

# Eden Robotics Report

Alessandrini Antoine

September 2020 - February 2022

# Contents

<b>Introduction</b>	<b>5</b>
Project . . . . .	5
Team members . . . . .	5
<b>1 Project definition</b>	<b>6</b>
1.1 Market study . . . . .	6
1.2 State of the art . . . . .	6
1.2.1 Arm . . . . .	6
1.2.2 Base . . . . .	6
1.2.3 Sensors . . . . .	6
1.2.4 Communication . . . . .	6
1.2.5 Storage . . . . .	6
1.3 Definition . . . . .	6
1.4 Specifications . . . . .	7
<b>2 Project Management</b>	<b>9</b>
2.1 Organization . . . . .	9
2.1.1 Non technical aspect . . . . .	9
2.1.2 Technical aspect . . . . .	9
2.2 Planification . . . . .	10
2.3 communication . . . . .	11
2.4 Resource management . . . . .	11
<b>3 Design</b>	<b>13</b>
<b>4 Robot characteristics</b>	<b>14</b>
4.1 Kinematics . . . . .	14
4.1.1 Base frame . . . . .	15
4.1.2 End effector frame . . . . .	16
4.2 Workspace . . . . .	16
4.3 Payload . . . . .	19
4.4 Precision . . . . .	19
4.5 Others . . . . .	20
<b>5 Simulation</b>	<b>21</b>
5.1 URDF Format . . . . .	21
5.2 Forward kinematics . . . . .	22
5.2.1 Matlab simulation . . . . .	22
5.2.2 Python simulation . . . . .	23
5.3 Inverse kinematics . . . . .	24
5.3.1 Matlab simulation . . . . .	24
5.3.2 Python simulation . . . . .	26
5.4 Torque Study . . . . .	26
5.4.1 General principal . . . . .	26
5.4.2 Joint results . . . . .	27
5.4.3 Hardware choices . . . . .	31
<b>6 JoystickControl</b>	<b>32</b>
6.1 Principle . . . . .	32
6.2 Open loop . . . . .	35
6.3 Closed loop . . . . .	35

<b>7</b>	<b>ROS</b>	<b>37</b>
7.1	Definition . . . . .	37
7.2	Training . . . . .	37
7.3	Packages . . . . .	38
7.4	Topics . . . . .	38
7.5	Nodes . . . . .	39
7.5.1	Forward kinematics . . . . .	39
7.5.2	Inverse kinematics . . . . .	39
7.5.3	Joystick . . . . .	40
7.5.4	Change frame . . . . .	40
7.5.5	Motors . . . . .	41
7.5.6	Camera . . . . .	41
7.6	Launch files . . . . .	41
7.6.1	Rviz . . . . .	42
7.6.2	Inverse kinematics . . . . .	42
7.6.3	Open loop . . . . .	42
7.6.4	Closed loop . . . . .	42
	<b>Conclusion</b>	<b>44</b>
<b>A</b>	<b>Forward kinematics python node</b>	<b>45</b>

## List of Figures

1.1	Octopus diagram . . . . .	7
2.1	Non Technial organization . . . . .	9
2.2	Technial organization . . . . .	10
2.3	WBS exemple . . . . .	10
2.4	Gantt exemple . . . . .	11
2.5	To do list exemple . . . . .	11
2.6	Financial budget extract . . . . .	12
2.7	Material budget extract . . . . .	12
2.8	Human budget extract . . . . .	12
4.1	Kinematics schema . . . . .	15
4.2	Workspace in xy plan for $z=0.07m$ . . . . .	17
4.3	Workspace in xz plan for $y=0m$ . . . . .	17
4.4	Workspace in yz plan for $x=-0.06m$ . . . . .	18
4.5	Payload mesure . . . . .	19
4.6	Precision . . . . .	20
5.1	URDF file extract . . . . .	22
5.2	Simulink model . . . . .	23
5.3	Matlab robot visualization . . . . .	23
5.4	Inverse kinematics simulink . . . . .	25
5.5	Joint block sensing . . . . .	26
5.6	Pelvis trajectory . . . . .	27
5.7	Pelvis measures . . . . .	28
5.8	Shoulder trajectory . . . . .	29
5.9	Shoulder measures . . . . .	29
5.10	Elbow trajectory . . . . .	30
5.11	Elbow measures . . . . .	31
6.1	Camera position . . . . .	32
6.2	Camera view . . . . .	33
6.3	Xbox Controller . . . . .	33
6.4	Open loop for joystick control . . . . .	35
6.5	Closed loop for joystick control . . . . .	36
7.1	Inverse kinematics node . . . . .	39
7.2	Rviz launch file . . . . .	42
7.3	Inverse kinematics launch file . . . . .	42
7.4	Open loop launch file . . . . .	42
7.5	Closed loop launch file . . . . .	43

# **Introduction**

## **Project**

This project was realized as part of our studies in connection with the company "ALL one Robotics". The initial objective was to build a robot to pick apples. Our client had noticed a great shortage of manpower in French and American farms. In France alone, 1 million seasonal workers are needed for the harvest. This labor force remains difficult to find and the crisis of the cider has amplified this phenomenon. Thus, in the United States only 80% of the apples would be picked, resulting in significant economic and food losses. This problem has given the idea to our customer to remotely operate the harvest, giving birth to the project "All One Robotics".

It is a project of 18 months of 12 students of the Ecole Centrale de Lille whose objective is to build a robot capable of picking apples and control it remotely. With the means available and the difficulties encountered, the project turned to the picking of tomatoes. We had to build a prototype and this would facilitate and accelerate the work.

Our Client, Patrick Kedziora is a French-American entrepreneur and founder of the start-up "All One Robotics". We could also count on coaches from Centrale Lille who helped us throughout the project, especially for the management of such a project: Mr. Denis le Picart and Mr. Roland Marcoin.

## **Team members**

In this project we were 12 students in the first year of the Ecole Centrale de Lille : Anna Berger, Anna Ducros, Antoine Alessandrini, Aya Skhoun, David Kirov, Héloïse Boyer-Vidal, Maxime Baquet, Noé Luxembourger, Simon Dahy, Simon Kurney, Thomas Jaouën et Victor Guinebertière. 11 of us came from preparatory classes (from all sections) and Simon K. was in double degree with his university in Germany.

# 1 Project definition

## 1.1 Market study

At the request of Mr. Kedziora, we began by conducting a market study. Even if he had already done it, it allowed us to better understand the expectations of our customer, the possible difficulties but especially to have the point of future users of our product.

Mr. Kedziora's will being to market the robot in the United States, we focused on this region even if we also studied the situation in France. For that we contacted farmers by asking them questions about :

- The size of their farms
- The method of collection and the duration
- The points of attention to pick apples
- The type of soil
- The difficulties encountered in finding personnel
- The interest for the use of robot : motivation and fear.
- The possible price.

It turned out that only the big farms (> 50 acres) were interested. Indeed, out of the twenty or so farms that we contacted, more than 80% were having difficulty recruiting, especially with the Covid crisis. This is not the case for small farms because they work mainly with local people and need fewer people. Thus, all the large farms were in favor of using robots. However, they warned us about the current existence of autonomous robots and the need to distinguish themselves. According to them, the big disadvantages of robotization today are: the price of access and maintenance. A remote-controlled robot can answer their request by reducing the cost. However, our robot must keep the advantages of autonomous robots: work all day, less personnel present. The continuous work is a primordial point because the robots are generally slower than the humans but can thus a better output on a complete day. Finally, technical characteristics were also put forward: Not too much pressure on the fruit, ability to drive on potentially muddy dirt roads, dimensions to respect.

We also went directly on the spot to visit orchards in order to have a better vision of all these constraints and to be able to better discuss with the farmers. It was thus of a great help to write the specifications.

## 1.2 State of the art

### 1.2.1 Arm

### 1.2.2 Base

### 1.2.3 Sensors

### 1.2.4 Communication

### 1.2.5 Storage

## 1.3 Definition

After conducting a market study and a state of the art, we were able to redefine the project. Initially, our client wanted us to design and manufacture a complete robot: a base and an arm. His main constraint was the flexibility of the robot to be able to adapt to other tasks and thus be used throughout the year. As we have seen in the state of the art, many bases already exist and are relatively cheap. Moreover, Mr. Kedziora also wanted us to start from a blank page in

order not to be influenced by the existing, so it would have taken too much time to focus on everything. So, with his agreement, we decided to focus on the arm, the hand and the control system.

Indeed, the main point of this project was to recover the fruits without damaging them. The hand was therefore the central element to distinguish us. We also kept the creation of the arm because the robot had to have a low cost and we wanted to imagine the least expensive design. For the same reason, we also kept the implementation of the control system.

Finally, during the course of the project and facing the difficulties we encountered, we also evolved the project during the year. The apple harvest became the cherry tomato harvest. This allowed us to reduce the dimensions of our robot while keeping almost all the constraints to respect. We were therefore able to make all the prototypes with the "traditional" tools of the plant. We were also able to meet the flexibility criteria required by our customer.

## 1.4 Specifications

Following this, we made a specification. These studies as well as the evolution of the project allowed us to estimate the expectations and constraints.

The general cases of use that we could list are the following :

- Picking up
- Storage of fruits
- Storage of robots
- Cleaning of the robot
- Recycling (not treated in the following)

We then made an "octopus diagram" which illustrates the links between the different functions:

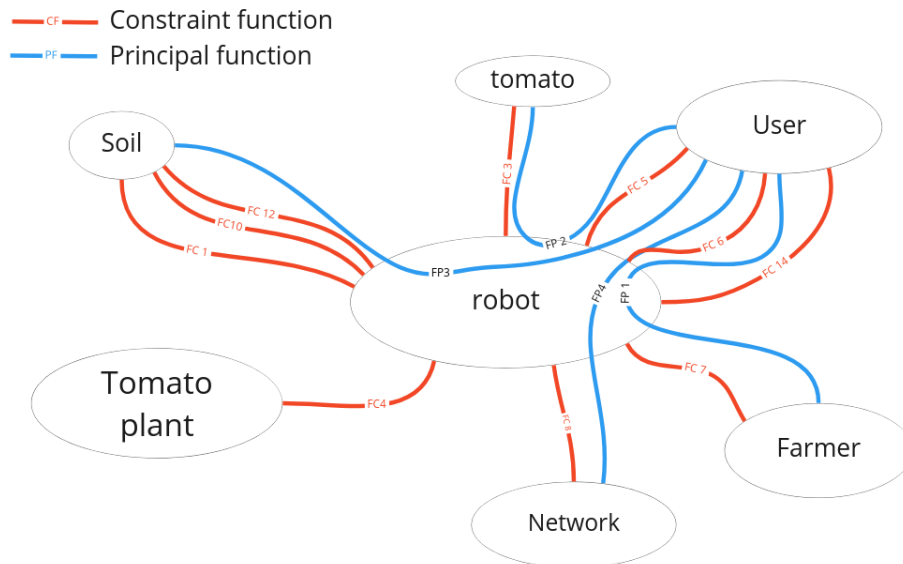


Figure 1.1: Octopus diagram

Only the main functions are presented here, the entire specification can be retained in appendix A. The main function is to be as efficient as a human over a day. Thus the arm will have to retrieve a tomato in less than 10s.

<b>Principal Function</b>	<b>Appreciation criteria</b>	<b>Level</b>	<b>flexibility</b>
PF 1: Be at least as efficient as a human	harvest duration	less than 10s	F2
PF 2: Picking tomatoes	The tomatoes are detached	80% of standard orchards are collected	F1
PF 3: Controlling the robot	Pick tomatoes	get 80% of tomatoes	F2
PF 4 Control the robot remotely	User can be in other place	500m distance	F2

Table 1: Principal functions extract

The main constraint function is to reach all the tomatoes. To measure this, we want our robot to be able to reach 80% of the tomatoes in a standard orchard.

<b>Constraint Function</b>	<b>Appreciation criteria</b>	<b>Level</b>	<b>flexibility</b>
CF 1: Keeping tomatoes intact	Forces exerted	less than 5N	F1
CF 2: Avoid branches and tomatoes	Precision	relative gap;1cm	F1
CF 3: Be easy to pilot	User feeling	Learn time;2h	F2

Table 2: Constraint functions extract



## 2 Project Management

### 2.1 Organization

#### 2.1.1 Non technical aspect

To manage the project in the best possible way, we defined from the beginning the non-technical roles to be taken in charge. The objective was to avoid the "tunnel effect" on these tasks which can be considered as secondary. They are however essential for the success of such a project.

We appointed a project manager in charge of transmitting information to all members and especially to the different technical poles. His mission was to ensure a follow-up of the progress as well as the update of the communication tools.

In order to help him, we also put in place a person in charge of the external communication. He communicated with the coaches as well as with the external stakeholders. Like the project manager, he was also in permanent communication with the client. In order to manage the budget as well as possible, financial provided for the client but also the possible trainings. Budget and training managers were appointed.








Charges		Person in charge
Transmission of information between the poles		Project manager
Project management tools	  	Antoine, Maxime, Anna B, David
Budget		Héloïse
external coordination		Victor
Training management		Anna B
Write Meeting report		Everybody

Figure 2.1: Non Technial organization

#### 2.1.2 Technical aspect

After having defined the project we were also able to define a technical organization. Three poles were created.

First a mechanical pole whose role was to design the different versions of the robot. Then to build it. An automatic pole in charge of the simulation of the robot and the control system. Finally a IT pole They worked on the user interface, the video acquisition and the remote connection.

Each pole had a person in charge to facilitate the follow-up. These regularly discussed with the project leader.

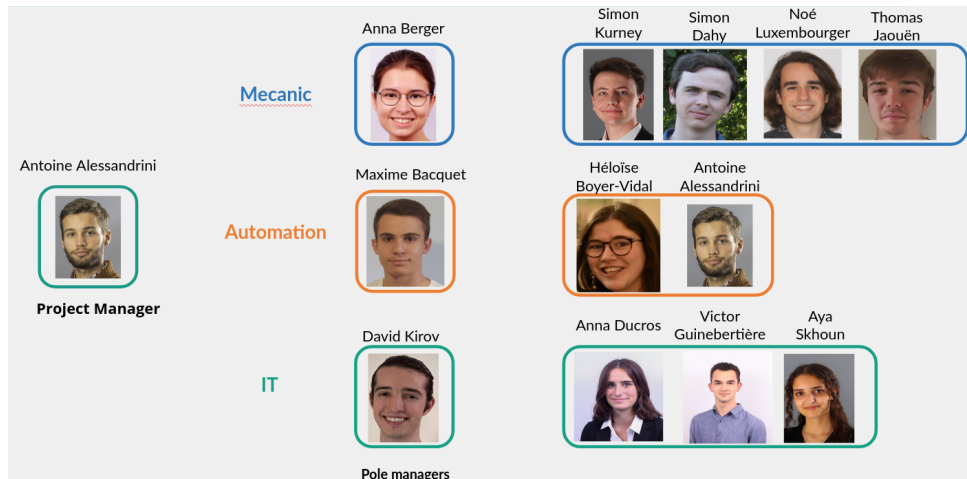


Figure 2.2: Technial organization

## 2.2 Planification

In order to manage the progress of the project, project management tools were put in place. They were updated at least weekly.

First of all a WBS (Work breakdown structure) was created. Its purpose is to break down the project into smaller pieces. It allows us to identify more precisely all the work to be done and the distribution between the different poles.

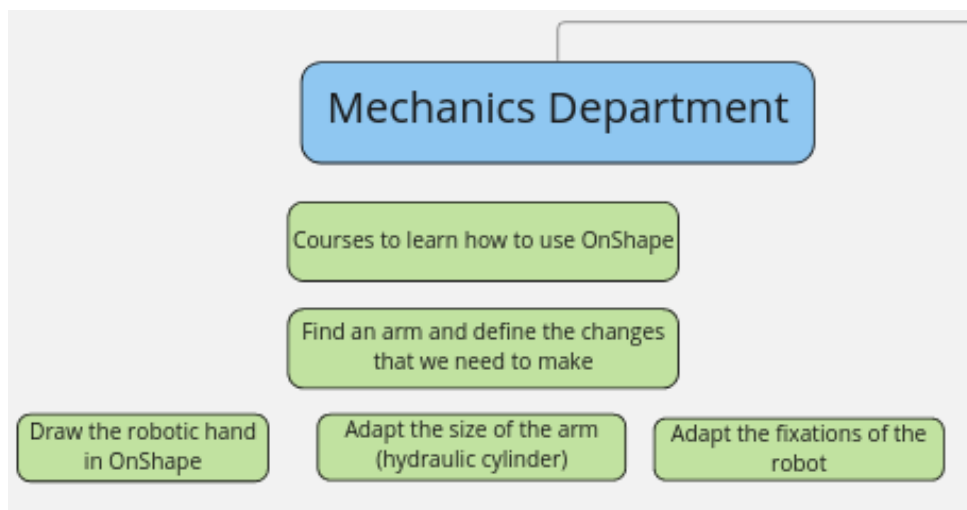


Figure 2.3: WBS exemple

Thanks to this document, we set up a Gantt chart. It illustrates the project schedule by indicating the tasks to be accomplished and the time. Each pole had its Gantt chart, which were linked to a global chart. This allowed us to anticipate certain delays and adapt. Milestones were also defined in order to have objectives all along the project. Their date were set on the audits during the project.

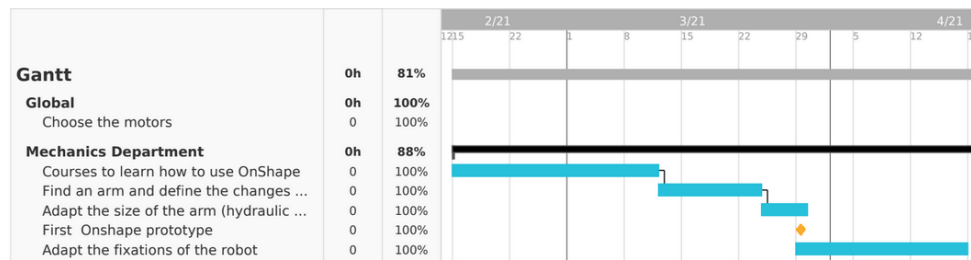


Figure 2.4: Gantt exemple

Finally, to manage the daily tasks and ensure both a good distribution of work between members and their achievement we used a to-do-list. It indicated the tasks to be carried out, the person in charge, the person in charge who followed the progress, the date and the status.

Automation Pole (Responsible : Maxime)		who ?	Responsible ?	when ?	State ?	Remarks
Members : Antoine, Maxime, Noé puis Heloise						
Training in the fundamentals of matlab/simulink	Antoine	▼			18/02/21	File
	Maxime	▼	Maxime	▼		
	Noé	▼				
Theoretical kinematic study of the arm	Antoine	▼			10/05/21	File
	Maxime	▼	Maxime	▼		Delay because study too complex manually, but after consulting with Mr. Kruszewski Matlab could allow to do this
	Noé	▼				
Dynamic study of the hand (torque equation)	Antoine	▼			15/05/21	File
	Maxime	▼	Maxime	▼		
	Noé	▼				
Kinematics study with matlab + workspace	Antoine	▼			10/05/21	File

Figure 2.5: To do list exemple

## 2.3 communication

Communication was a key point for success. We were fortunate to have a client who was very involved in the project. Thus, we had weekly meetings with him to review the progress. It was also an opportunity to take advantage of his expertise on the management of such a project as well as his ideas. During these meetings we discussed both the technical and management aspects. He also allowed us to meet people such as the CTO of BMW who gave us precious advice. At the request of Mr. Kedziora, we did not have a person in charge of taking notes. A schedule was defined in order to have a rotation between all the members and all benefit from this experience. A template for the uniformity was set up.

Each pole also met one afternoon per week to work and exchange on the technical part of the project, and a monthly meeting was held between the pole managers and the project manager to ensure the smooth running of the project.

To exchange information, we had a Google Drive to store all the documents. Github folders were also created for the IT part. Finally, we also communicated with our client via Whatsapp. We also had a messenger group with all members.

## 2.4 Resource management

Two types of means can be taken into account. First, the direct financial expenses. These are all the purchases we made. As explained earlier, Heloise was responsible for this tracking. She could therefore easily ensure that the expenses were in line with the client's wishes. These expenses also included the use and purchase of materials from central offices.

Many non-direct expenses were also taken into account. First the cost of our labor summarized in the table below. But also the consulting hours. Indeed, Centrale offered the possibility to contact professors and to benefit from their knowledge. So we contacted different people who redirected us to the teachers who could help us the most. To ensure full and balanced use as needed, Anna kept a document describing their use.

dernière mise à jour :	Type d'achat	Lieu d'achat	Pole	Cout (€)
08/12/2020	Inscription Conférence Fira	Site Fira	robotique	39€
13/03/2021	AZ delivery cablede remplacement	Amazon	informatique	4,99€
	Inno Maker Rasberry Pi	Amazon	informatique	13,99€
	LEICKE 72 Alimentation	Amazon	informatique	19,99

Figure 2.6: Financial budget extract

Matériel école	nom	prix	quantité	Temp d'impression
impression main et doigts	ASA PRO	16,28	50	5h50
impression main V2	ABS	18,8	110	18h
	ASA PRO	28,97	161	

Figure 2.7: Material budget extract

These hours were a great help. We were able to count on our coaches to help us manage this project. We were able to count on consultants, including Mr. Kruszewski, who were really involved in the project. Thanks to him, we were able to benefit from an almost weekly follow-up to perpetually advance the project. It was also a question of privileged moments to deepen certain subjects with them.

Pole	Number of days per person	Detail	Daily rate	Total
Mecanic	81	5 people available at 75% 1 day a week for 18 months	36,23 €	14 671,13 €
IT	81	4 people available at 75% 1 day a week for 18 months	36,23 €	11 736,90 €
Automation	81	3 people available at 75% 1 day a week for 18 months	36,23 €	8 802,68 €
			Total	35 211 €

Figure 2.8: Human budget extract

### 3 Design

## 4 Robot characteristics

In parallel to the design, we simulated the operation of the robot using Matlab, Simulink and python. The work explained from now on is done on the last prototype presented in the previous section. However, the principle applied has been the same throughout the project. The simultaneous work was important in order to anticipate the delays due to the manufacturing of the robot. The arm has 4 links as we can see in figure 4.1. From the bottom to the top, we will refer to these by their numbers or by a name we have assigned: pelvis, shoulder, elbow, wrist.

### 4.1 Kinematics

**Definition :** Given a vector  $x = [x_1 x_2 x_3]^T \in \mathbb{R}^3$ , we define :

$$[x] = \begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix}$$

**Definition :** Given two vectors  $x$  and  $y$  in  $\mathbb{R}^3$ , we define :  $x \times y = [x] \cdot y$

**Definition :** For a joint, we define the pitch  $h = \frac{v}{w}$  with  $v$  : the linear speed and  $w$  the angular speed

**Definition :** The screw is a  $6 \times 1$  vector that represent the angular velocity when  $\dot{\theta} = 1$  and the linear velocity of the origin when  $\dot{\theta} = 1$ .  $S = \begin{bmatrix} s_w \\ s_v \end{bmatrix}$  with  $s_v = hw - s_w \times q$  where  $h$  is the pitch and  $q$  is a point on the

**Definition :** For a given reference frame, a screw axis  $S$  is written as

$$S = \begin{bmatrix} s_w \\ s_v \end{bmatrix}$$

where either (i)  $\|s_w\| = 1$  or (ii)  $\|s_w\| = 0$  and  $\|s_v\| = 1$ . If the pitch is finite ( $h = 0$  for a pure rotation), then  $s_v = hs_w - s_w \times q$  where  $q$  is a point on the axis of the screw

The figure below shows the kinematics schema of the robot. The figure defines an  $\{s\}$  frame at the bottom, an  $\{e\}$  frame at the end effector position and a  $\{c\}$  frame at the camera position. The robot is at its home configuration. The joint are represented with the rotation (positive rotation about the axes is by the right hand rule).

The parameters can be found with Onshape and are listed below:

$L_0 = 0.069m$	$d_0 = 0m$	$h_0 = 0.06m$
$L_1 = 0.116m$	$d_1 = 0.018m$	
$L_2 = 0.16m$	$d_2 = 0.042m$	
$L_3 = 0.155m$	$d_3 = 0.01413m$	
$L_c = 0.053m$	$d_c = 0.0105m$	$h_c = 0.0815m$
$L_e = 0.2377m$	$d_e = 0.0105$	$h_e = 5.10^{-5}m$

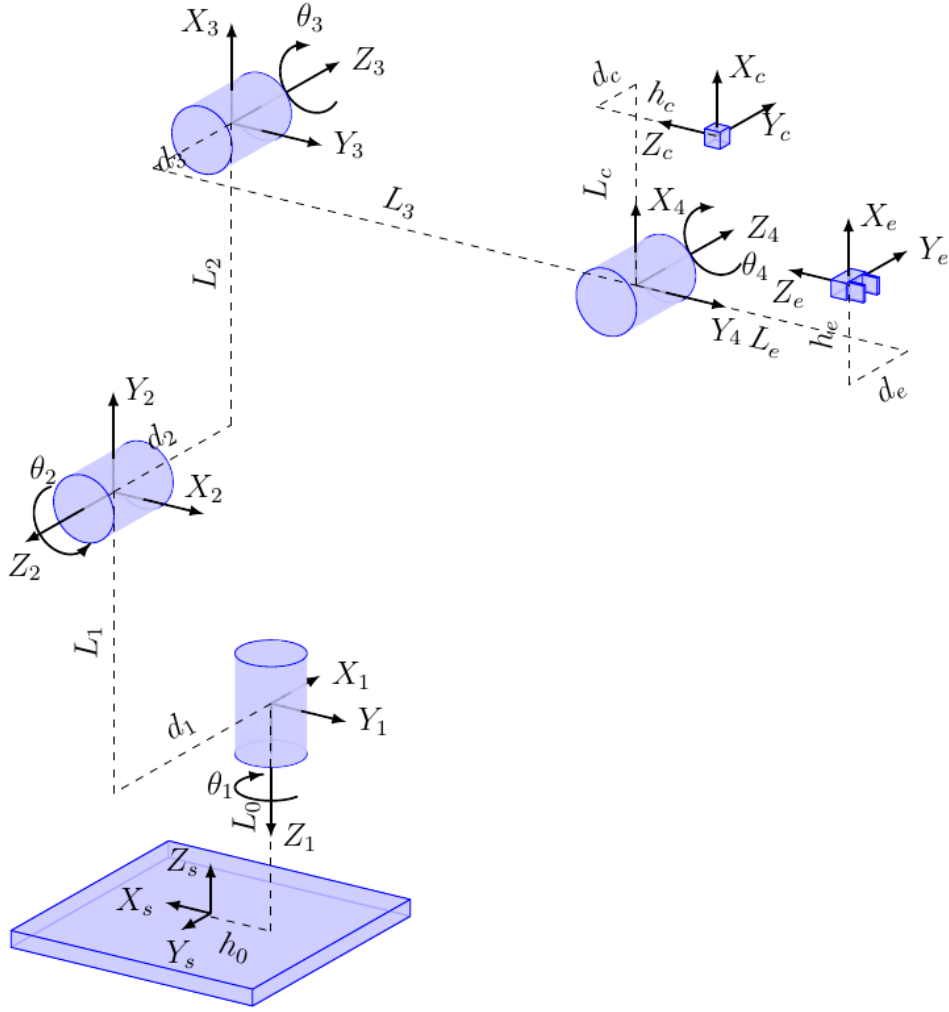


Figure 4.1: Kinematics schema

We can then define  $M_c$  and the  $M_e$  the transformation matrix ( $T_{sc}$  and  $T_{se}$ ) when the robot is at its home configuration.

$$M_c = \begin{bmatrix} 0 & 0 & 1 & -h_0 - L_3 - h_c \\ 0 & -1 & 0 & d_1 - d_2 + d_3 + d_c \\ 1 & 0 & 0 & l_0 + l_1 + l_2 + l_c \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ and } M_e = \begin{bmatrix} 0 & 0 & 1 & -h_0 - L_3 - h_c \\ 0 & -1 & 0 & d_1 - d_2 + d_3 + d_c \\ 1 & 0 & 0 & l_0 + l_1 + l_2 + l_c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

#### 4.1.1 Base frame

In this subsection we study the kinematics parameters in the base frame  $\{s\}$ . It will be the one used in the followings sections.

The rotation axis  $S_{w_i}$  of each joint in  $\{s\}$  are :

$$S_{w_1} = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}, S_{w_2} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, S_{w_3} = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix}, S_{w_4} = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix},$$

We can also write the position of each joint  $q_1, q_2, q_3, q_4, q_c, q_e$  in  $\{s\}$ . Lining up the position as columns, we get :

$$\begin{bmatrix} -h_0 & -h_0 & -h_0 & -h_0 - L_3 & -h_0 - L_3 - h_c & -h_0 - L_3 - L_e \\ 0 & d_1 & d_1 - d_2 & d_1 - d_2 + d_3 & d_1 - d_2 + d_3 + d_c & d_1 - d_2 + d_3 + d_e \\ L_0 & L_0 + L_1 & L_0 + L_1 + L_2 & L_0 + L_1 + L_2 & L_0 + L_1 + L_2 + L_c & L_0 + L_1 + L_2 + h_e \end{bmatrix}$$

There are all pure rotation joint, using the position, the rotation axis and the formula define in above we can calculate the screw axis  $S_1, S_2, S_3, S_4$  in  $\{s\}$ .. Lining up them as columns, we get :

$$S_{list} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & -1 \\ -1 & 0 & 0 & 0 \\ 0 & -L_0 - L_1 & L_0 + L_1 + L_2 & L_0 + L_1 + L_2 \\ -h_0 & 0 & 0 & 0 \\ 0 & -h_0 & h_0 & h_0 + L_3 \end{bmatrix}$$

#### 4.1.2 End effector frame

In this subsection we study the kinematics parameters in the end effector frame  $\{e\}$ . However, it is not the one that will be use later.

The rotation axis  $S_{w_i}$  of each joint in  $\{e\}$  are :

$$S_{w_1} = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}, S_{w_2} = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix}, S_{w_3} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, S_{w_4} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix},$$

We can also write the position of each joint  $q_1, q_2, q_3, q_4, q_c, q_e$  in  $\{e\}$ . Lining up the position as columns, we get :

$$\begin{bmatrix} -h_e - L_2 - L_1 & -h_e - L_2 & -h_e & -h_e & -h_e + L_c & 0 \\ d_e + d_3 - d_2 + d_1 & d_e + d_3 - d_2 & d_e + d_3 & d_e & d_e - d_c & 0 \\ L_e + L_3 & L_e + L_3 & L_e + L_3 & L_e & L_e - h_c & 0 \end{bmatrix}$$

There are all pure rotation joint, using the position, the rotation axis and the formula define in above we can calculate the screw axis  $B_1, B_2, B_3, B_4$  in  $\{e\}$ .. Lining up them as columns, we get :

$$B_{list} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 1 \\ -1 & 0 & 0 & 0 \\ 0 & L_e + L_3 & -L_e - L_3 & -L_e \\ -L_e - L_3 & 0 & 0 & 0 \\ d_e + d_3 - d_2 + d_1 & h_e + L_2 & -h_e & -h_e \end{bmatrix}$$

## 4.2 Workspace

In order to define our robot, we have determined the workspace. To do this, we used the forward kinematic computed with python as explained in section 5. Knowing the limits of each link, we made loops to evaluate the reachable positions in the whole space. We then represented the working space according to the 3 planes of the robot: top, front and side. Obviously, this does not allow to visualize the complete workspace, but it is the clearest way to show it. However, our calculations allow us to access the complete workspace, i.e. all the points reachable by the robot. We also put the python code to represent these figures. The robot is also present on the figures in red in its zero configuration to better estimate the working space.



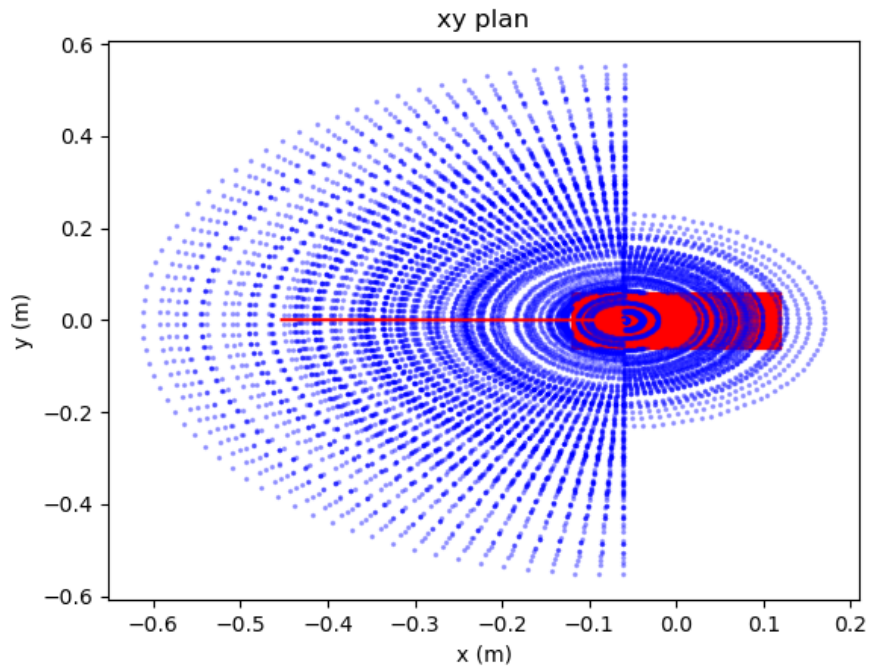


Figure 4.2: Workspace in xy plan for  $z=0.07\text{m}$

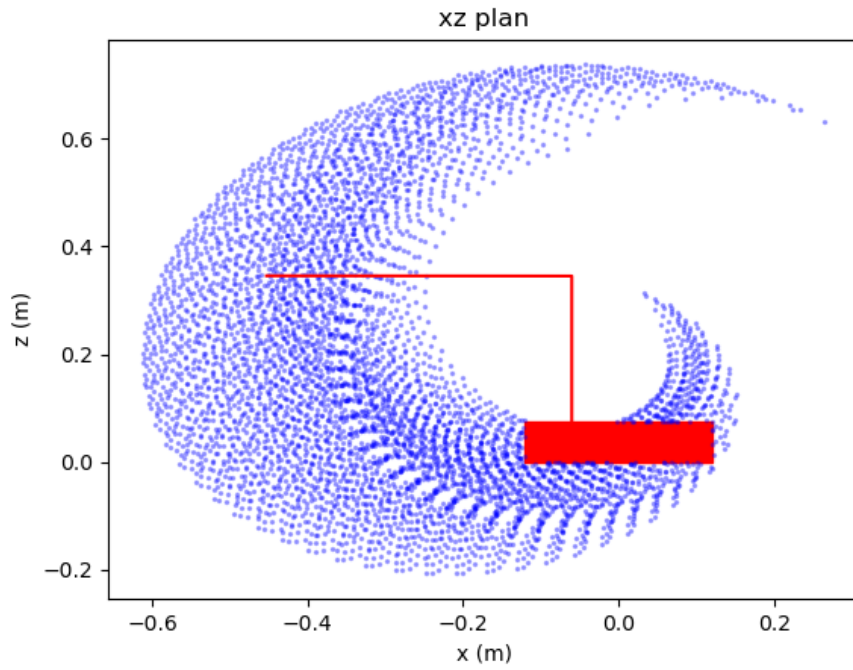


Figure 4.3: Workspace in xz plan for  $y=0\text{m}$

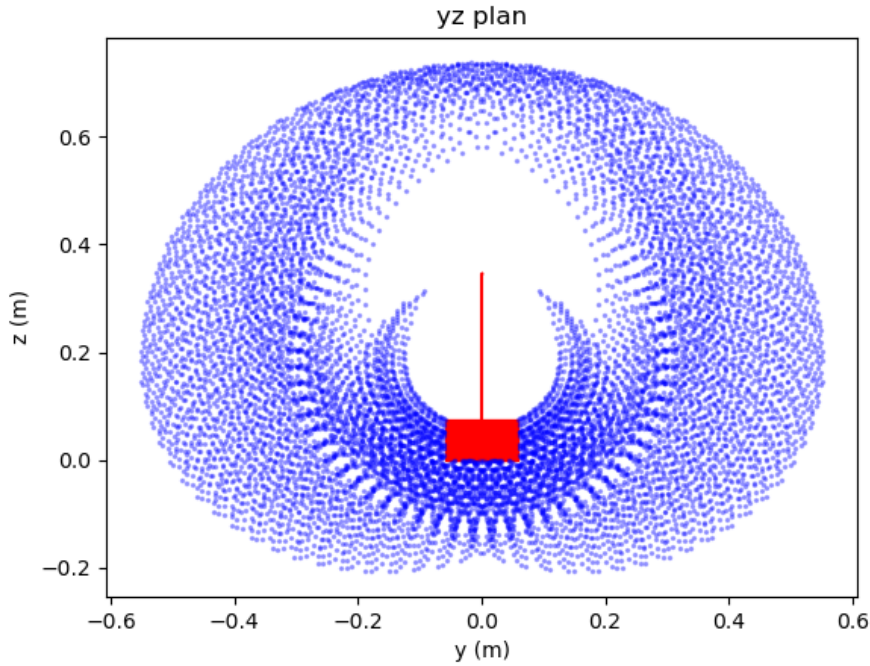


Figure 4.4: Workspace in yz plan for  $x=-0.06\text{m}$

```

1  # import kinematics parameters
2  from parameters import *
3  import matplotlib.pyplot as plt
4  # joint values in radian
5  pelvis_limits = list(np.arange(-1.57,1.57,0.2))
6  shoulder_limits = list(np.arange(-1.57,0.69,0.3))
7  elbow_limits = list(np.arange(-1.57,1.05,0.4))
8  wrist_limits = list(np.arange(0,1.57,0.4))
9  # loop on values to get all positions
10 x,y,z = [],[],[]
11 for pelvis in pelvis_limits:
12     for shoulder in shoulder_limits:
13         for elbow in elbow_limits:
14             for wrist in wrist_limits:
15                 thetalist = np.array([pelvis,shoulder,elbow,wrist])
16                 # compute transformation matrix and point
17                 t = mr.FKinSpace(m_e,screw_list,thetalist)
18                 p = t.dot(np.array([0,0,0,1]))[:-1]
19                 x.append(p[0])
20                 y.append(p[1])
21                 z.append(p[2])
22 # plot xy plan
23 plt.axes()
24 plt.plot(x,y,".b",ms = 3,alpha=0.3) # plot position
25 plt.plot([p1[0],p2[0],pe[0]], [p1[1],0,pe[1]],"-r") # plot robot arm
26 rectangle = plt.Rectangle((-0.12,-0.06), 0.24, 0.12, fc='red',ec="red")
27 plt.gca().add_patch(rectangle) # plot robot body
28 plt.title('xy plan')

```

```

29 plt.xlabel('x (m)')
30 plt.ylabel('y (m)')

```

### 4.3 Payload

Another important characteristic of our robot is the payload. This is the maximum mass that can be lifted by our robot before breaking. A cherry tomato weighs on average 15g, so we need a robot able to lift at least 2 times that to avoid any problem. To measure the payload, we placed the robot in the "worst" configuration: arm stretched flat. We then measured the intensity of the shoulder motor to lift a given mass. Then we held the robot in position by blocking it and we operated the same motor by measuring the current at the moment of the break. For a DC motor the current is proportional to the torque which is proportional to the mass, it is then possible to determine the maximum mass that the robot can support.

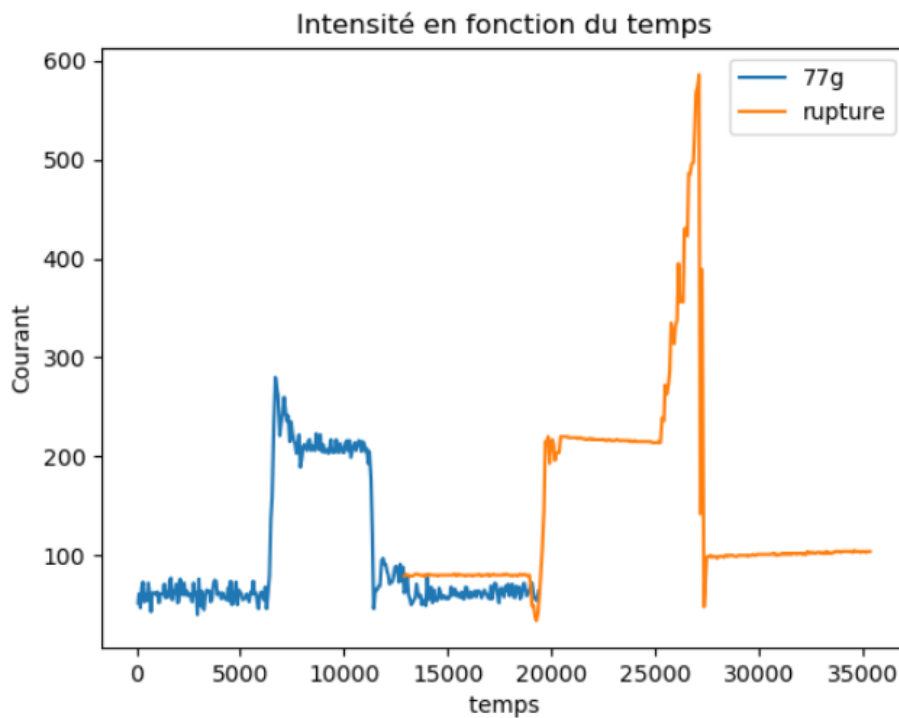


Figure 4.5: Payload mesure

The graph below shows the intensity in a specific unit of the motor. We measured an intensity of 300 for a mass of 77g and an intensity of 600 at breakage. Thus, **the payload of our robot is 144g**. We can note that this is not the limit of the robot. Indeed, it is the pvc gear that broke and not the physical limits of the robot. A stronger gear would allow to increase the maximum mass if needed.

### 4.4 Precision

Finally, we have determined the accuracy of our robot. For this, we subjected our robot to a movement of 10cm from left to right several times. We made sure to mark the trajectory of our robot and we measured the real distance covered. It was 12cm on average which makes a **accuracy of 2cm**. However, as we will see later, this is sufficient with the control method we have selected because the error is naturally compensated by the user.

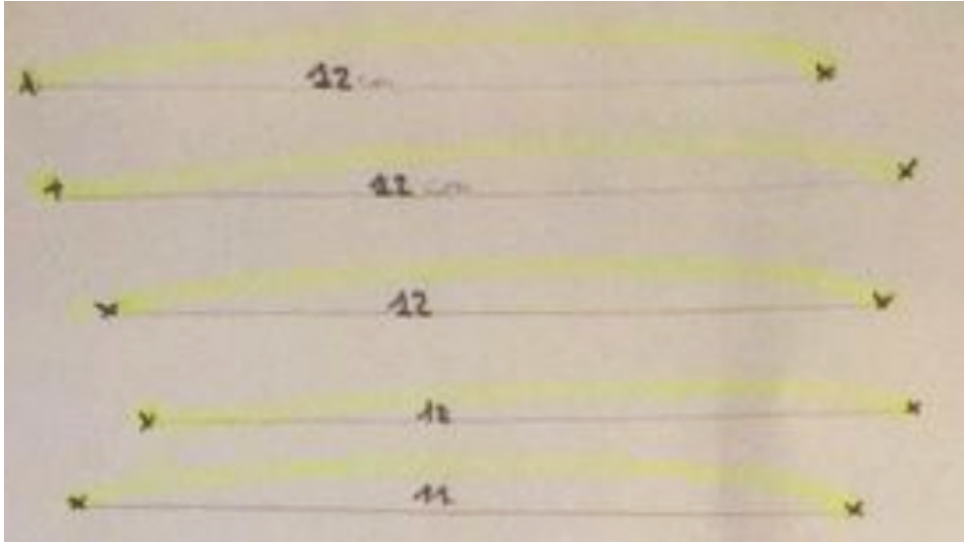


Figure 4.6: Precision

#### 4.5 Others

We also have other characteristics that we have measured:

- height : 70cm including 50cm of arms
- mass: 2.7kg of which for 1.2kg the base

## 5 Simulation

### 5.1 URDF Format

The Universal Robot Description Format (URDF) is an XML (eXtensible Markup Language) file format used by the Robot Operating System (ROS) to describe the kinematics, inertial properties, and link geometry of robots. A URDF file describes the joints and links of a robot:

- **Joints :** Joints connect two links: a parent link and a child link. A few of the possible joint types include prismatic, revolute (including joint limits), continuous (revolute without joint limits), and fixed (a virtual joint that does not permit any motion). Each joint has an origin frame that defines the position and orientation of the child link frame relative to the parent link frame when the joint variable is zero. The origin is on the joint's axis. Each joint has an axis 3-vector, a unit vector expressed in the child link's frame, in the direction of positive rotation for a revolute joint or positive translation for a prismatic joint.
- **Links :** While the joints fully describe the kinematics of a robot, the links define its mass properties. These start to be needed in Chapter 8, when we begin to study the dynamics of robots. The elements of a link include its mass; an origin frame that defines the position and orientation of a frame at the link's center of mass relative to the link's joint frame described above; and an inertia matrix, relative to the link's center of mass frame, specified by the six elements on or above the diagonal. (Since the inertia matrix is symmetric, it is only necessary to define the terms on and above the diagonal.)

This format will also be useful to build the Simulink model of the robot. Thankfully, the library **onshape-tp-robot** in python can transform Onshape design into an URDF model. It is very important that you have respected the rules explained in the last section. It will download the stl file of each part and create all the joint and the links from the main assembly. The inertia matrices and the mass are also imported for each block.

As explain on the librairy documentation, you should create onshape API key (see onshape developer portal). It is recommended to store them on your bashrc or zshrc because the secret key will no longer be shown.

```
export ONSHAPE_API=https://cad.onshape.com
export ONSHAPE_ACCESS_KEY=Your_Access_Key
export ONSHAPE_SECRET_KEY=Your_Secret_Key
```

Then, you should create a folder where you want your urdf file to be construct and write a config.json file:

```
~$ mkdir -p robot\_urdf && touch robot\_urdf/config.json
```

The config file must contain at least the following fields :

```
{
  "documentId": "document-id",
  "assemblyName": "onshape assembly",
  "outputFormat": "urdf"
}
```

The documentId is the number (below XXXXXXXXXX) you can find in Onshape URL: <https://cad.onshape.com/documents/XXXXXXXXXX/w/YYYYYYYYY/e/ZZZZZZZZ>.

Once this is done, if you properly installed and setup your API key, just run the following command. It will create an urdf file and put the stl file in the folder.

```
~$ onshape-to-robot robot_urdf
```

The file will contains only the joints define in the final assembly. This is why you need subassemblies with the fixed part. As we can see on the extract below, the camera has a fixed joint with the hand. Also, it downloads all the stl files and describes the visual position of each. However, it has only one global parameter for each subassemblies that define the inertia.

```
<link name="camera">
  <visual>
    <origin xyz="-0.0505004 -3.25261e-18 -0.083" rpy="-3.14159 8.99294e-30 -3.04281e-32" />
    <geometry>
      <mesh filename="package:///camera.stl"/>
    </geometry>
    <material name="camera material">
      <color rgba="0.282353 0.54902 0.160784 1.0"/>
    </material>
  </visual>
  <collision>
    <origin xyz="-0.0505004 -3.25261e-18 -0.083" rpy="-3.14159 8.99294e-30 -3.04281e-32" />
    <geometry>
      <mesh filename="package:///camera.stl"/>
    </geometry>
    <material name="camera material">
      <color rgba="0.282353 0.54902 0.160784 1.0"/>
    </material>
  </collision>
  <inertial>
    <origin xyz="-0.003 9.37179e-20 -0.0015" rpy="0 0 0"/>
    <mass value="0.000135" />
    <inertia ixx="5.0625e-10" ixy="-1.00385e-41" ixz="7.79417e-40" iyy="5.0625e-10" iyz="6.90342e-42" izz="8.1e-10" />
  </inertial>
</link>

<joint name="camera" type="fixed">
  <origin xyz="0.0560004 0.08 -0.0105" rpy="1.5708 3.04281e-32 8.99294e-30" />
  <parent link="hand" />
  <child link="camera" />
  <axis xyz="0 0 1"/>
  <limit effort="1" velocity="20" />
  <joint_properties friction="0.0"/>
</joint>
```

Figure 5.1: URDF file extract

## 5.2 Forward kinematics

To calculate the forward kinematics of our arm we used two different methods. This allowed us to compare the results and validate them.

### 5.2.1 Matlab simulation

To begin with, Matlab offers the possibility to import a URDF file describing a robot to make a Simulink model using the Simscape toolbox. This is the easiest method when you have the URDF file. It also has the advantage of visually simulating the robot. Indeed, the model is based on the stl files of each part. To create the model, simply run the following command in Matlab :

```
1 %% create robot model from urdf
2 smimport(<path to urdf>)
```

It will open a simulink file, once it is created, the joints must be modified so that the motor torque is calculated automatically and the desired angles can be entered manually. Finally, a position sensor linking the reference frame and the target frame must be added. We can then obtain the position of the hand for a given angle vector.

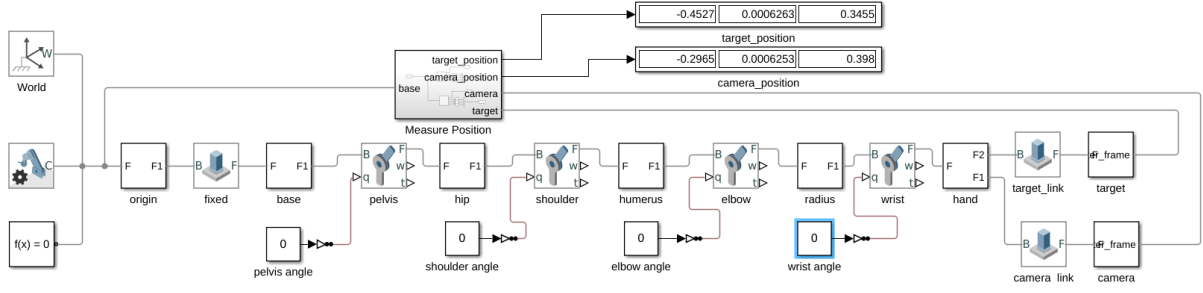


Figure 5.2: Simulink model

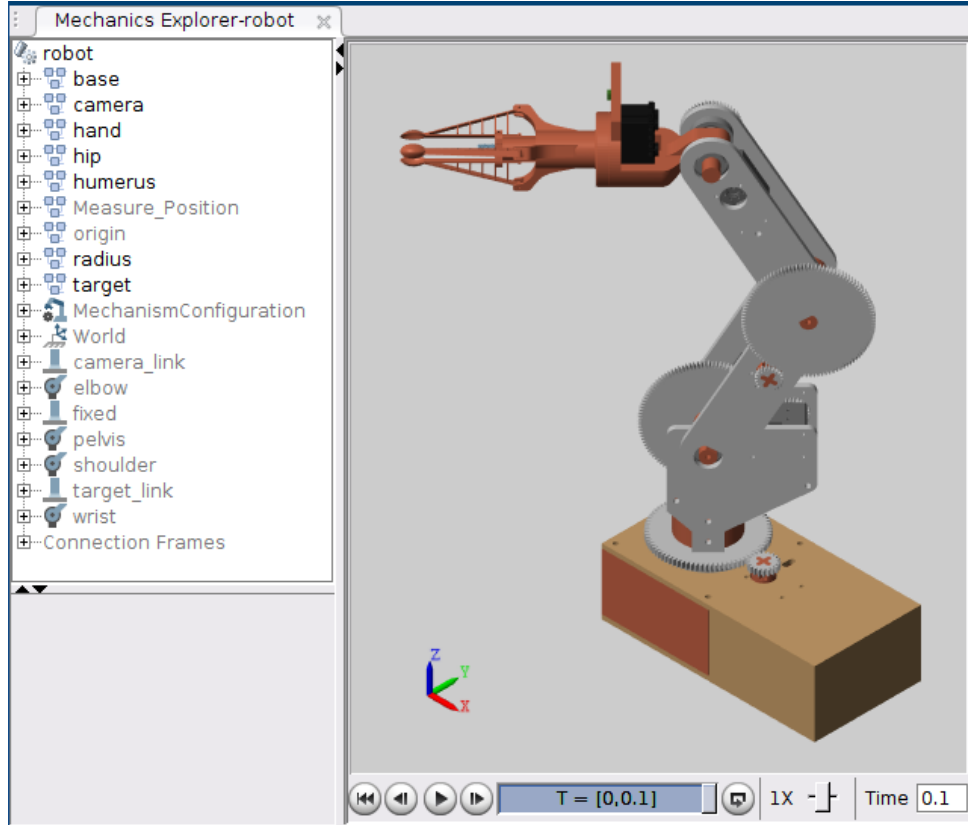


Figure 5.3: Matlab robot visualization

It should be noted that a target block positioned at the center of the fingers has been added in the Onshape model. It is in fixed link with the hand and this link is defined in the general assembly. Thus, when creating the URDF file, this target appears separately. It allows to have an easy way to locate it. the mass of this object, which must be defined is  $10^{-5}$ , to be considered as zero.. The same thing has been done at level of the camera position.

### 5.2.2 Python simulation

**Definition :** Let  $S = (w, v)$  be a screw axis. If  $\|w\| = 1$  then, for any distance  $\theta \in \mathbb{R}$  traveled along the axis,

$$e^{[S]\theta} = \begin{bmatrix} e^{[w]\theta} & (I\theta + (1 - \cos\theta)[w] + (\theta - \sin\theta)[w]^2)v \\ 0 & 1 \end{bmatrix}$$

If  $w=0$  and  $\|v\| = 1$  then

$$e^{[S]\theta} = \begin{bmatrix} I & v\theta \\ 0 & 1 \end{bmatrix}$$

The second approach is more theoretical and is based on the kinematic parameters seen previously. To realize the calculations we used the python library modern robotics. It has already created functions to calculate the direct kinematics.

This library is based on the exponentials of matrices to calculate the position of the end effector from the coordinates of each link. It uses the following formula:

$$T(\theta) = e^{[S_1]\theta_1} e^{[S_2]\theta_2} e^{[S_3]\theta_3} e^{[S_4]\theta_4} M$$

where  $\theta$  is a  $4 \times 1$  vectors of joint coordinates,  $S_i$  is the screw axes of the joint  $i$  and  $M$  is the transformation matrix when the robot is at its zero configuration.

So we can use the function FKInSpace which takes as arguments  $M, \theta$  and  $S_{list}$  as defined above. Depending on whether we want the position of the camera or the end effector, we just have to change the  $M$  matrix. This method was faster to perform the calculations and faster to test a large number of values. However, it does not allow visualization.

```

1  # import kinematics parameters
2  from parameters import *
3  import modern_robotics as mr
4  # define desired angles
5  thetalist = np.array([0,0,0,0])
6  # get transformation matrix and extract the position
7  t = mr.FKInSpace(m_e,screw_list,thetalist)
8  p = t.dot(np.array([0,0,0,1]))[:-1]
```

For the same set of angles, we then obtain the following results using Matlab and python:

joints (rad)	Matlab position (m)	Python Position (m)
[0 0 0 0]	[-0.4527 0.00063 0.3455]	[-0.4527 0.00063 0.3455]
[pi/4,-pi/3,0,pi/3]	[-0.1286 0.06948 0.07537]	[-0.1286 0.06949 -0.07533]
[pi/4,pi/6,-pi/6,pi/3]	[-0.2259 0.1668 0.4583]	[-0.2258 0.1667 0.4583]

Table 1: Matlab and Python result for forward kinematics

As we can see, the results are identical at  $10^{-4}m$ . We can therefore validate the kinematic model of our robot. The position of matlab is correct since it comes from an explicit model and allows a visualization.

### 5.3 Inverse kinematics

In the same way we used two methods to calculate the inverse kinematics. The objective is to determine the coordinates of each link from a given position.

#### 5.3.1 Matlab simulation

Once we had a simulink model, we were able to create a matlab variable that represents the robot. It is obtained with the script below. This variable contains information about each link (position, parent, child) but also about each body (mass, center of mass and inertia). The bodies are numbered from 1 to 7. The number 1 corresponds to the base and 7 to the end effector. We could identify them from the information on the mass and inertia.



```

1 %% create robot matrix
2 open_system('robot.slx')
3 S=sim('robot.slx')
4 [robot,importInfo] = importrobot(gcs)
5 robot.DataFormat = 'column';

```

The Robotic System toolbox then allows to calculate the inverse kinematics. Indeed, there is a block *inverse kinematics* which takes as input a desired configuration (transformation matrix), weights which allow to adjust the importance of reaching the desired rotation and translation according to the 3 axes and returns the list of the coordinates of each link. This block takes as parameter the robot variable created before. You must then indicate the name of the body corresponding to the end effector.

The block also offers the possibility to choose the resolution method. We have selected the Levenber-Marquardt method leaving the original resolution parameters. This method requires to provide an initial value, if possible close to the real value. As we will explain in the following parts, our robot will always start in the same position. We therefore provided this angle vector as origin.

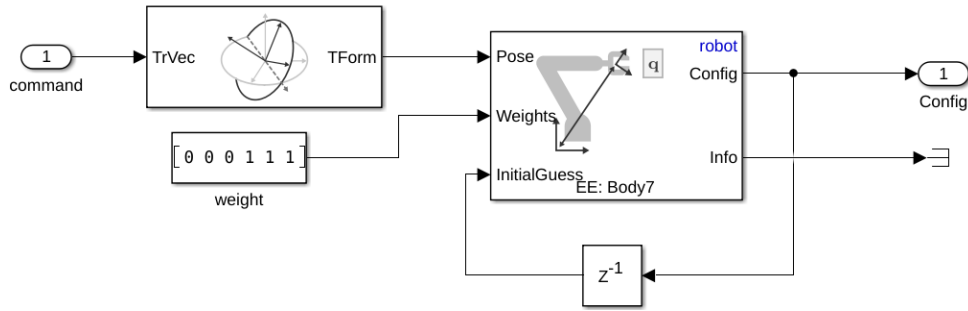


Figure 5.4: Inverse kinematics simulink

We are only interested in the position of the hand. Indeed, the hand being symmetrical the orientation of the fingers to catch objects is not important. Thus, the points for the orientation are null and the points for the position are at the maximum.

To verify the validity of the method, we retrieved the values of the link coordinates found by this block for desired end effector positions. Then, we submitted the robot in direct kinematics to this angles and verified that the positions are the same. Even if we know a set of angles corresponding to this position, depending on the initial hypothesis, the angles found can be different. Thus, comparing the value of the angles is not a good way to make sure that the resolution is working properly. As we do not take into account the orientation, we have only compared the positions.

desired position (m)	real position (m)	angles (rad)
[-0.4527 0.00063 0.3455]	[-0.4542 0.00063 0.3455]	[ $9.10^{-6}$ $9.10^{-3}$ $9.10^{-3}$ 0]
[-0.1286 0.06948 0.07537]	[-0.1288 0.06968 0.0739]	[0.7854 0.6981 0.7006 1.377]
[-0.2259 0.1668 0.4583]	[-0.2269 0.1678 0.4585]	[0.7855 0.6981 -0.1312 0.6766]

Table 2: Inverse kinematics results with Matlab

In the case of the table above, the initial assumption is always  $[0 \ 0 \ 0 \ 0]$ . The desired position and the actual position are identical at  $10^{-3}m$ . We can therefore validate the model. Nevertheless, the closer the initial value is to the final result, the smaller the error is. As we will see later, in our case, the arm will start in its zero configuration. We therefore know the value of these angles. In order to be as accurate as possible, this will be our starting point for calculating the inverse kinematics in all cases. As we can see from the table, this is sufficient to obtain a satisfactory result in all cases.

### 5.3.2 Python simulation

## 5.4 Torque Study

### 5.4.1 General principal

As soon as we were able to create a first model, we were interested in the necessary motor torque. The objective was to determine the characteristics of each linkage. Even if it was not the final robot, the different prototypes allowed us to have an estimation that we updated as soon as a choice was made.

In order to determine the necessary motors, we measured the following variables:

- torque
- rotation speed
- power

▼ Sensing
<input type="checkbox"/> Position
<input checked="" type="checkbox"/> Velocity
<input type="checkbox"/> Acceleration
<input checked="" type="checkbox"/> Actuator Torque
<input type="checkbox"/> Lower-Limit Torque
<input type="checkbox"/> Upper-Limit Torque

Figure 5.5: Joint block sensing

All these measurements were made in simulation with Matlab. In the Simscape model, we can modify the linkage blocks to return the torque and speed. The Power is measured by multiplying the two. The units are not specified and depend on the units defined. In our case :

- length : m
- time : s
- mass : kg
- angle : rad
- force : N
- power : W

We develop in the following sub-sections the results for each link. After having estimated the characteristics a margin of 30% was taken, our consultant explained to us that it was a security.

In reality only the power delivered interested us to select the engines. Indeed, if an engine is able to provide a certain power, we must look at the tuple (torque, speed) allowing this power. If one of the two glands does not meet the necessary values, a gearbox can be added.

Let's take a gear of speed  $r$ ,  $\omega' = r\omega$  and  $C' = \frac{C}{r}$  or  $P' = \omega'C' = r\omega\frac{C}{r} = \omega C = P$ . The gear allows to modify the torque and the speed without modifying the power. Therefore, a corresponding gearbox must exist.

A static study was also conducted. We placed the robot in the "worst" configuration for each link and then we measured the torque required to maintain the robot in this position. The torques indicated on the motors' data sheet do not usually guarantee a static hold. To make sure that our motors are powerful enough, we multiplied this value by 3 to choose them.

The table below lists the characteristics required for each linkage as well as the estimates obtained on the first prototype. It is important to note that this one, although farther from reality, is still close. Moreover the final values are all lower because we have reduced the mass of the robot. In order to anticipate the delivery time, we based ourselves on results obtained on intermediate prototypes. We can then validate this method.

## 5.4.2 Joint results

### 5.4.2.1 Pelvis

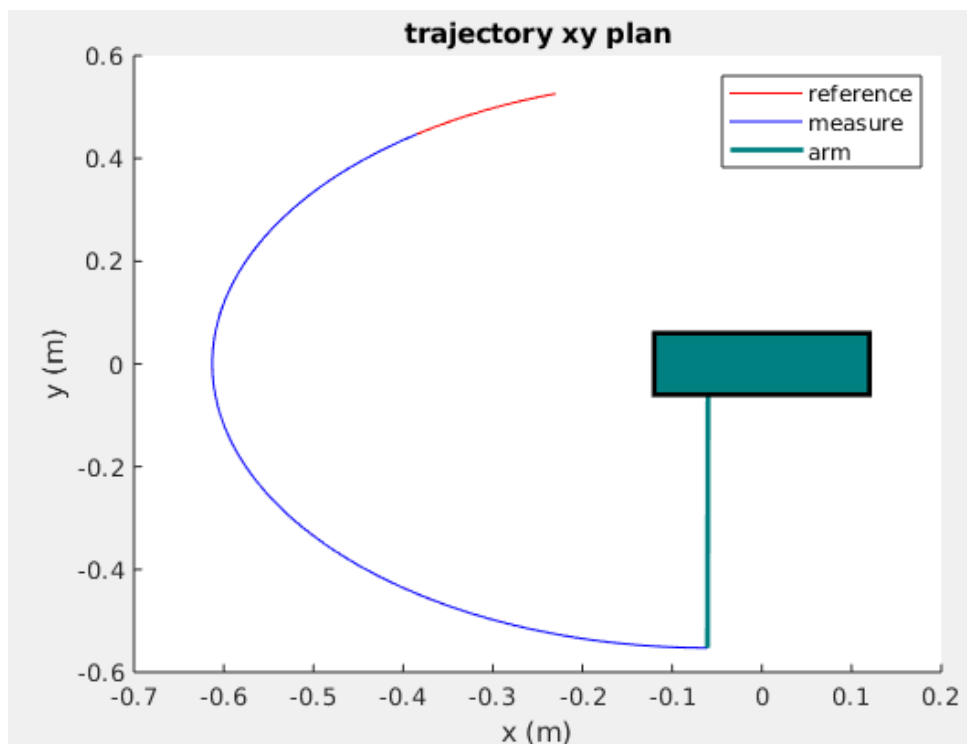


Figure 5.6: Pelvis trajectory

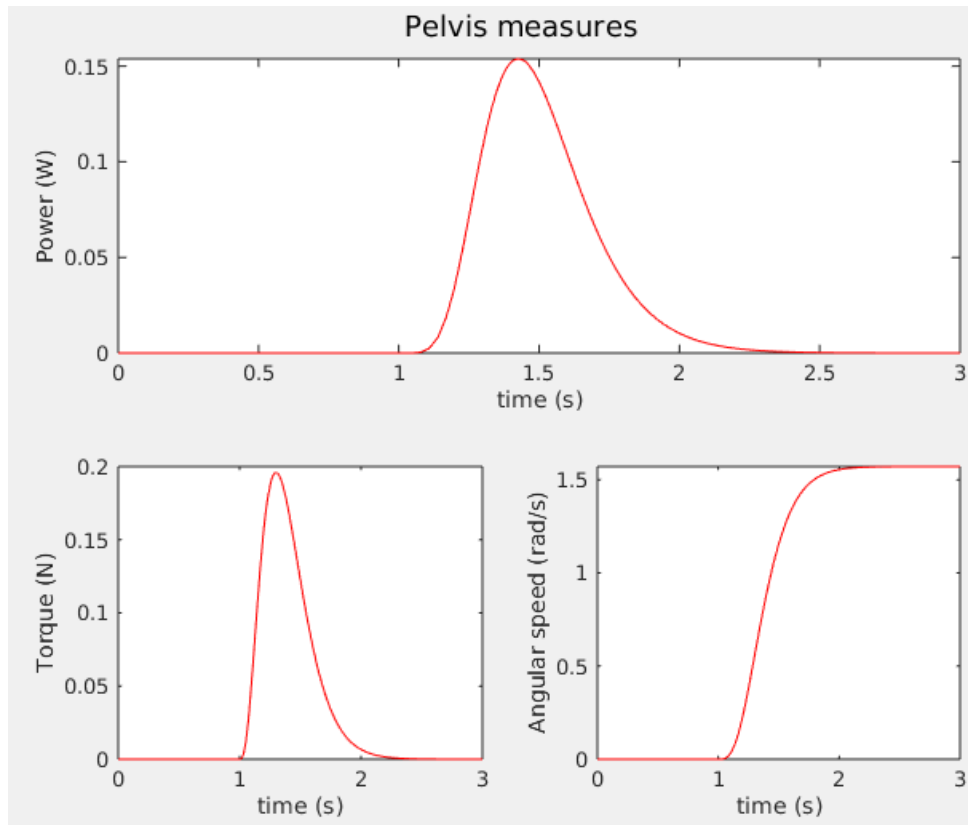


Figure 5.7: Pelvis measures

#### 5.4.2.2 Shoulder

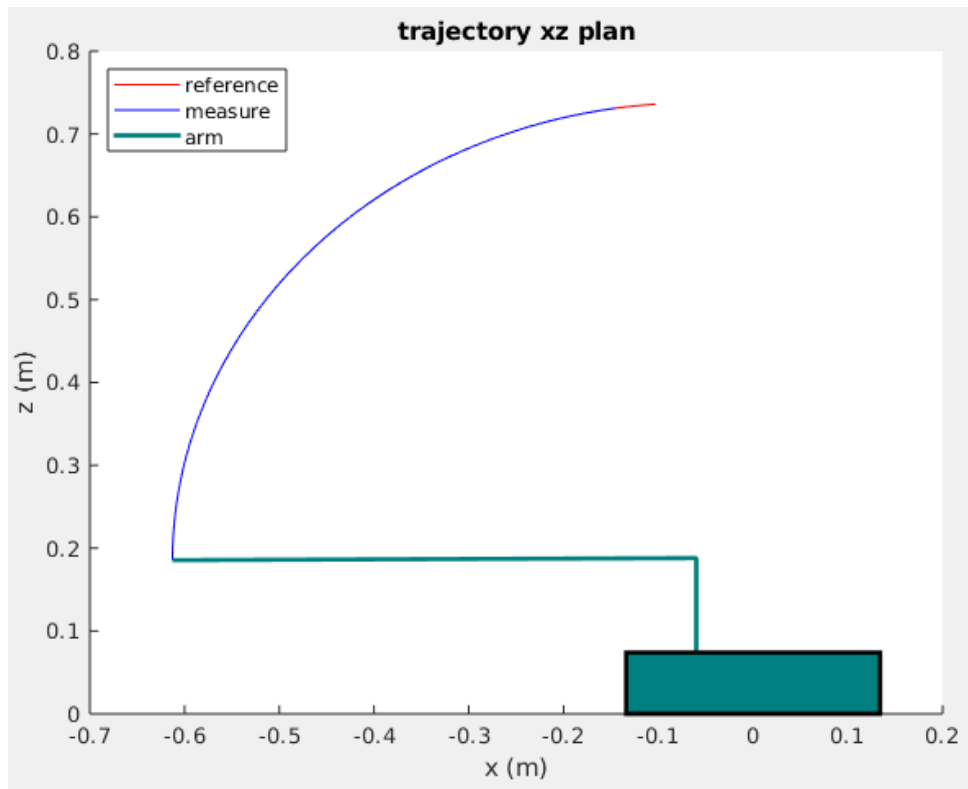


Figure 5.8: Shoulder trajectory

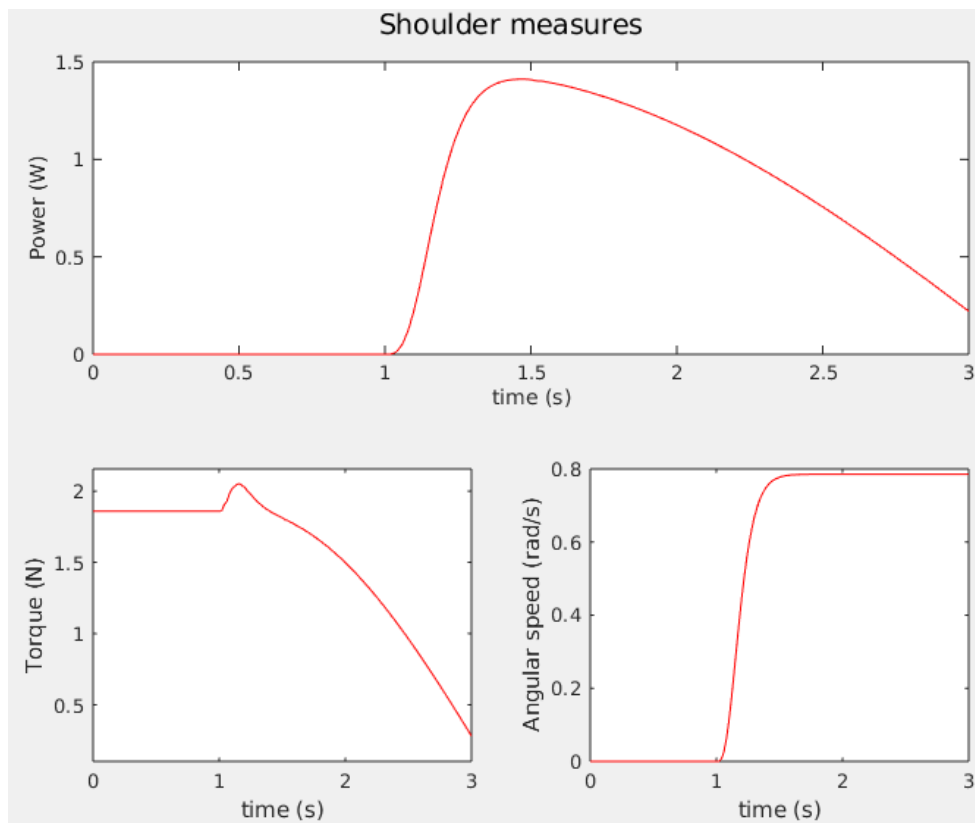


Figure 5.9: Shoulder measures

#### 5.4.2.3 Elbow

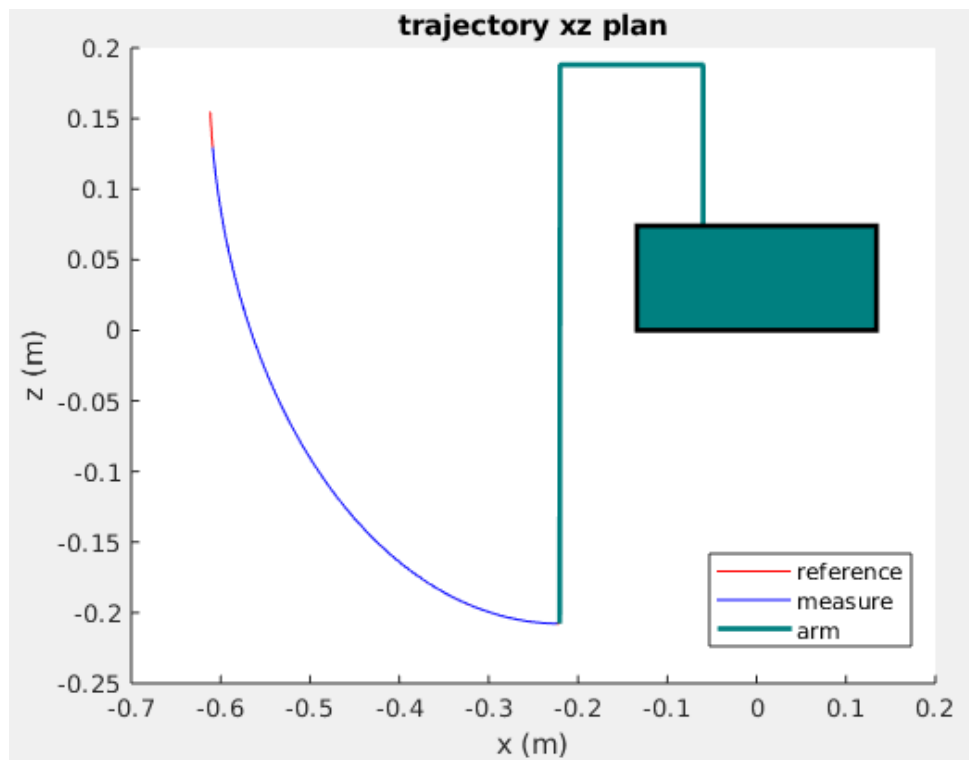


Figure 5.10: Elbow trajectory

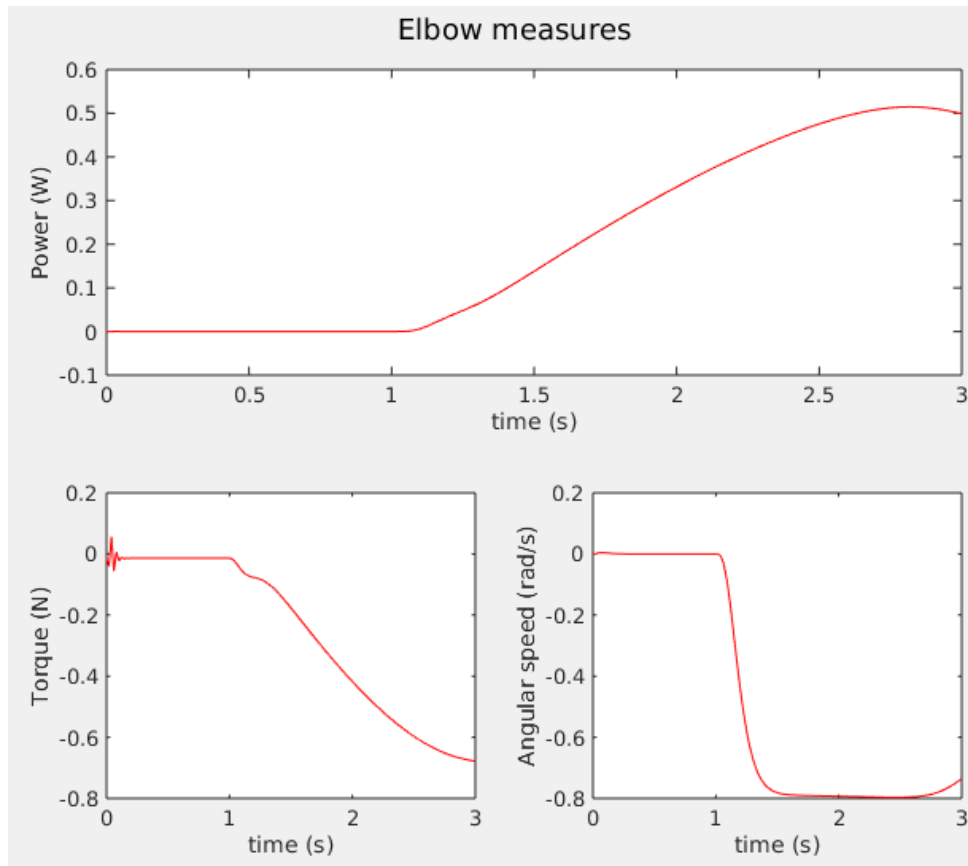


Figure 5.11: Elbow measures

#### 5.4.2.4 Wrist

### 5.4.3 Hardware choices

## 6 JoystickControl

### 6.1 Principle

The initial request of our customer was a remote controlled arm. This was the easiest and fastest solution to implement. Mr. Kedziora also wanted to avoid using artificial intelligence in order not to lengthen the prototype manufacturing time. Despite our fears about the difficulties that this type of order can pose, he really insisted that we use this method, which is the first one that we have put in place.

In this configuration, a camera, fixed on the arm at the level of the hand, allows the user to have a video return. An Xbox controller in our case is used to control the arm. The commands sent by the user are done in the camera's frame.

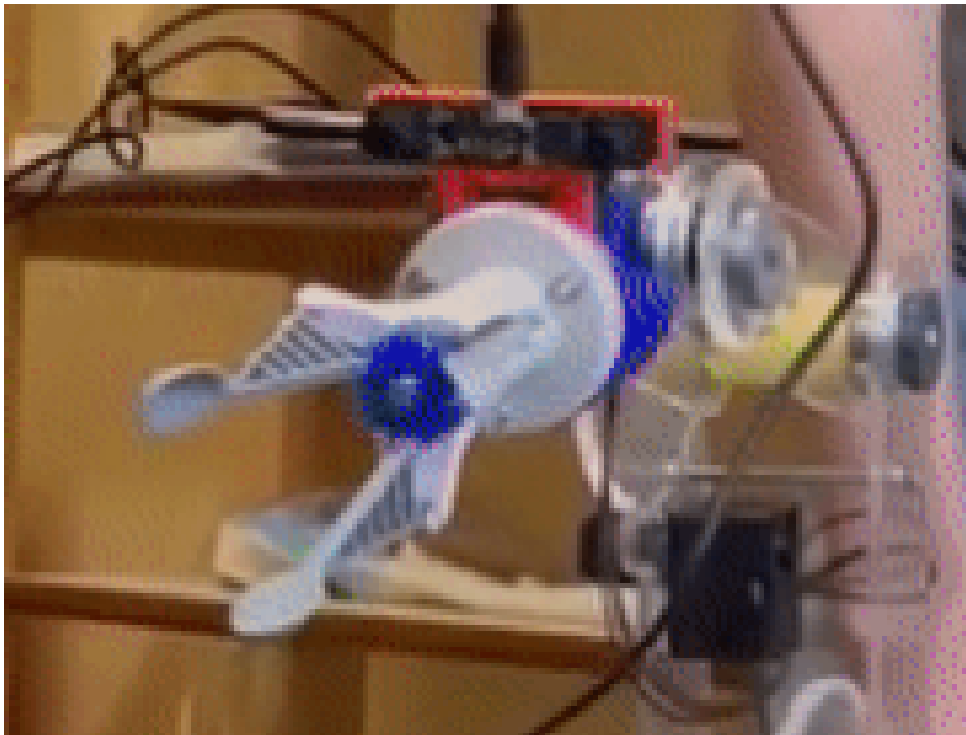


Figure 6.1: Camera position



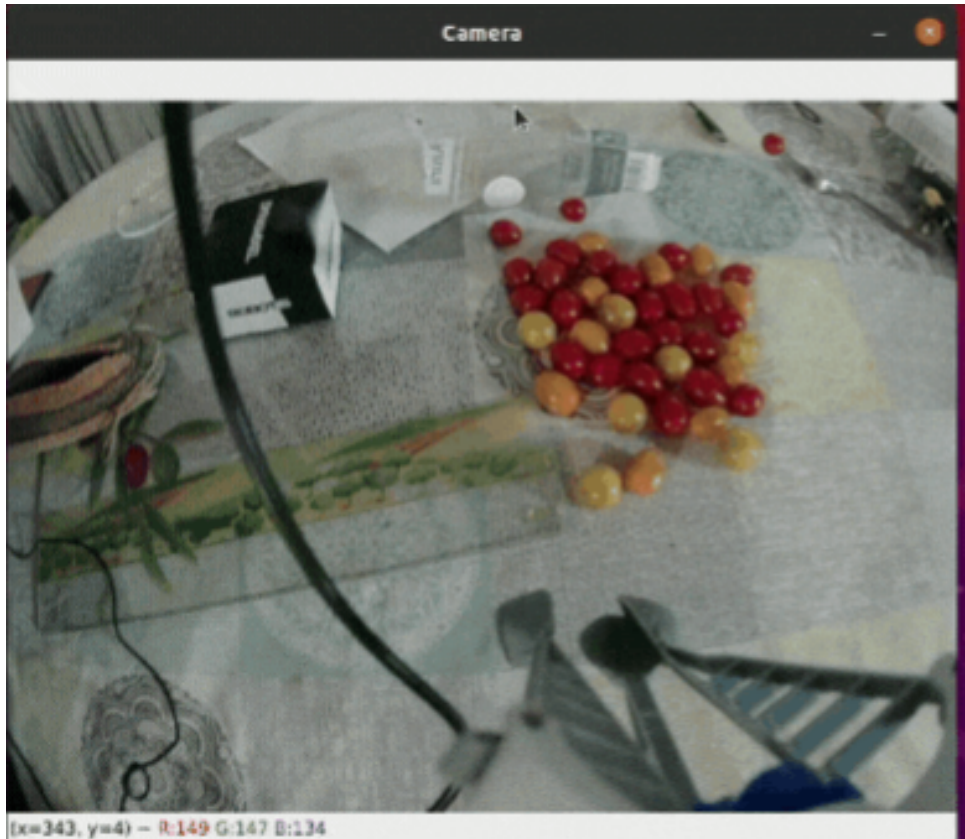


Figure 6.2: Camera view

The control of the three axes in the camera frame is done with the two joysticks on the joystick. The left joystick allows to move according to the plane (x,y) of the camera and the right joystick manages the depth. Some features listed below have also been added on the buttons to facilitate the work of the user and the proper functioning of the arm.

- authorize the movement: button A
- prohibit the movement: button B
- return to zero position : button X
- return to deposit position : button Y



Figure 6.3: Xbox Controller

As explained in the previous section, the inverse kinematics calculates the necessary angles for a given position in the base frame  $\{s\}$ . However, it seemed more natural to us to make the user work in the camera reference frame  $\{c\}$  and then to change the reference frame. For this, we reused the modern robotics library. In the same way that we can calculate the transformation matrix between the base and the end effector from the position of the links, we can calculate the one between the base and the camera.

$$T_{sc} = \prod_{n=1}^4 e^{[S_i]\theta_i} M_c$$

Then the relation between the position in the reference frame and the position in the reference frame is the following:

$$p_s = T_{sc}p_c$$

Attention,  $T_{sc}$  being a 4x4 matrix,  $p_s$  and  $p_c$  are homogeneous positions:  $p = [p \ 1]$ . At each command sent by the user, we retrieve the position of each of the links and then we apply the following code that puts into practice the previous equations. We obtain the position in the base frame.

```

1      # import kinematics parameters
2      from parameters import *
3      import modern_robotics as mr
4      # receive angles and desired position from ros
5      thetalist = get_angles()
6      p_c = get_position()
7      p_c = np.array([p_c 1])
8      # get transformation matrix and change the position
9      t = mr.FKinSpace(m_c,screw_list,thetalist)
10     p_s = np.dot(t,p_c)
11     p_s = p[:3]
```

The instruction sent by the user is added to an offset. Indeed, we want to control the position of the center of the hand. This one is fixed in the camera frame since there is no motor link between the camera and the hand. Thus, we can calculate the position of the center of the hand in the camera frame  $\{c\}$  when the robot is in its zero configuration. This offset will then be added to the command. An absence of set point will not correspond to the position (0,0,0) but to the position of the hand so that the robot does not move. The calculation of this offset is done in the following way. The transformation matrix from the base of the camera  $M_c$  and of the end effector  $M_e$  in the zero configuration have been given in section 4. By multiplying these two matrices, we can obtain the transformation between the end effector and the hand  $T_{ce}$ . Then in the same way as we changed the reference frame from camera to base, we can pass the point (0,0,0) in the end effector frame into the camera frame.

$$\begin{aligned}
T_{ce} &= T_{cs}T_{se} = T_{sc}^{-1}T_{se} \\
p_c &= T_{ce}p_e \\
\text{offset} &= T_{ce} \cdot [0 \ 0 \ 0 \ 1]^T
\end{aligned}$$

```

1  # import kinematics parameters
2  from parameters import *
3  import modern_robotics as mr
4  # receive angles and desired position from ros
5  p_e = np.array([0, 0, 0, 1])
6  # get transformation matrix and change the position
7  t_ce = np.dot(np.linalg.inv(m_c),m_e)
8  translation = np.dot(t,p_e)[:3]

```

## 6.2 Open loop

In this section, we only present the operation and results of the open loop using the tools explained above. The implementation of each block as well as the information exchanges are done with the help of ROS. Everything is detailed in the dedicated part.

The user sends a desired position with the joystick, a change of reference frame is performed and then the necessary angles are calculated. The diagram above explains the different blocks used and the information exchanged between them. The camera block is separate because it does not really intervene in the loop. It is just used to obtain a video return.

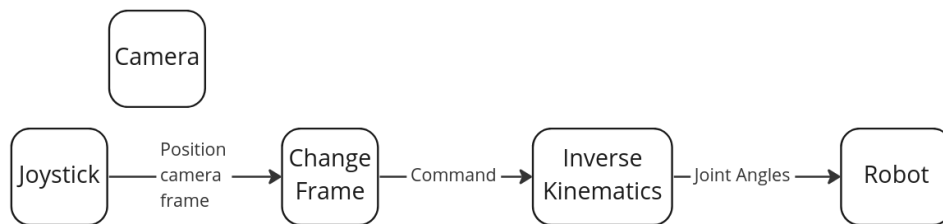


Figure 6.4: Open loop for joystick control

To test the open loop, we subjected our system to steps along the three camera axes. We measured the error, response time and overshoot. The results are detailed below. All the results presented were obtained in simulation. We did not buy a sensor to measure the position of the real robot. We have therefore based ourselves on the simulation results and on the observations of the arm movements to validate the control.

Using the joystick, the user gives an approximate position and corrects the desired position thanks to the camera feedback. As the results show, the open loop is accurate and allows to follow a position. Thus, it was not necessary to set up a closed loop on the real robot. We thus avoided buying a position sensor and saved development and delivery time.

## 6.3 Closed loop

However, we still servoed the robot in position in simulation. To stick to the reality, we have modified a little the robot in simulation compared to the reference robot used. Thus, we were closer to a real case and we were able to test a positional servoing using a proportional integral corrector.

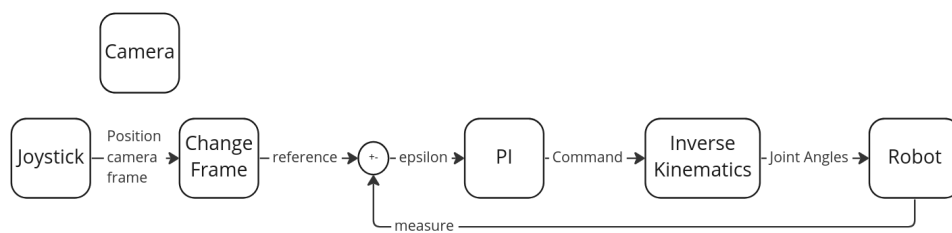


Figure 6.5: Closed loop for joystick control

## 7 ROS

For our project, we used `rosl noetic`.

### 7.1 Definition

As the name suggests, ROS (Robot Operating System) is an operating system for robots. Like operating systems for PCs, servers or stand-alone devices, ROS is a complete operating system for service robotics.

It is composed of a set of free open source software allowing to develop software for robotics. ROS is a meta operating system, something between the operating system and the middleware.

ROS is therefore positioned as a facilitator of robotics projects. Researchers or engineers in R&D departments no longer spend time creating a new ecosystem for each new robotics project. ROS has an accelerating effect on R&D by reducing costs and time to market. It is therefore very interesting for the rapid development of a prototype

**Master:** The master is a ROS process that helps nodes locate each other and establish communication channels based on their publisher subscriber relationships and any services. The master also manages the parameter server. The master is typically started through the `roscore` command-line tool, or it is automatically started via a `roslaunch` call.

**Topic:** A topic is a many-to-many information transport system. It is typed, it is necessary to specify which type of messages we transport and based on the system of subscription / publication

**Node:** A node is an instance of an executable; it can be linked to an engine, a sensor, or purely software. A node can post messages to a topic or subscribe to messages in a topic. They are independent processes, the crash of a node does not crash the whole ROS (Notion of micro-kernel on which ROS is based)

**Launch file:** Launch files are very common in ROS to both users and developers. They provide a convenient way to start up multiple nodes and a master, as well as other initialization requirements such as setting parameters.

**Package:** A collection of source code, configuration files, and other resources that implements functionality in ROS. By dividing related functionality into packages, it can be shared with others and reused across projects. All packages must have a `package.xml` file defining its manifest.

**Workspace:** A workspace is a folder where you modify, build, and install packages.

### 7.2 Training

Following the advice of our consultant, we quickly understood the advantage of using ROS. For that, the members of the automatic pole followed trainings to understand how to use it, to understand the principle, the functioning and the possible applications. Fortunately, the site offers many tutorials to learn through practice. This training was necessary and should be done as soon as possible to become familiar with totally foreign tools. We did everything using python and because we had developed everything with it.

This training on ROS is over. We knew how to create a workspace, packages and nodes. We also decided to link Matlab and ROS to accelerate the development. So we followed the tutorials proposed by matlab for the use of ROS with simulink and the development of package from model Simulink. This allowed to accelerate the tests but especially the handling of ROS because the commands were relatively automated by Matlab. However, we were less free for the package creation. Indeed, once the model is generated, it is difficult to directly modify the package code. You have to go back to the Simulink model and regenerate the package. Once we understood how python packages work, we limited the export of Simulink to packages to use more python node which are more easily and faster to modify. We still continued to use the connection between ROS and Matlab for testing.

### 7.3 Packages

For this project, we decided to separate the tools by categories. Four packages have been created and one is from an existing ROS package.

- **arm** : It contains all the nodes concerning the arm (forward kinematics, change frame, joystick conversion). It also contains all the information about the arm (kinematics parameters, rviz file, gazebo file, stl file) allowing the simulation with ROS only.
- **inverse kinematics** : The inverse kinematics having been generated from Simulink, it constitutes a separate package containing only the inverse kinematics node.
- **motors** : It contains all the codes related to the motors. The node that listens to the instructions and sends them to the engines.
- **camera** : It contains all nodes related to the camera (webcam display, object tracking, object detection)
- **joy** : It allows to get Xbox controller data.

### 7.4 Topics

We list here all the topics that will be used in the command chain. Be careful, inside the nodes, they can have different names to be more generic and reusable in other projects without having to modify them. But as we will see, we can change the name of these topics at the time of launch to give them a new value linked to the project. It is therefore this name that we make explicit.

- `/Joystick/ref/joy` : [Joy (sensor\_msgs)] joystick movement and button pressed
- `/Camera/ref/position` : [Point (geometry\_msgs)] command position in camera frame
- `/Robot/ref/position` : [Point (geometry\_msgs)] command position in base frame
- `/Robot/state/joint` : [JointState (sensor\_msgs)] position of each joints
- `/Robot/state/position` : [PoseStamped (geometry\_msgs)] end effector position in base frame
- `/Robot/state/reference` : [PointStamped (geometry\_msgs)] end effector target position in base frame
- `/EndEffector/state/position` : [Point (geometry\_msgs)] end effector state position in base frame (forward kinematics)
- `/Motors/state/current` : [JointState (sensor\_msgs)] current delivered by each motors
- `/Motors/state/present_position` : [JointState (sensor\_msgs)] real position of each motors

## 7.5 Nodes

To start a node, you have to source ROS, source the workspace that contains the package if this has not been done in the terminal and then launch the node.

```
~$ source /opt/ros/noetic/setup.bash
~$ roscore
~$ rosrn <package_name> <node_name>
```

For all python nodes, a class has been created. It allows to initialize the node and the variables used but also to indicate the name, the frequency and the topics to subscribe and publish. An update function is also present which takes care of publishing and listening at the requested frequency. An example is given in the appendix.

### 7.5.1 Forward kinematics

The forward kinematics worked very well with python with results equivalent to  $10^{-4}m$  compared to the simulink model. So we started from this file to create the node. This node is part of the arm package.

It subscribes to the topic `/Robot/ref/joint` and publishes `/EndEffector/state/position`. The calculation is done exactly the same way as explained in part 5.2

### 7.5.2 Inverse kinematics

The inverse kinematics node was created from simulink. The python model did not provide any results, so it was easier to use the code generator of matlab.

The node subscribes to the topic `/Robot/ref/position` and publish the topic `/Robot/ref/joint`. It publishes the x,y,z coordinates of the desired position. The output topic publishes a list containing the position of each link as well as an other one with their names and a header indicating the time.

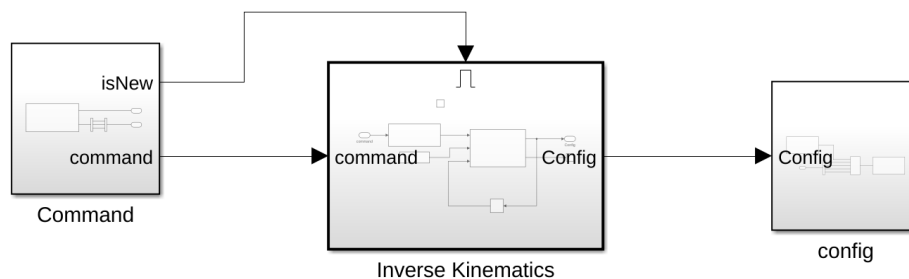


Figure 7.1: Inverse kinematics node

To connect to ROS with Matlab you will need to configure ROS Network Address. You can access this in the **Simulation tab** by selecting **ROS Toolbox > ROS Network**. You will need to set :

- ROS Master
- Network Address

To run files that contain ros block, you need a ROS master. You can run it on a terminal with the following commande :

```
~$ source /opt/ros/noetic/setup.bash
~$ roscore
```

The ros folder contains files that are generated by Matlab to create ros packaga from a simulink file. To generate them you need set parameters. You can access this in the **modeling tab > Model Settings > Hardware Implementation > Hardware board settings > Target hardware resources > Groups > Build Options**. You need to set :

- Device Adress (127.0.0.1 if localhost)
- Username
- password
- Ros folder (already set)
- path to catkin workspace

Then you just have to generate the files. It will create :

- build\_ros\_model.sh
- <model\_name> .tgz

From the folder where are the files, you can now create your package with the following command. The package and the node will have the same name as the simulink model.

```
~$ ./build_ros_model.sh <model>.tgz <catkin_workspace_path>
```

### 7.5.3 Joystick

To get the information from a controller, we used an existing package available on the ROS wiki site. This one allows to get the joysticks movements and the preset buttons for any controller. Only the order of the information can change depending on the controller.

In our case, we used an Xbox controller which is the one presented in the package. Then we created a node that converts the actions of the controller into the robot's command. The buttons used have already been specified in section 6.1 as well as the use of joysticks. The joysticks take values between -1 and 1 depending on the inclination. For a smooth movement and to avoid too big jumps between two topic sends (100Hz), we multiply this value by 0.03. Thus, the robot can move only 3cm along each axis between two sends.

### 7.5.4 Change frame

As explained in section 6.1, the command is sent to the camera frame. It is then necessary to change the reference frame using the formula and the code seen previously.

The node listens to the topics */Robot/state/joint* and */Camera/ref/position* in order to calculate the position in the base frame. It then publishes the topic */Robot/ref/position*. This one is sent only if there is a new command.



### 7.5.5 Motors

In addition to the advantages mentioned above, there is a dynamixel python library allowing to send commands and to get information from the motors. A ROS package also exists, which allowed us to start from this one to create our own. All the examples can be found on the dynamixel website. (dynamixel SDK and dynamixel workbench). The ROS examples are in C++ but are easily transferable to python.

The node for motors allows to transform the value of each link into a motor command. It listens to the node /Robot/state/joint. Then it converts this value according to the present reducer and passes it to the indicated unit of the motors.

The motor angles are not in radian, the documentation indicates that they can take values between  $\pm 1,048,575$  and that one turn ( $2\pi$ ) corresponds to 4,095. At startup, the robot must be in its zero configuration. We then get the value of each motor at time zero which is then used as an offset for the command. We also multiply the desired position by a variable called orientation worth  $\pm 1$  to ensure the correct direction of rotation of the motor.

To make sure that the robot is working properly and that it will not break, the node also publishes the current and the position of each motor. These are obtained directly via functions of the dynamixel library.

Below are the two functions that control the robot and publish the current and position of the motors.

```
1  def getGoalPositions(self):
2      """ transform joint position into a motor position """
3      pos = self.joints*self.gearRatio
4      pos = pos*4095/(2*math.pi)
5      pos = self.initPosition+self.orientation*pos
6      self.goalPositions = pos.astype(int)
7
8  def actuate(self):
9      while not rospy.is_shutdown():
10         # write position
11         self.getGoalPositions()
12         self.motors.write_position(self.goalPositions)
13         # read and publish current and position
14         self.present_position = self.motors.read_position()
15         self.present_current = self.motors.read_current()
16         self.publish_current()
17         self.publish_present_position()
18         self.rosRate.sleep()
```

### 7.5.6 Camera

As it is, the camera node is a bit special. It doesn't listen and publish any topic but only displays a camera return from a webcam on the computer. Pour cela, nous avons utilisé la bibliothèque openCV

## 7.6 Launch files

Each of the created launch files allows to launch a complete function of the robot. Their content is described as a block diagram. A square represents a node and an arrow a topic.

To launch a launch file, you have to source ros, source the workspace and launch it :

```
~$ source /opt/ros/noetic/setup.bash
~$ source /catkin_ws/devel/setup.zsh
~$ roslaunch <package_name> <file>
```

### 7.6.1 Rviz

rviz is a 3D visualizer for the Robot Operating System (ROS) framework. It what we use to simulate our robot and visualize the movement.



Figure 7.2: Rviz launch file

### 7.6.2 Inverse kinematics



Figure 7.3: Inverse kinematics launch file

### 7.6.3 Open loop

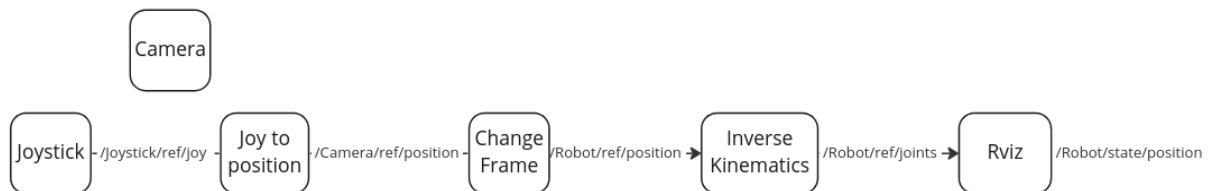


Figure 7.4: Open loop launch file

### 7.6.4 Closed loop

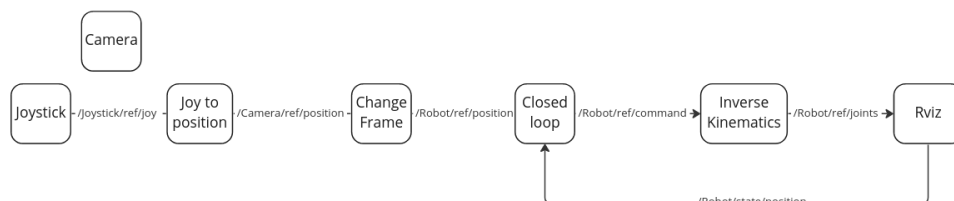


Figure 7.5: Closed loop launch file

## Conclusion

## A Forward kinematics python node

```
1  #!/usr/bin/env python3
2  import rospy
3  from geometry_msgs.msg import Point
4  from sensor_msgs.msg import JointState
5  from kinematics.parameters import *
6
7  class Forward_Node:
8      """
9      Node for the forward kinematics of the arm
10     """
11     def __init__(self,rosName="forward_node"):
12         # Init ROS node
13         rospy.init_node(rosName, anonymous=True)
14         try:
15             rate = rospy.get_param('/rate')
16         except :
17             rate = 100
18         self.rosRate = rospy.Rate(rate)
19         # init variables
20         self.config = m_e
21         self.thetalist = np.array([0,0,0,0])
22         self.point = np.array([0,0,0,0])
23         # init parameters
24         self.screw_list = screw_list
25         self.config_init = m_e
26         # init publisher and subscriber
27         self.initGraph()
28
29     def initGraph(self):
30         self.pub = rospy.Publisher('/EndEffector/state/position',
31                                     Point,
32                                     queue_size=10)
33         self.sub = rospy.Subscriber('/Robot/ref/joint',
34                                     JointState,
35                                     self.on_receive_callback,
36                                     queue_size=10)
37
38     def on_receive_callback(self,data):
39         self.thetalist = np.array(data.position)
40
41     def publish(self):
42         msg = Point()
43         msg.x = self.point[0]
44         msg.y = self.point[1]
45         msg.z = self.point[2]
46         self.pub.publish(msg)
47
48     def updateConfig(self):
49         while not rospy.is_shutdown():
```

```

50         self.config = mr.FKinSpace(self.config_init,
51                                     self.screw_list,
52                                     self.thetalist)
53         self.point = self.config.dot(np.array([0,0,0,1]))[:-1]
54         self.publish()
55         self.rosRate.sleep()
56
57     def main():
58         forward = Forward_Node()
59         forward.updateConfig()
60
61     if __name__ == '__main__':
62         try:
63             main()
64         except rospy.ROSInterruptException:
65             pass

```