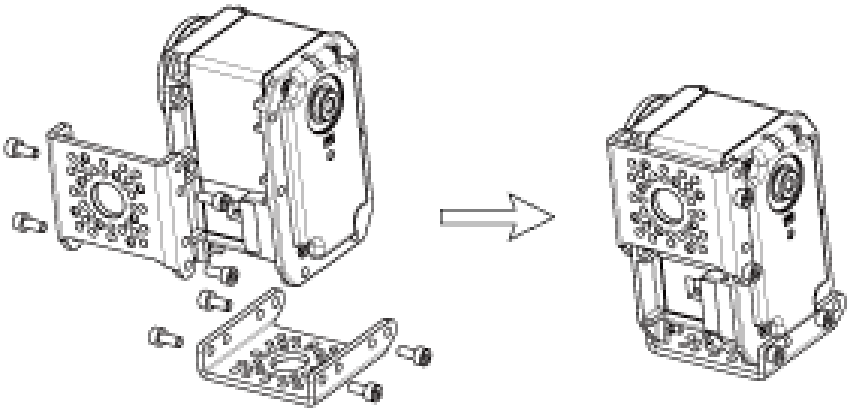

Figure A.3: Mounting FR05-S101 Side Frame

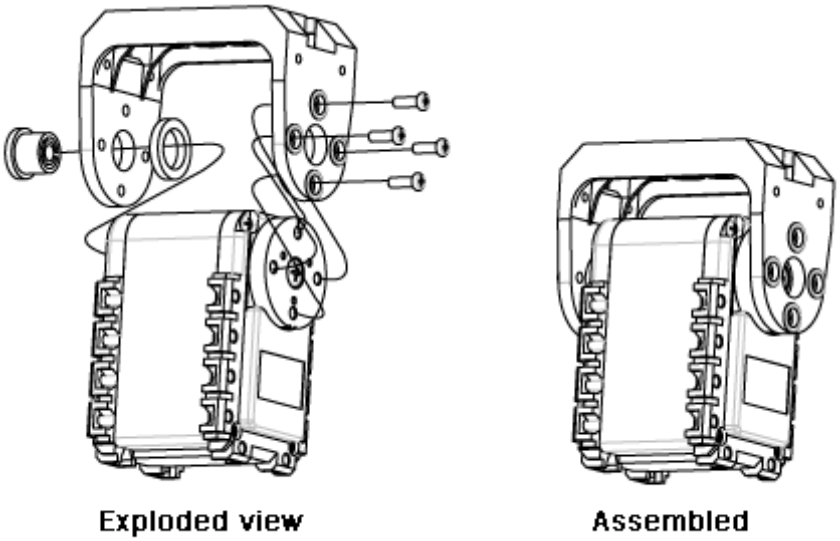

Figure A.4: Mounting F2 Frame

frame. This is very convenient since, as already discussed, mounting the laser in that way simplifies the inverse kinematic very much. Figure A.5 shows the same with a `FR05-H101` hinge frame and a MX-64 motor. Note that since that frame is slightly shorter than the `F2` counterpart, we have to mount laser differently on that motor, obtaining a different and a bit more complex inverse kinematic

A.3 Data Analysis Pipeline

In the experiment section in chapter 4 we analyze and report the result obtained with our experiments. Here we want to describe the pipeline built to collect those data, put them together and analyze them.
Of course, the main part regard data collection. Thank to ROS, we were able to recorder entire sessions from each user. *Entire* means that we can store in bag files every single message sent through ROS and then analyze the data offline or even replay the entire experiment.
From the collected bag file, we extract all the messages relevant for the experiment and store them as `pandas` dataframes. For that, a library was provided.

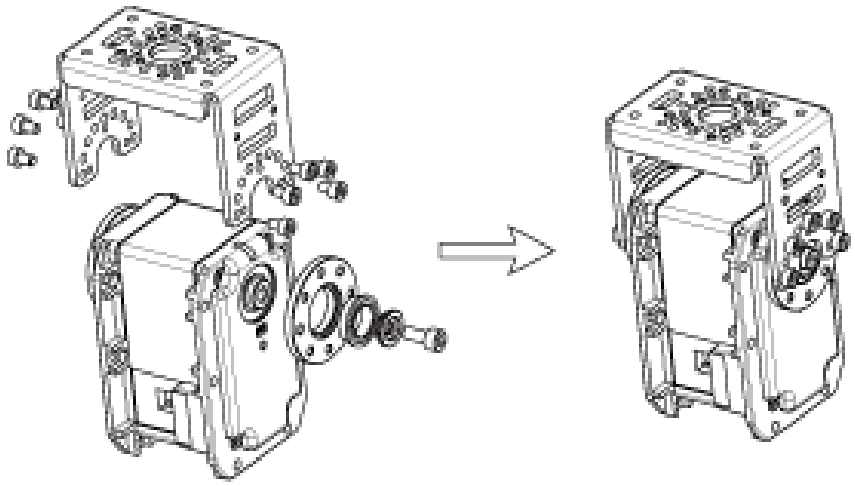

Figure A.5: Mounting FR05-H101 Hinge Frame

Here are the main topics extracted for each experiment:
pointing experiment:

- `arm_pointer`, containing estimated 3D points on the ground pointed by the human (in human frame);
- `laser_point`, containing 3D points of the laser dot;
- `target_point`, containing 3D points of the targets on the ground (published just for convenience);

relloc experiment:

- `relloc_human_pose`, containing estimated 3D points of the human in the turret frame, computed and published for application purposes;
- `gt_point`, containing 3D ground truth points, i.e. the points where the user should stand when doing the relloc. We need to publish since they are randomly shuffled for each user, thus we can not assume that we already know them.

Given that, is easy to understand how we obtain the results discussed in chapter 4. For the pointing experiment, we confront `arm_pointer` points with `laser_point` ones, considering also their distances from the targets for each session. For the relloc one, for each session we compute the distance between the ground truth point and the estimated one.
It is also worth to say that the aforementioned concept of *session* needed a bit of work to be implemented, as each dataframe needed to be parsed and each session distinguished from the others. To do that easily, we implemented a machine state mechanism which publishes a string message over a ROS topic containing the current state of the system. With a clever use of that concept, we are able to mark the begin and the end of each session and then recover it when parsing the data.
To conclude, we can add that the code to parse the data, analyze and print/plot results is available as convenient `ipython` notebooks. In that way, we can easily run and analyze new data eventually collected.

Appendix B

Code Listings

Here we report some important code listings. Note that none of them was included in the document for readability reasons. The entire code is available online [1]. Note that here we refer only to the second turret model.

tf_broadcaster module

publish_tf_tree method

This method publishes each frame composing the turret model exploiting the `tf` library. So, this the internal representation the system has of the turret. In other words, this is what moves our "virtual" model.

Listing B.1: Publish Turret tf Tree

```
1  def publish_tf_tree(self, pan, tilt):
2      self.br.sendTransform((0, 0, self.dz_to_motor_1_flange),
3                            tf.transformations.quaternion_from_euler
                                 (0, 0, pan),
4                            rospy.Time.now(), "pan_link", "base_link")
5      self.br.sendTransform((0, 0, self.dz_to_motor_2_flange),
6                            tf.transformations.quaternion_from_euler(
                                 np.pi/2, -np.pi/2, 0),
7                            rospy.Time.now(), "tilt_link", "pan_link")
8      self.br.sendTransform((0, 0, 0),
9                            tf.transformations.quaternion_from_euler
                                 (0, 0, tilt - np.pi/2),
10                           rospy.Time.now(), "laser_pointer_link", "
                                 tilt_link")
11     self.br.sendTransform((0.075, 0, 0),
12                           tf.transformations.quaternion_from_euler
                                 (0, 0, -np.pi/2),
13                           rospy.Time.now(), "laser_link", "
                                 laser_pointer_link")
14     self.br.sendTransform((5.0, 0, 0),
15                           tf.transformations.quaternion_from_euler
                                 (0, 0, 0),
```

[1] INSERTREPOHERE

```
16                          rospy.Time.now(),
17                          "debug_link", "laser_link")
18      self.rate.sleep()
```

ik method

Given a 3D (x, y, z) point, that method computes the inverse kinematic and return the corresponding pan and tilt values.

Listing B.2: Inverse Kinematic

```
1  def ik(self, x, y, z):
2          pan = np.math.atan2(y, x)
3          dz = self.dz_to_motor_1_flange + self.
              dz_to_motor_2_flange - z
4          b = np.math.sqrt(x**2 + y**2)
5          c = np.math.sqrt(dz**2 + b**2)
6          tilt1 = np.math.atan2(b, dz)
7          tilt2= np.math.atan2(c, 0.075)
8          tilt = -np.pi/2+ tilt1 + tilt2
9          return pan, tilt
```

update_target_pose_point method

This is a ROS topic callback used to update the turret model each time a new target (laser) point is received. Note that, depending on the system state, different transformation could be required to compute the correct point, as we can see in the code. In any case, we end up calling the `publish_joint_state` method (see next section) to send the goal pan and tilt angles to the turret controller interface. We also publish a laser marker in our simulation environment (`rviz`) for debug purposes (e.g. when things go bad, we can see if they were right at least in the simulator).

Listing B.3: Update Target Pose Point

```
1  def update_target_pose_point(self, msg):
2      if self.state == "following_arm":
3          self.stamped_point = PointStamped()
4          self.stamped_point.header = msg.header
5          self.stamped_point.header.stamp = rospy.Time()
6          self.stamped_point.point = msg.pose.position
7          self.target_point = self.tf.transformPoint("base_link",
               self.stamped_point)
8          self.pan_angle, self.tilt_angle = self.ik(self.
              target_point.point.x, self.target_point.point.y, self
              .target_point.point.z)
9          self.br.sendTransform((self.target_point.point.x, self.
              target_point.point.y, self.target_point.point.z),
10                               tf.transformations.
                                    quaternion_from_euler(0, 0, 0),
11                               rospy.Time.now(),
12                               "debug_point",
13                               "base_link")
```

```
14
15          laser_marker = self.create_laser_marker(self.
                target_point.point)
16          self.publish_joint_state(self.pan_angle, self.
                tilt_angle)
17          self.marker_pub.publish(laser_marker)
18      elif self.state == "marking_kobuki_goal":
19          self.stamped_point = self.kobuki_goal
20          self.stamped_point.header.stamp = rospy.Time()
21          self.target_point = self.tf.transformPoint("base_link",
                self.stamped_point)
22          self.pan_angle, self.tilt_angle = self.ik(self.
                target_point.point.x, self.target_point.point.y, self
                .target_point.point.z)
23          self.br.sendTransform((self.target_point.point.x, self.
                target_point.point.y, self.target_point.point.z),
24                                tf.transformations.
                                    quaternion_from_euler(0, 0, 0),
25                                rospy.Time.now(),
26                                "debug_point",
27                                "base_link")
28          laser_marker = self.create_laser_marker(self.
                target_point.point)
29
30          self.publish_joint_state(self.pan_angle, self.
                tilt_angle)
31          self.marker_pub.publish(laser_marker)
32      elif self.state == "marking_relloc":
33          try:
34              (trans,rot) = self.listener.lookupTransform('
                    base_link', 'human_footprint', rospy.Time(0))
35          except (tf.LookupException, tf.ConnectivityException,
                tf.ExtrapolationException):
36              pass
37          self.target_point.point.x = trans[0]
38          self.target_point.point.y = trans[1]
39          self.target_point.point.z = trans[2]
40          self.relloc_human_pos_pub.publish(self.target_point)
41          self.pan_angle, self.tilt_angle = self.ik(self.
                target_point.point.x, self.target_point.point.y, self
                .target_point.point.z)
42          self.br.sendTransform((self.target_point.point.x, self.
                target_point.point.y, self.target_point.point.z),
43                                tf.transformations.
                                    quaternion_from_euler(0, 0, 0),
44                                rospy.Time.now(),
45                                "debug_point",
46                                "base_link")
47          laser_marker = self.create_laser_marker(self.
                target_point.point)
48
```

```
49          self.publish_joint_state(self.pan_angle, self.
               tilt_angle)
50          self.marker_pub.publish(laser_marker)
```

publish_joint_state method

Method to publish pan and tilt values through standard ROS `JointState` messages on a topic. In that way, the node that actually moves the motor can subscribe to that topic and perform the action.

Listing B.4: Publish Joint State

```
1 def publish_joint_state(self, pan, tilt):
2     joint_state_msg = JointState()
3     joint_state_msg.header = Header()
4     joint_state_msg.header.stamp = rospy.Time.now()
5     joint_state_msg.name = ['pan_link', 'laser_link']
6     joint_state_msg.position = [math.degrees(pan), math.degrees
         (tilt - np.pi/2)]
7     joint_state_msg.velocity = []
8     joint_state_msg.effort = []
9     self.joint_state_pub.publish(joint_state_msg)
```

mx64_turret module

set_joint_angle method

Callback for the topic mentioned in B, which adjusts the value accounting for the true zero positions of the motors and then calls the `dynamixel_workbench` provided service to move the motors.

Listing B.5: Set Joint Angle

```
1  def set_joint_angle(self, msg):
2      angles = msg.position
3      new_pos = {}
4      new_pos_print = {}
5      vel = {}
6
7      for idx, i in enumerate(self.joint_ids):
8          new_pos[i] = angles[idx] + self.zeros[i]
9          if i == 2:
10             new_pos[i] = -angles[idx]
11         self.set_joint_angle_srv("rad", i, math.radians(new_pos
              [i]))
```

mx64_trajectory_publisher module

run_floor_trajectory

An extract of the `run_floor_method` that computes the points to draw an ∞ shape trajectory and publish each point at a fixed rate to topic subscribed by the `tf_broadcaster` module already explained in B

Listing B.6: Run ∞ Trajectory

```
def run_floor_trajectory(self, shape):
    if shape.data == '8':
        t=-90
        while t<1000:
            if not rospy.is_shutdown() and self.drawing:
                rad = math.radians(t)
                scale = 2 / (3 - math.cos(2*rad));
                y = 100*scale * math.cos(rad);
                x = self.inf_center+(100*scale * math.sin(2*rad
                    ) / 2);
                #rospy.loginfo("(x, y): (%.5f, %.5f) " % (x, y)
                    )
                msg_point = PointStamped()
                msg_point.header = Header()
                msg_point.header.stamp = rospy.Time.now()
                msg_point.point.x = x/100.0
                msg_point.point.y = -y/100.0
                msg_point.point.z = 0.0
                self.target_pub.publish(msg_point)
                t+=self.step
                self.rate.sleep()
            else:
                break
    elif ...
```

Kobuki Demos Related Code

GoalQueue Class

We implemented that convenient class to keep a fixed size queue, thus a First In First Out (FIFO) structure. Here we put the last point pointed by the human on the ground. The class provides a simple method, `goal_detected`, which detects when the user has been pointing at the same point, with a tolerance given by a threshold, for three seconds. It considers the furthest point in the queue from the first point that has entered it (the oldest), computing it with the `distance_from_oldest` method. If that point is within the distance threshold then the `goal_detected` returns `True`.

Listing B.7: Goal Queue Class

```
class GoalQueue:
    def __init__(self, size, threshold):
        self.tail = -1
```

```
4           self.MAX_SIZE = size-1
5           self.x_queue = np.zeros(size)
6           self.y_queue = np.zeros(size)
7           self.threshold = threshold
8       def enqueue(self, x, y):
9           if self.tail == self.MAX_SIZE:
10              self.x_queue = shift(self.x_queue, -1, cval = x)
11              self.y_queue = shift(self.y_queue, -1, cval = y)
12          elif self.tail < self.MAX_SIZE:
13              self.tail += 1
14              self.x_queue[self.tail] = x
15              self.y_queue[self.tail] = y
16      def distance_from_oldest(self):
17          px, py = self.x_queue[0], self.y_queue[0]
18          return np.sqrt((px - self.x_queue[1:])**2 + ((py - self
                .y_queue[1:]))**2)
19
20      def goal_detected(self):
21          max_distance = np.max(self.distance_from_oldest())
22          if max_distance <= self.threshold and self.tail == self
                .MAX_SIZE:
23              return True
24          else:
25              return False
26      def clear(self):
27          self.tail = -1
28          self.x_queue = np.zeros(self.MAX_SIZE+1)
29          self.y_queue = np.zeros(self.MAX_SIZE+1)
```

update_to_go_point method

If the kobuki is in the state in which is waiting for the goal point, that method checks, through the help of the `GoalQueue` class, if a goal point has been provided. In that case it sets the goal point and clears the queue.

Listing B.8: Update To Go Point

```
1  def update_to_go_point(self, msg):
2      if self.state == 'wait_goal_point':
3          self.goal_queue.enqueue(msg.pose.position.x, msg.pose.
               position.y)
4          if self.goal_queue.goal_detected():
5              stamped_point = PointStamped()
6              stamped_point.header = msg.header
7              stamped_point.header.stamp = rospy.Time() #= msg.
                   header
8              stamped_point.point = msg.pose.position
9              self.go_to_point = stamped_point#self.tf.
                   transformPoint("base_link", stamped_point)
10             self.state = 'goal_point_acquired'
11             self.goal_queue.clear()
```

The `kobuki_follow_trajectory` counterpart is the same for the start point acquisition. Then, it loops each following point and, after checking whether an end point is detected, it publishes it to be used later to follow the trajectory. If an end point is detected, it moves the system to the following state.

Listing B.9: Update To Go Point

```
def update_to_go_point(self, msg):
    if self.state == 'wait_start_point':
        self.goal_queue.enqueue(msg.pose.position.x, msg.pose.
           position.y)
        if self.goal_queue.goal_detected():
            stamped_point = PointStamped()
            stamped_point.header = msg.header
            stamped_point.header.stamp = rospy.Time()
            stamped_point.point = msg.pose.position
            self.kobuki_trajectory_pub.publish(stamped_point)
            self.state = 'start_point_acquired'
            self.goal_queue.clear()
    if self.state == 'acquiring_trajectory':
        self.goal_queue.enqueue(msg.pose.position.x, msg.pose.
           position.y)
        stamped_point = PointStamped()
        stamped_point.header = msg.header
        stamped_point.header.stamp = rospy.Time()
        stamped_point.point = msg.pose.position
        self.kobuki_trajectory_pub.publish(self.tf.
           transformPoint("base_link", stamped_point))
        if self.goal_queue.goal_detected():
            self.state = 'trajectory_acquired'
            self.goal_queue.clear()
```

button_cb method

Callback method to check if the button on the Metawear IMU device has been pressed. Since pressing the button once will produce more than one `True` message, we make ourselves sure to consider only the first.

Listing B.10: Button Callback

```
def button_cb(self, msg):
    if msg.data == True and self.last_button == False:
        self.button_down = msg.data
    self.last_button = msg.data
```

run method

Method that runs the interaction for the kobuki go to goal demo. Each step is triggered with the wrist IMU button and confirmed with an audio feedback. When the goal point is set, it set the state of the `tf_broadcaster` node to make the laser mark the goal point.

Listing B.11: Run Kobuki Go To Goal

```
def run(self):
    while not rospy.is_shutdown():
        try:
            print("Press the button to go\n")
            while not self.button_down:
                pass
            if self.button_down:
                self.button_down = False
                self.kobuki_reset_odom.publish()
                goal = MotionRellocGoal()
                goal.max_duration = rospy.rostime.Duration(5)
                goal.min_samples = 250
                self.relloc_client.send_goal_and_wait(goal)
                self.sound_client.playWave('/home/gabry/
                   catkin_ws/src/hmri_pt_laser/nodes/sounds/beep
                   .wav')
                self.state = 'wait_goal_point'
            while not self.state == 'goal_point_acquired':
                pass
            if self.state == 'goal_point_acquired':
                                    self.sound_client.playWave('/
               home/gabry/catkin_ws/src/    hmri_pt_laser/nodes/
               sounds/beep.wav')
                self.state = 'moving'
                self.kobuki_goal_pub.publish(self.go_to_point)
                self.state_pub.publish("marking_kobuki_goal")
        except rospy.ROSException, e:
            if e.message == 'ROS time moved backwards':
                rospy.logwarn("Saw a negative time change.
                   Resetting internal state...")
                self.reset_state()
    print("End")
```

The `kobuki_follow_trajectory` looks exactly alike, a part the fact that must pass through a couple of different states before moving.

Listing B.12: Run Kobuki Follow Trajectory

```
def run(self):
    while not rospy.is_shutdown():
        try:
            print("Press the button to go\n")
            while not self.button_down:
                pass
            if self.button_down:
                self.button_down = False
                self.kobuki_reset_odom.publish()
                goal = MotionRellocGoal()
                goal.max_duration = rospy.rostime.Duration(5)
                goal.min_samples = 250
                self.relloc_client.send_goal_and_wait(goal)
                self.sound_client.playWave('/home/gabry/
                   catkin_ws/src/hmri_pt_laser/nodes/sounds/beep
```

```
                    .wav')
15                  self.state = 'wait_start_point'
16              while not self.state == 'start_point_acquired':
17                  pass
18              if self.state == 'start_point_acquired':
19                  self.sound_client.playWave('/home/gabry/
                        catkin_ws/src/hmri_pt_laser/nodes/sounds/beep
                        .wav')
20                  self.state = 'acquiring_trajectory'
21              while not self.state == 'trajectory_acquired':
22                  pass
23              if self.state == 'trajectory_acquired':
24                  self.sound_client.playWave('/home/gabry/
                        catkin_ws/src/hmri_pt_laser/nodes/sounds/beep
                        .wav')
25                  self.kobuki_trajectory_start.publish()
26                  self.state_pub.publish("
                        following_kobuki_trajectory")
27
28          except rospy.ROSException, e:
29              if e.message == 'ROS time moved backwards':
30                  rospy.logwarn("Saw a negative time change.
                        Resetting internal state...")
31                  self.reset_state()
32      print("End")
```

Kobuki Controller

Here we have all the methods to move the kobuki to a goal point or follow a list of goal point with the same PID controller. Here we break it into pieces to explain it.

Listing B.13: Kobuki PID Controller

```
1  distance_tolerance = 0.2 #distance tolerance from goal point
       when moving
2  max_linear_speed= 0.14   #max thymio linear speed
3  max_angular_speed = 1    #max Thymio angular speed
4
5  #Parameter for linear velocity PID Controller
6  vel_P = 1
7  vel_I = 0
8  vel_D = 0
9
10 #Parameter for angular velocity PID Controller
11 ang_P = 2
12 ang_I = 0
13 ang_D = 0
14
15 """
16 Useful class to implement a PID controller
17 """
18 class PID:
```

```
19
20      def __init__(self, Kp, Ki, Kd):
21          self.Kp = Kp
22          self.Ki = Ki
23          self.Kd = Kd
24          self.last_e = None
25          self.sum_e = 0
26
27      def step(self, e, dt):
28          """ dt should be the time interval from the last method
                call """
29          if(self.last_e is not None):
30              derivative = (e - self.last_e) / dt
31          else:
32              derivative = 0
33          self.last_e = e
34          self.sum_e += e * dt
35          return self.Kp * e + self.Kd * derivative + self.Ki *
                self.sum_e
36
37  """
38  Useful method to implement angle difference
39  """
40  def angle_difference(angle1, angle2):
41      return np.arctan2(np.sin(angle1-angle2), np.cos(angle1-
            angle2))
42      assert(np.isclose(angle_difference(0.5*np.pi, 0),  0.5*np.
            pi))
43      assert(np.isclose(angle_difference(1.5*np.pi, 0), -0.5*np.
            pi))
44      assert(np.isclose(angle_difference(np.pi*-2/3, np.pi*2/3),
            np.pi*2/3))
45      assert(np.isclose(angle_difference(0, 2*np.pi), 0))
```

Here follows the kobuki class. Explanation are inserted as comments.

Listing B.14: Kobuki Class

```
1   class Kobuki:
2       def __init__(self):
3           """init"""
4           rospy.init_node('kobuki_controller', anonymous=True)
5
6           self.velocity_publisher = rospy.Publisher('/mobile_base
                /commands/velocity',
7                                                     Twist,
                                                          queue_size
                                                          =10)
8
9           # A subscriber to the topic '/odom'. self.update_state
                is called
10          # when a message of type Pose is received.
11          self.pose_subscriber = rospy.Subscriber('/odom',
```

```
12                                                  Odometry, self.
                                                       update_state)
13          #subscriber to receive goal point
14          self.goal_point_pub= rospy.Subscriber('/mobile_base/
               commands/goal_point', PointStamped, self.
               goal_point_cb)
15          self.kobuki_goal_pub = rospy.Publisher('
               kobuki_goal_point', PointStamped, queue_size=10)
16          self.state_pub = rospy.Publisher("state", String,
               queue_size = 1)
17
18          #subscriber to start follow the trajectory
19          self.kobuki_trajectory_start = rospy.Subscriber("/
               mobile_base/commands/trajectory_start", Empty, self.
               follow_trajectory)
20
21          #subscriber to receive trajectory point
22          self.kobuki_trajectory_sub = rospy.Subscriber("/
               mobile_base/commands/trajectory_point", PointStamped,
                self.trajectory_point_cb)
23          self.trajectory = []
24
25
26          self.current_pose = Pose()
27
28          #we always keep the yaw as it's useful
29          self.yaw = 0
30          self.current_twist = Twist()
31          # publish at this rate
32          self.hz =10.0
33          self.rate = rospy.Rate(self.hz)
34
35          #All the needed PID
36          self.vel_controller= PID(vel_P, vel_I, vel_D)
37          self.angle_controller = PID(ang_P, ang_I, ang_D)
38          self.dt = 1.0/self.hz
39
40          self.tf = tf.TransformListener()
41          self.br = tf.TransformBroadcaster()
42
43      #store the received trajectory point
44      def trajectory_point_cb(self, msg):
45          self.trajectory.append(msg)
46      #follow the trajectory by picking each point only if it is
           at least at 20cm from the previous one
47      def follow_trajectory(self, msg):
48          prev_point = self.trajectory[0]
49          stamped_point = self.tf.transformPoint("base_link",
               self.trajectory[0])
50          self.move2goal((stamped_point.point.x, stamped_point.
               point.y))
```

```
51          for point in self.trajectory[1:]:
52              stamped_point = self.tf.transformPoint("base_link",
                    point)
53              if self.get_distance_bw_points(prev_point.point,
                   point.point)>= 0.20:
54                  self.move2goal((stamped_point.point.x,
                       stamped_point.point.y))
55                  prev_point = point
56          self.state_pub.publish('following_target')
57          self.trajectory = []
58
59      #go to (single) goal point and tells tf_broadcaster to
           point the laser at it
60      def goal_point_cb(self, msg):
61          stamped_point = self.tf.transformPoint("base_link", msg
               )
62          self.move2goal((stamped_point.point.x, stamped_point.
               point.y))
63          self.state_pub.publish('following_target')
64
65      def update_state(self, data):
66          """A new Odometry message has arrived. See Odometry msg
                definition."""
67
68          self.current_pose = data.pose.pose
69          self.current_twist = data.twist.twist
70          quat = (
71              self.current_pose.orientation.x,
72              self.current_pose.orientation.y,
73              self.current_pose.orientation.z,
74              self.current_pose.orientation.w)
75          (roll, pitch, yaw) = euler_from_quaternion (quat)
76          self.yaw = yaw
77
78      #return the distance between the current position of the
           kobuki (obtained with odometry) and a goal point
79      def get_distance(self, goal_x, goal_y):
80          distance = sqrt(pow((goal_x - self.current_pose.
               position.x), 2) + pow((goal_y - self.current_pose.
               position.y), 2))
81          return distance
82      def get_distance_bw_points(self, point, other_point):
83          distance = sqrt(pow((point.x - other_point.x), 2) + pow
               ((point.y - other_point.y), 2))
84          return distance
85
86      #Use 2 PID controller to move the kobuki to a goal point (
           one for linear and one for angular velocity)
87      def move2goal(self, moves):
88          goal_pose = Pose()
89          rospy.loginfo(moves)
```

```
90
91          goal_pose.position.x = moves[0]
92          goal_pose.position.y = moves[1]
93
94          vel_msg = Twist()
95
96          while self.get_distance(goal_pose.position.x, goal_pose
               .position.y) >= distance_tolerance:
97
98              #Porportional Controller
99              #linear velocity in the x-axis:
100             dist_error = self.get_distance(goal_pose.position.x
                   , goal_pose.position.y)
101             vel_msg.linear.x = np.clip(self.vel_controller.step
                   (dist_error, self.dt), 0.0, max_linear_speed)
102             vel_msg.linear.y = 0
103             vel_msg.linear.z = 0
104
105             #angular velocity in the z-axis:
106             vel_msg.angular.x = 0
107             vel_msg.angular.y = 0
108             vel_msg.angular.z = np.clip(self.angle_controller.
                   step(angle_difference(atan2(goal_pose.position.y
                   - self.current_pose.position.y, goal_pose.
                   position.x - self.current_pose.position.x), self.
                   yaw), self.dt), -max_angular_speed,
                   max_angular_speed)
109
110             #Publishing our vel_msg
111             self.velocity_publisher.publish(vel_msg)
112             self.rate.sleep()
113         #Stopping our robot after the movement is over
114         vel_msg.linear.x = 0
115         vel_msg.angular.z =0
116         self.velocity_publisher.publish(vel_msg)
117
118 if __name__ == '__main__':
119
120     kobuki = Kobuki()
121
122     try:
123         while not rospy.is_shutdown():
124             rospy.spin()
125     except rospy.ROSInterruptException:
126         system.exit(1)
```

References

[1] B. Gromov, L. Gambardella, and A. Giusti, "Robot identification and localization with pointing gestures," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct. 2018. to appear.

[2] M. T. Wolf, C. Assad, M. T. Vernacchia, J. Fromm, and H. L. Jethani, "Gesture-based robot control with variable autonomy from the JPL BioSleeve," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 1160–1165, 2013.

[3] J. Cacace, A. Finzi, V. Lippiello, M. Furci, N. Mimmo, and L. Marconi, "A control architecture for multiple drones operated via multimodal interaction in search & rescue mission," *SSRR 2016 - International Symposium on Safety, Security and Rescue Robotics*, pp. 233–239, 2016.

[4] B. Gromov, L. M. Gambardella, and A. Giusti, "Video: Landing a drone with pointing gestures," in *Companion of the 2018 ACM/IEEE International Conference on Human-Robot Interaction*, pp. 374–374, ACM, 2018.

[5] V. Villani, L. Sabattini, G. Riggio, C. Secchi, M. Minelli, and C. Fantuzzi, "A Natural Infrastructure-Less Human - Robot Interaction System," vol. 2, no. 3, pp. 1640–1647, 2017.

[6] B. Gromov, L. M. Gambardella, and G. A. Di Caro, "Wearable multi-modal interface for human multi-robot interaction," *2016 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, pp. 240–245, 2016.

[7] A. G. Brooks and C. Breazeal, "Working with Robots and Objects: Revisiting Deictic Reference for Achieving Spatial Common Ground," *Gesture*, pp. 297–304, 2006.

[8] D. Droeschel, J. Stückler, and S. Behnke, "Learning to interpret pointing gestures with a time-of-flight camera," *Proceedings of the 6th international conference on Human-robot interaction - HRI '11*, pp. 481–488, 2011.

[9] B. Großmann, M. R. Pedersen, J. Klonovs, D. Herzog, L. Nalpantidis, and V. Krüger, "Communicating Unknown Objects to Robots through Pointing Gestures," in *Advances in Autonomous Robotic Systems 15th Annual Conference, TAROS 2014*, (Birmingham), pp. 209–220, Springer, 2014.

[10] A. Cosgun, A. J. B. Trevor, and H. I. Christensen, "Did you Mean this Object?: Detecting Ambiguity in Pointing Gesture Targets," in *HRI'15 Towards a Framework for Joint Action Workshop*, 2015.

[11] M. Pateraki, H. Baltzakis, and P. Trahanias, "Visual estimation of pointed targets for robot guidance via fusion of face pose and hand orientation," *Computer Vision and Image Understanding*, vol. 120, pp. 1–13, 2014.

[12] D. Akkil and P. Isokoski, "Accuracy of Interpreting Pointing Gestures in Egocentric View," in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pp. 262–273, 2016.

[13] J. Nagi, A. Giusti, L. M. Gambardella, and G. A. Di Caro, "Human-swarm interaction using spatial gestures," in *IEEE International Conference on Intelligent Robots and Systems*, pp. 3834–3841, 2014.

[14] S. Pourmehr, V. Monajjemi, J. Wawerla, R. Vaughan, and G. Mori, "A robust integrated system for selecting and commanding multiple mobile robots," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 2874–2879, 2013.

[15] M. Van den Bergh, D. Carton, R. De Nijs, N. Mitsou, C. Landsiedel, K. Kuehnlenz, D. Wollherr, L. Van Gool, and M. Buss, "Real-time 3D hand gesture interaction with a robot for understanding directions from humans," *Proceedings - IEEE International Workshop on Robot and Human Interactive Communication*, pp. 357–362, 2011.

[16] S. Abidi, M. Williams, and B. Johnston, "Human pointing as a robot directive," *ACM/IEEE International Conference on Human-Robot Interaction*, pp. 67–68, 2013.

[17] A. Jevtić, G. Doisy, Y. Parmet, and Y. Edan, "Comparison of Interaction Modalities for Mobile Indoor Robot Guidance: Direct Physical Interaction, Person Following, and Pointing Control," *IEEE Transactions on Human-Machine Systems*, vol. 45, no. 6, pp. 653–663, 2015.

[18] M. Tölgyessy, M. Dekan, F. Duchoň, J. Rodina, P. Hubinský, and L. Chovanec, "Foundations of Visual Linear Human-Robot Interaction via Pointing Gesture Navigation," *International Journal of Social Robotics*, vol. 9, no. 4, pp. 509–523, 2017.

[19] M. Monajjemi, S. Mohaimenianpour, and R. Vaughan, "UAV, come to me: End-to-end, multi-scale situated HRI with an uninstrumented human and a distant UAV," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, no. Figure 1, pp. 4410–4417, IEEE, oct 2016.

[20] K. Nickel and R. Stiefelhagen, "Pointing Gesture Recognition based on 3D-Tracking of Face , Hands and Head Orientation Categories and Subject Descriptors," *Proceedings of the 5th international conference on Multimodal interfaces*, pp. 140–146, 2003.

[21] J. Sugiyama and J. Miura, "A wearable visuo-inertial interface for humanoid robot control," in *ACM/IEEE International Conference on Human-Robot Interaction*, pp. 235–236, IEEE, mar 2013.

[22] R. A. Bolt, ""Put-that-there": Voice and Gesture at the Graphics Interface," *Proceedings of the 7th annual conference on Computer graphics and interactive techniques - SIGGRAPH '80*, pp. 262–270, 1980.

[23] A. Giusti, J. Nagi, L. Gambardella, and G. A. Di Caro, "Cooperative sensing and recognition by a swarm of mobile robots," *IEEE International Conference on Intelligent Robots and Systems*, pp. 551–558, 2012.

[24] Z. Zivkovic, V. Kliger, R. Kleihorst, A. Danilin, B. Schueler, G. Arturi, C.-C. Chang, and H. Aghajan, "Toward low latency gesture control using smart camera network," in *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, pp. 1–8, IEEE, 2008.

[25] K. Nickel and R. Stiefelhagen, "Visual recognition of pointing gestures for human-robot interaction," *Image and Vision Computing*, vol. 25, no. 12, pp. 1875–1884, 2007.

[26] J. L. Taylor and D. McCloskey, "Pointing," *Behavioural Brain Research*, vol. 29, pp. 1–5, jul 1988.

[27] O. Herbort and W. Kunde, "Spatial (mis-) interpretation of pointing gestures to distal spatial referents," *Journal of Experimental Psychology: Human Perception and Performance*, vol. 42, no. 1, pp. 78–89, 2016.

[28] J. O'Rourke, *Computational Geometry in C (2Nd Ed.)*. New York, NY, USA: Cambridge University Press, 1998.

[29] "Dynamixel official website." `http://www.robotis.us/dynamixel/`. Accessed: 2018-10-1.

[30] "Dynamixel ax-12+ datasheet." `http://support.robotis.com/en/product/actuator/dynamixel/ax_series/dxl_ax_actuator.htm`. Accessed: 2018-10-1.

[31] "Scorpionx mx-64 robot turret kit." `https://www.trossenrobotics.com/p/ScorpionX-RX-64-robot-turret.aspx`. Accessed: 2018-10-1.

[32] "Mx64 description website." `https://www.trossenrobotics.com/p/mx-64t-dynamixel-robot-actuator.aspx`. Accessed: 2018-10-1.

[33] "Dynamixel mx-64 datasheet." `http://support.robotis.com/en/product/actuator/dynamixel/mx_series/mx-64at_ar.htm`. Accessed: 2018-10-1.

[34] "Kobuki web site." `http://kobuki.yujinrobot.com/about2/`. Accessed: 2018-10-1.

[35] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[36] "Ros core-components." `http://www.ros.org/core-components/`. Accessed: 2018-10-1.

[37] "Numpy." `http://www.numpy.org`. Accessed: 2018-10-1.

[38] "Matplotlib." `https://matplotlib.org`. Accessed: 2018-10-1.

[39] "Kobuki ros library." `http://wiki.ros.org/kobuki`. Accessed: 2018-10-1.

Acronyms

UAV	Unmanned Aerial Vehicle
ROS	Robot Operating System ... III
HRI	Human-Robot Interaction ... VII
ToF	time-of-flight ... VIII
RGB	Red Green Blue
RGB-D	Red Green Blue Depth
IMU	Inertial Measurement Unit ... VIII
DPI	Direct Physical Interaction ... IX
SLAM	Simultaneous Localization and Mapping ... X
EMGs	electro-myography sensors ... X
PID	Proportional–Integral–Derivative controller
DoF	Degree of Freedom ... 1
relloc	relative localization ... 28
FIFO	First In First Out ... 55

Thanks to

todo