EPFL

CHILI
computer-human interaction
in learning and instruction

LEARN
Center for Learning Sciences

Making Reachy the robot use a touchscreen

Author:
Elisa Bianchi

Supervisor(s):
Utku Norman, Barbara Bruno

Professor:
Pierre Dillenbourg

January 6, 2022

Contents

1 Introduction

We have been playing alongside and against robots and computers for quite some time now. Whether it's the unbeatable final boss or the trusted sidekick, computers and algorithms have been teaching users how to play and think in regard to its respective games. This help, however, is visible through the screen, and only the consequences of the actions are observable by the human.

The goal of this project is to bridge this gap, make the robot and the human construct a solution together, in particular through the collaborative problem-solving activity JUSThink. This activity aims at improving the computational thinking skills of the human by collaborating with a humanoid robot, Reachy, to solve an unfamiliar problem on networks. Thus, in this project, our objective is to upgrade the motor skills of Reachy, so that it can physically use the touchscreen as it takes part in the activity and interacts with the human.

The overall process can be broken down into three main steps : firstly, the calibration of the screen, so that Reachy knows where the screen is located in relation to itself. Then, the data received from the screen needs to be preprocessed, and adjusted to Reachy's coordinate system. Once we have all the screen's information as well as the destination points, we can direct Reachy to physically touch the point. We will include some analysis on the accuracy and precision of the touched points. The final step is to link this to the JUSThink activity, which then fulfils the goal of the project.

1.1 Reachy

The robot we are working with, Reachy, is a humanoid open source robot developed by Pollen Robotics [6]. It is equipped with two 7 degrees of freedom arms and one gripper hand "Otis". This allows for a precise control of the arms, and can be used to act on the screen the same way as the human does : by actually touching/tapping on the touchscreen.

1.2 Screen-app

The screen-app [3] is an application developed by Mr. Norman to monitor the touchscreen. It includes in particular a ROS2 node with a publisher, which publishes on a topic every time the touchscreen is pressed, released, and dragged (topics respectively named : mouse_press, mouse_release, mouse_drag). By subscribing to the mouse_press topic, we can collect information on the screen coordinates of the pressed points. It will be mainly used for the calibration step of the project.

1.3 JUSThink

JUSThink is the main application for which we want Reachy to physically interact with the touchscreen. Developed by the CHILI Lab, it consists of creating a network for the Swiss train routes to connect all cities with the least cost involved. A human player and a robot take turns, each constructing one route, and hopefully ending with an optimal

solution. An overview of the project's architecture is depicted in Figure 1. Once the calibration has been completed, we can connect the game and Reachy. This works by having a ROS2 node with a publisher, who publishes onto the intended_points topic the array of points Reachy must press on (representing the pixel location of the two cities we want to connect). In parallel, the subscriber to the intended_points topic fetches the data from it. This new data is transmitted to the screen_touch function, which processes it and calls the adequate functions to make Reachy move.

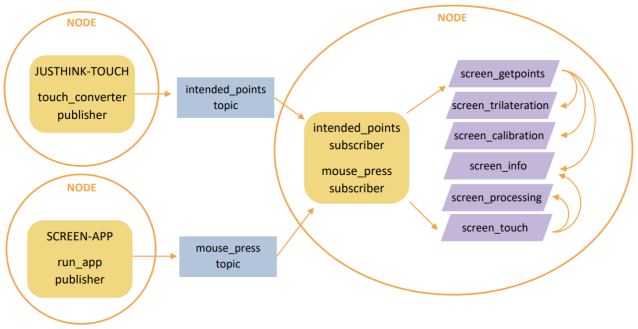

Figure 1: Project Architecture

2 Screen Calibration

The first logical step in this project is for Reachy to be able to know where the touchscreen is located in relation to its own coordinate system. This step is crucial, as it ensures that no matter the screen's position (tilt, shift, rotation) as long as it is in Reachy's workspace it can interact with it, and this will not have negative consequences on Reachy's touch capabilities. Moreover, to make this process as modular and simple as possible, the calibration points are any three points on the screen, not necessarily the corners.

2.1 Collecting Data-points

To start off, Reachy needs to gather some information on the screen. This step requires help from the human, who will guide Reachy in compliant mode and make it touch three pairwise-non-colinear points on the screen. Compliant mode means that all the robot's motors are turned off, and one can make Reachy move freely without any resistance. Upon touching the screen, we simultaneously gather the x and y coordinates of the pressed point in the screen's coordinate system by listening to the mouse_press topic (noted : S_x_i, S_y_i), as well as the x, y and z positions in Reachy's coordinate systems from the forward-kinematics function [1] (noted : R_x_i, R_y_i and R_z_i). This process is repeated three times, to gather three calibration points, which we can name A, B and C.

As the interaction is supposedly between Reachy and a human, we only consider the case where the screen is laying flat on a table. To ensure that this is the case, we compare A, B and C's z-position. If the difference between the three z-positions is too great (here

$\geq$ 5 cm), we ask the user to lay the screen on a flat surface and restart the calibration. Otherwise, we store the average value as the fixed_z. From these three points, we can also compute some of the screen's basic information, which are the pixel-size and the screen-size. For the latter, we only need to know the screen's resolution, here 1080×1920px.

We firstly compute the "Reachy distance" between two points I and J, meaning the distance in meters based on Reachy's coordinates system :

$$reachy\_IJ = \sqrt{(R\_x_j - R\_x_i)^2 + (R\_y_j - R\_y_i)^2} \quad m \tag{1}$$

As well as "screen distance" between two points I and J, meaning the distance in pixels based on the screen's coordinates system :

$$screen\_IJ = \sqrt{(S\_x_j - S\_x_i)^2 + (S\_y_j - S\_y_i)^2} \quad px \tag{2}$$

With these two distances, we can now use the three points A, B and C we just gathered to compute the pixel-size and the screen-size, as follows :

$$px\_size = \frac{1}{3} \cdot \left[\frac{reachy\_AB}{screen\_AB} + \frac{reachy\_AC}{screen\_AC} + \frac{reachy\_CB}{screen\_CB} \right] \quad m \cdot px^{-1} \tag{3}$$

$$\begin{aligned} screen\_size &= (screen\_size\_x\_px * px\_size, screen\_size\_y\_px * px\_size) \\ &= (1080 * px\_size, 1920 * px\_size) \end{aligned} \tag{4}$$

2.2 Screen-corners Trilateration

Once three data-points have been collected, we can now trilaterate the screen corners to compute the screen's coordinate axis. Trilateration is a mathematical method of determining the relative position of a point using the geometry of triangles, just like triangulation. Unlike the latter, which uses angles and distances to position a point, trilateration uses the distances between a minimum of two reference points. We use three points to get one single possible solution, as depicted in Figure 2.

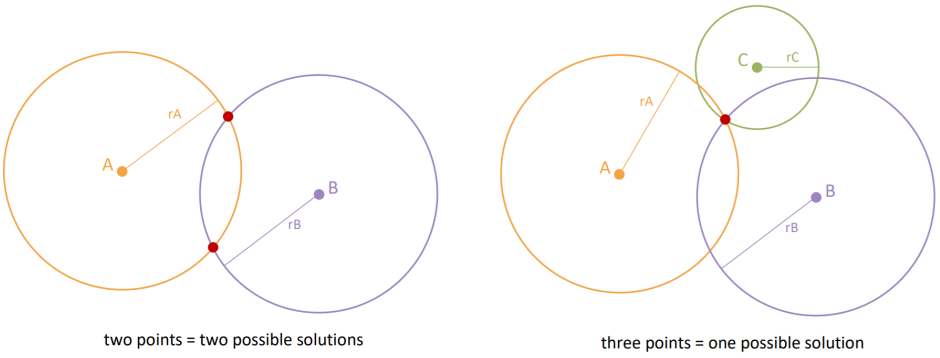

Figure 2: Two vs. Three points trilateration

The computation for the trilateration works by expecting the screen-size distance between a point I and the goal-point J to be proportionate to the distance Reachy has to move between those two same points, meaning :

$$reachy\_IJ = screen\_IJ * px\_size \tag{5}$$

The following calculation is reiterated three times, with the corners of the screen being the goal points. They are D(0, 0), E(0, 1920) and F(1080, 0). Since we know the goal point screen coordinates, we can use the above formulae to derive the reachy distance between A, B, C and the end points. We then calculate two simple two-circle intersections, using A, B and C's Reachy coordinates as the circle origins, and the distances as the circle's respective radiuses, giving us four potential points. Due to the fact that we are working with floating points from Reachy's forward kinematics function, the intersection is not exact. Therefore, we take the two closest points as the intersection, which represent the Reachy coordinates of the point we wanted. We illustrate this process for point D in Figure 3.

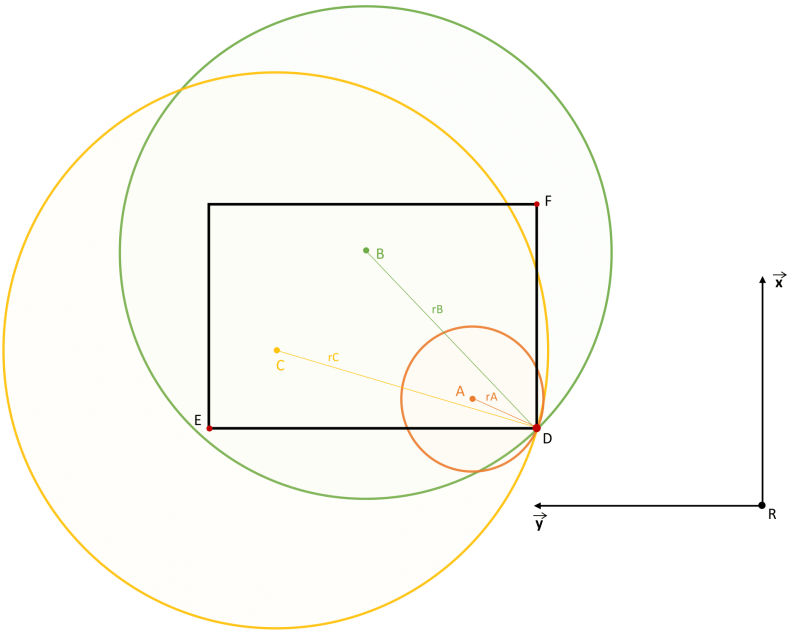

Figure 3: Example of trilateration for point D(0, 0)

Manual computation was done to ensure that the code was outputting correct intersection points. Using GeoGebra geometry [5], over 20 three-circle intersection combinations were tested, and the code's output was compared to the geometric solution, which matched up 100% of the time, given some floating point margin.

3 Data Processing

Now that Reachy has acquired some sense of where the screen is, and some basic information on it, we can look into how to adapt the information so that Reachy knows where the point we want it to press is located. More precisely, we can look into how to go from the screen coordinate system to Reachy's, and vice-versa. This is done through some basic linear algebra matrix manipulation.

We firstly want to compute how to go from Reachy's coordinate system to the screen's system. Since we assume that the screen lays flat, and it cannot be distorted (the screen is always a rectangle), we are only looking at translation and rotation matrices.

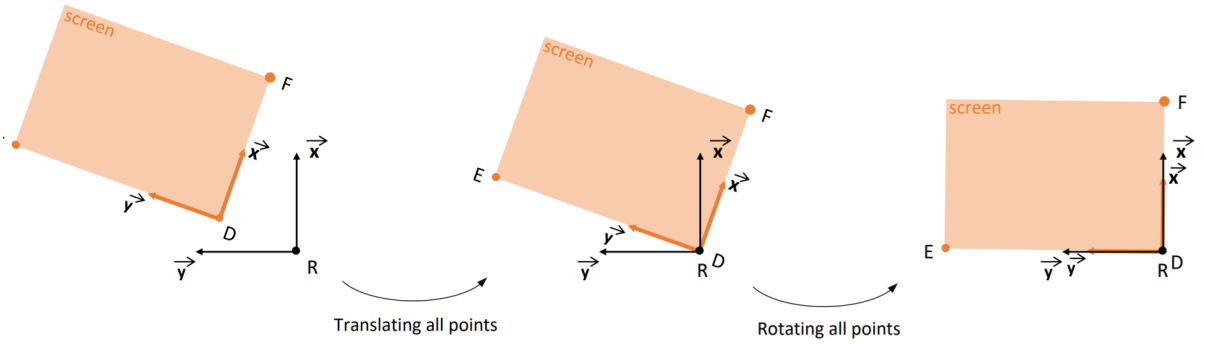

Figure 4: Screen translation and rotation

As illustrated in Figure 4, since we already previously computed the point D as the screen origin, its Reachy coordinates trivially give the translation matrix. As for the rotation matrix, we must compute the angle of rotation θ between E and the $\vec{y}$ axis, as shown in Figure 5. We have :

$$\tan(\theta) = \frac{R\_y_E}{R\_x_E} \iff \theta = \arctan(\frac{R\_y_E}{R\_x_E}) \tag{6}$$

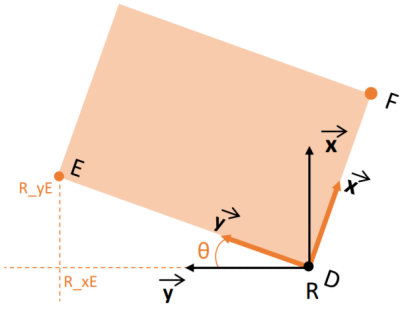

Figure 5: Angle θ of the screen's rotation

Which gives us the two matrices :

$$T = \begin{bmatrix} R\_x_D \\ R\_y_D \end{bmatrix} \qquad R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Hence, from Reachy's coordinates to the screen's coordinate system, we must compute :

$$(R_{eachy\_coord} - T) \cdot R \cdot \frac{1}{px\_size} = S_{creen\_coord} \tag{7}$$

And we compute the inverse if we wish to go from the screen's coordinates to Reachy's :

$$(S_{creen\_coord} \cdot px\_size \cdot R^{-1}) + T = R_{eachy\_coord} \tag{8}$$

This final equation (8) is the one we will apply later on when we are given a pixel screen destination point (x, y).

4 Reachy Movement

Now that Reachy knows where the screen is, and how to reach it, we can finally make it move.

4.1 Screen Interaction

When starting the process, the right arm position that Reachy is in is considered as its "rest position". This is the position that it will be returning to in between turns.

Once the calibration and screen processing is completed, Reachy will be waiting in compliant mode for an array of destination points. Upon receiving the array of points to reach, Reachy processes them with the adequate translation and rotation matrices, as described in part 3. Reachy will then become non-compliant and reach, in sequence, each of the points, in the order they were given to it. An overview of this whole process is illustrated in Figure 6.

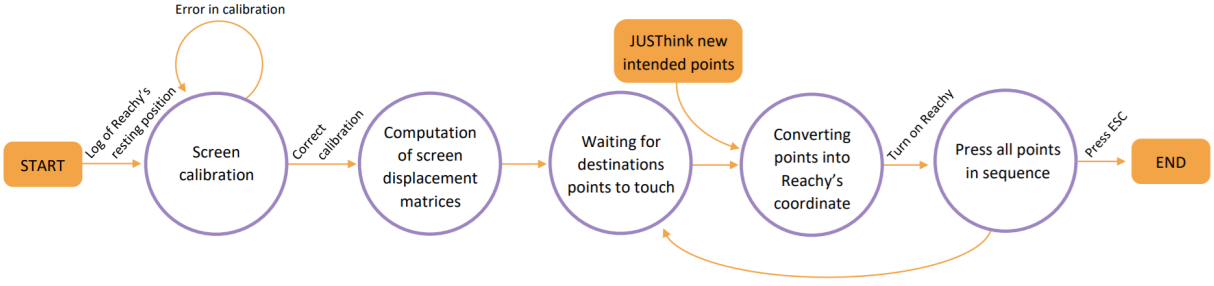

Figure 6: Screen Pressing Process

The simulation of pressing on the screen and releasing involves a three-step process, which is applied to each point : Reachy goes 20 cm above the desired point, reaches down

to the fized_z-level of the screen, which was previously determined during the calibration, and back to the position 20 cm above. This sequence allows Reachy to move freely in between the points, without risking touching the screen involuntary. The movement itself is done through Pollen Robotics' goto() function [2], which allows interpolating the movement trajectory, and using minimum jerk we can obtain a smooth movement. It also allows us to pass a 4x4 matrix containing the motors' rotation and the Cartesian destination coordinates, making it easy to control where we want Reachy to press. The matrix in question looks like the following :

$$\text{destination\_matrix} = \begin{bmatrix} 0 & 0 & -1 & R\_x \\ 0 & 1 & 0 & R\_y \\ 1 & 0 & 0 & fixed\_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Once the array of goal points has been iterated trough, and all points have been pressed, Reachy will return to its rest position, and become compliant. This will leave a place for the human to play their own turn, without having Reachy's arm disturb the playing field.

4.2 Testing

To verify the overall precision and accuracy of Reachy's touch capabilities, we conduct a set of three different tests. In all of them, we calibrate the screen, and ask Reachy to touch the same point, or set of points, 30 times. The results are as follows :

4.2.1 One point - no resting

In this first test, we make Reachy touch one single point thirty times, without going back to its resting position in between each press. The graphs in Figure 7 are the same, but with a different sized scale : on the left scaled to the screen size, and on the right a zoomed-in view, scaled to the data, to highlight position errors. The red dot is the predicted point we asked Reachy to press, here (639, 771).

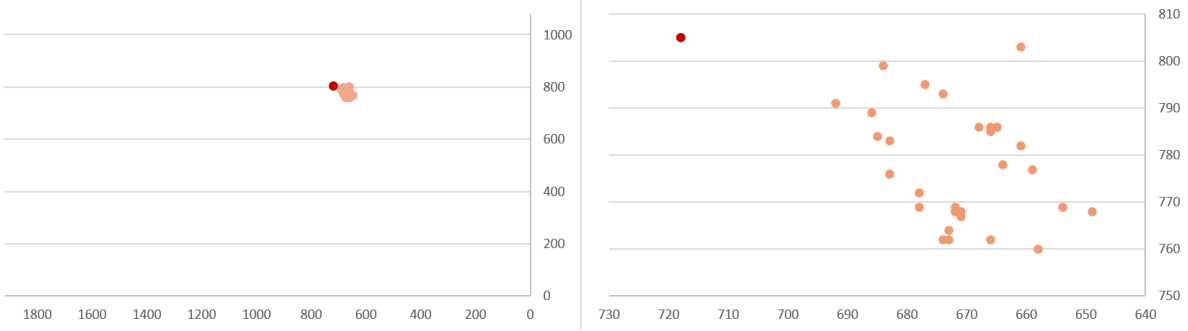

Figure 7: Single point press, without resting

Average distance :	(-46.83, -27.84)
Standard deviation :	(10.06, 12.13)
Accuracy :	94.9%
Precision :	99.1%

For this first test, we have great results for both the accuracy and the precision. The points pressed by Reachy on the screen are a bit off, by 36px on average, but they are quite consistent, with a standard deviation of only 11.

4.2.2 One point - with resting

In this second test, we want to see if the resting position has any effect on Reachy's capabilities. Therefore, we give Reachy a single point to touch thirty times, but it goes back to its resting position in between each press. The graphs in Figure 8 are the same data but with a different sized scale : on the left scaled to the screen size, and on the right a zoomed-in view, scaled to the data, to highlight position errors. The red dot is the predicted point we asked Reachy to press, here (718, 805).

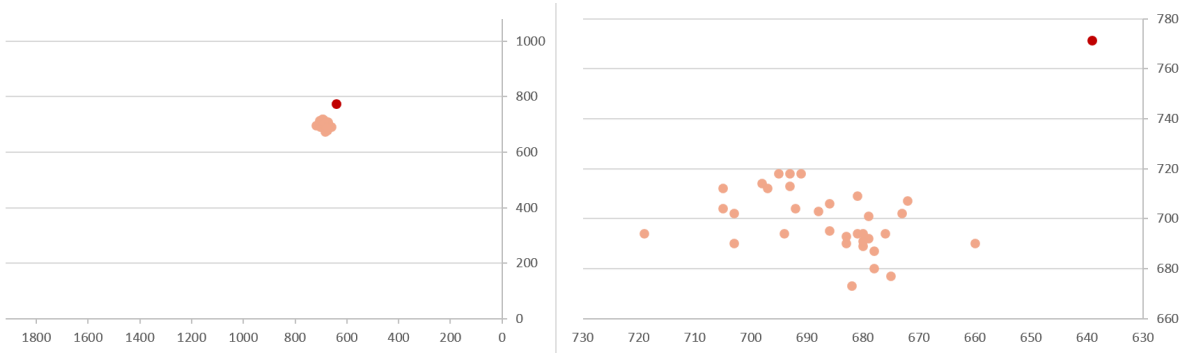

Figure 8: Single point press, with rest in between

Average distance :	(59.78, -84.09)
Standard deviation :	(12.00, 11.77)
Accuracy :	89.7%
Precision :	98.9%

For this second test, we lost quite a bit on the accuracy, 6% less than the previous test, however the precision is still great. The points pressed by Reachy on the screen are off by 70px on average, but they remain consistent, with a standard deviation of only 12.

4.2.3 Two points - without resting

So far we tested Reachy over a simple, single, point. We now add a second one, and make Reachy press the two points (1 and 2) one after the other repeatedly, thirty times each, without going back to the rest position until the very end. The graphs in Figure

9 are the same data but with a different sized scale : on the left, points both scaled to the screen size, and on the right point1 and point2 individually, scaled to the data, to highlight position errors. Point1 is in green, point2 in orange, and the red dots represent the predicted points we asked Reachy to press, respectively (727, 275) and (268, 762).

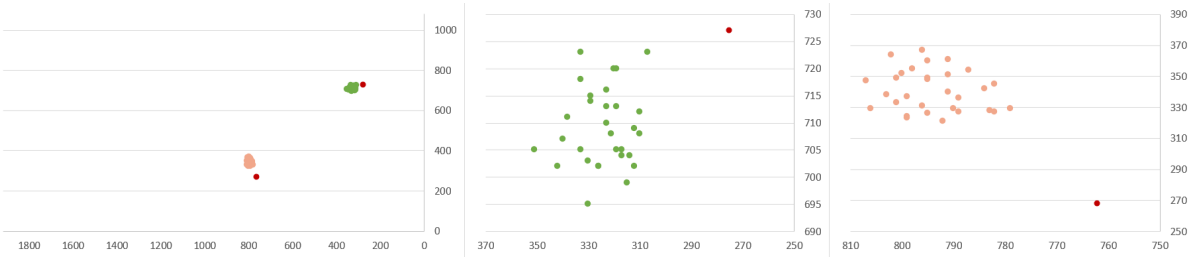

Figure 9: Two point pressed, without resting in between

	Point 1	Point 2
Average distance :	(32.80, -17.90)	(31.90, 72.73)
Standard deviation :	(13.60, 7.80)	(7.36, 13.42)
Accuracy :	93.3 %	85.7%
Precision :	99.2%	99.3%

For this last test, we have two very different results. For point 1 great accuracy and precision, with the lowest average distance so far, with 25px on average. For point 2 the precision is very similar, but the accuracy is of only 85.7%, with an average error of 60px.

Overall, we can see that the accuracy is good, yet improvable, with an average over the three tests of 90.9%. However, the precision we have from Reachy is excellent, reaching around 99% on all tests.

5 Activity Interaction

This final step of the project, and perhaps the most valuable, is to link Reachy's new-found capabilities to the game. This step requires us to create a new subscriber to JUSThink's intended_point topic. JUSThink's publisher, upon pressing the "A" key, publishes on its topic an array of two points [point_1, point_2], representing respectively the origin and the destination the computer's algorithm wants to join via a train track.

The current version of this implementation requires the human user to press the "A" key, and is a computer only game, meaning that Reachy plays the game alone. Throughout the game, whenever new points are published onto the topic, we store them in an array before passing them to Reachy. This implies that the "A" key can be pressed in the middle of an action, or twenty times in a row, all of the information is stored and will be

processed one after the other by Reachy.

We can now have Reachy connect cities, and physically help a user find an optimal solution to the train-connecting problem Switzerland seems to be facing. A video demonstration of Reachy's touch capabilities can be found in the following reference : [4].

6 Limitations & Extensions

6.1 Problems encountered

Throughout the semester, some issues arose that limit Reachy's performance.

Firstly, the current setup with a stylus pen is not ideal. The stylus usually works by transmitting the static electricity from one's hand to the screen through the aluminium body, but Reachy being a robot, there is no static electricity to transmit. Covering the extremity with aluminium foil seems to work about 70% of the time.

Another limitation is that the calibration step is quite delicate. When pressing a point on the screen, which launches the calibration process, the "simultaneous" gathering of a point's screen and Reachy coordinates is not so simultaneous. It takes 3 to 4 seconds for the forward kinematics process to actually register. This means that if the user moves Reachy's arm too fast after pressing a point on the screen, the calibration will be incorrect.

The final limitation still regards the calibration step. Due to the forward kinematics function's use of the joint position to calculate the Cartesian coordinates, the use of floating points and the motor's slight backlash [7], the precision of this function is not excellent, especially when calibrating with points that are close to each other.

These last two issues could explain the accuracy of only 90.9% registered during the testing.

6.2 Future improvements

From where we currently are, we can imagine a couple future directions for improvement.

Firstly, the improvement of the calibration process and extension of the screen interaction to non-flat surfaces would render the project more modular, making it usable in a wider range of environments (in front of a classroom for example).

A second melioration would make the most of the Otis' hand precision. At the moment, the big touchscreen requires Reachy to span over large distances in between points and is therefore not necessary. However, this inclusion would allow Reachy to play on smaller screens, or play games with more details, requiring to do fine movements.

A final improvement would be the inclusion of the left arm. Up to here, only the right arm had an Otis hand, which allowed for Reachy to hold the screen stylus. By integrating the second arm, we could imagine Reachy pressing the point with the arm closest to it, or perhaps playing more complex games requiring to press two points simultaneously.

7 Conclusion

Through this semester project, we wanted to bridge the current gap between Reachy and a physical touchscreen interaction. The implementation of several basic functionalities, which are the screen calibration, the processing of the data and commanding Reachy's movement, allow for an interaction between Reachy and a user whilst playing.

The aim also was to keep it modular and adaptable to any game. Reachy currently plays the JUSThink game, but one can easily imagine having Reachy play any other simple-enough game, such as TicTacToe, by making the screen_subscriber subscribe to a different topic containing an array of destination points to press.

The measurements gathered from testing show us that, with a correct calibration, Reachy is truly precise, and his accuracy is good, yet improvable.

8 Acknowledgments

I would like to thank my supervisors, Utku Norman and Barbara Bruno, who advised and accompanied me throughout this semester. Thank you for always answering my questions and giving advice when I was stuck on my reasoning.

I took great pleasure working on Reachy this semester. This was my first big project alone, and I have learned considerably from it. Seeing Reachy move and actually interact with the screen by the end of the semester was truly fulfilling.

References

[1] Pollen Robotics. *Arm Kinematics*. 2020. URL: https://docs.pollen-robotics.com/sdk/first-moves/kinematics/#forward-kinematics.

[2] Pollen Robotics. *Goto() Function*. 2020. URL: https://docs.pollen-robotics.com/sdk/first-moves/arm/#goto-function.

[3] Utku Norman. *A Simple Application for Monitoring Mouse and Key Events in ROS (2)*. 2021. URL: https://github.com/utku-norman/screen_app.

[4] Elisa Bianchi. *Final video demonstration of Reachy's touch capabilities*. 2022. URL: https://go.epfl.ch/ebianchi-video-demo.

[5] GeoGebra. *Geometry*. 2022. URL: https://www.geogebra.org/geometry?lang=en.

[6] Pollen Robotics. *Reachy 2021 Documentation*. 2022. URL: https://docs.pollen-robotics.com/.

[7] Wikipedia. *Backlash (engineering)*. 2022. URL: https://en.wikipedia.org/wiki/Backlash_(engineering).