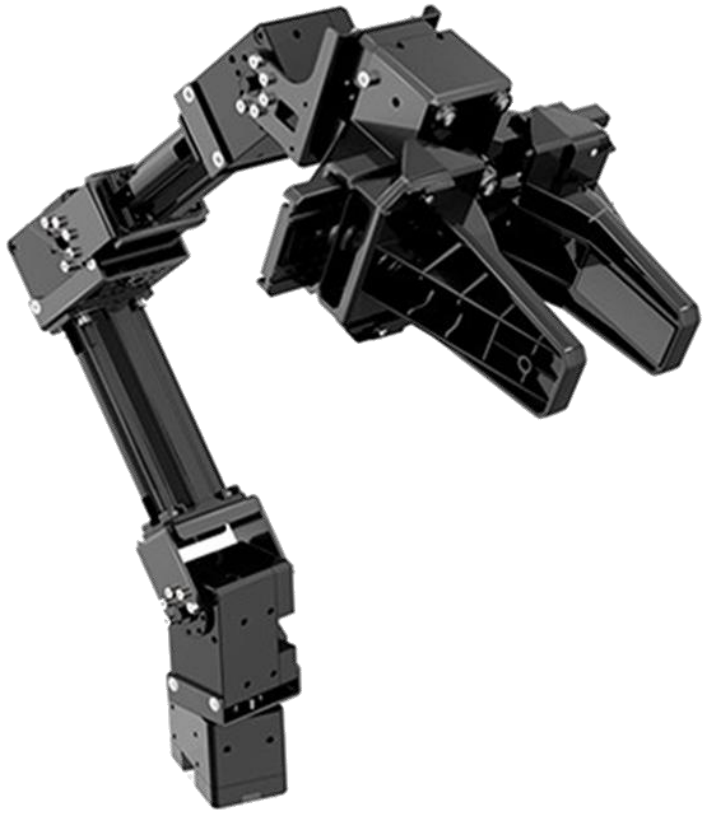
Development of a small-scale ball-tossing robotic arm[1]



ANSEAUME Louis – DANESI Nathaël – DONG Yuanhao – HAZEBROUCK Félix – MAHAUT Raphaël – MRABET Hatim – PRETET Thibault – SARKIS Christian

[1] Jokingly referred to as “King Pong” by the team.

0 – Table of Contents

I – Introduction

I.1 – Context

The work presented in this document is meant to represent the continuation of last year's work with the very same robotic arm (Openmanipulator-X Robotis RM-X52-TNM), the report of which is provided in appendix. While last year's work revolved around the sorting of various materials erratically positioned on a robotic arm's workbench, the work detailed in this report is centered around the tossing by a robotic arm of a small item into a target container.

The problem of teaching a robot to properly throw an item into a target is a vastly complex one – as evidenced by a recent paper published *IEEE Transactions on Robotics* [1] – but remains nonetheless very relevant today as a potential means to drastically improve both performance and range of standard industrial robotic arms. A robot's potential ability to accurately throw items into targets could be combined with sorting schemes to completely remove the need of human action in some extremely tedious and sometimes dangerous works, such as the sorting of mechanical components in an assembly line or that of heavy, radioactive or otherwise threatening waste.

Therefore, tackling the following problems appears as a most relevant endeavor:

I.2 – Problem statement

- How does a robotic arm manage to throw an item according to a defined trajectory?
- How does a robotic arm locate the items to throw and its targets in 3D space?
- How does a robotic arm know which trajectory the thrown item should follow?

I.3 – Goals and approach

The work presented in this report tackles the small-scale problem of getting a robotic arm to throw a table tennis ball in a small cardboard cup as a simplified mock-up of larger scale industrial efforts.

Indeed, several simplifications were made for the sake of simplicity and of time-constraints respect: the regular shape of the thrown ball completely eliminates the herculean task of determining how to grasp an item to better throw it, and the chosen target – an immobile cardboard cup facing upwards – is very simple compared to potentially mobile sorting boxes of various shapes and orientations.

In order to allow the robotic arm to detect the target cup in 3D space, it will be connected to a Zed2 stereo camera for depth sensing. The main steps of the work presented in this document are the following:

- Detect the target cup and measure its position in 3D space using the Zed2 camera.
- Send the position of the cup relative to the robot to the latter as a target.
- Determine the movements of the robotic arm's joints best suited for long-distance throwing.
- Determine the trajectory that the thrown ball should follow, and the commands to send to the robot's joints so that such trajectory is indeed followed after throw.

II – Computer vision

II.1 – Goal statement and assumptions

This section focuses on the three main tasks that the Zed2 camera must accomplish to help the robotic arm throw the ball in the target cup:

- Detect the cup.
- Measure the cup's coordinates in 3D space.
- Convert the cup's coordinates into the robot's frame to send a relevant target to the robot.

It should be noted that these three tasks are related to different fields: they require a combination of 2D image processing to detect the cup in the first place, 3D vision to measure the cup's position in space, and frame change linear algebra to convert said position into the robot's frame.

In addition, we chose to only make a single assumption regarding the position of the camera relative to the robot, in order to maximize ease of use: the camera can be placed at any reasonable distance and in any orientation as long as both the robot itself and the target cup remain visible enough in its field of view. Consequently, the camera also needs to detect the robot itself to perform the frame change into the robot frame.

Lastly, we deemed the 2D image processing necessary to detect the cup more complex than the one which would be needed to detect the ball that the robot must throw. Therefore, we assumed the position of the ball to be known, and the only objects that must be detected by the camera are the robot itself and the cup.

II.2 – Presentation of the Zed2 camera and its SDK

The Zed2 camera (See Figure 1 below) is an industrial grade wired stereo camera, that is a pair of two cameras facing in the same direction that are used together to see in 3 dimensions (much like a pair of eyes), produced by StereoLabs.

Figure 1: Zed2 stereo camera.

The Zed2 comes with a Software Development Kit (SDK) containing several features, including the camera's ability to measure depth and 3D positions. The SDK must be installed in order for the code provided in appendix to work and in order to use the camera at all, and requires the installation of CUDA as alongside it to perform computations (an installation guide can be found on StereoLabs's website following this link: https://www.stereolabs.com/docs/get-started-with-zed/).

II.3 – Introduction to stereo vision

This section briefly introduces the key points of stereo vision, also known as 3D vision, to offer a basic understanding of the way it works. The purpose of this section is to act as a failsafe, should the Zed2 camera ever cease to properly function or should no computer able to install the Zed2 SDK be available for an extended period of time, so that stereo vision could be recreated using 2 standard cameras and OpenCV. If necessary, more detailed guidelines can be found in [2] and [3].

II.3.A – Mathematical depth problem and need of 2 cameras

The main idea behind stereo vision is that one needs at least 2 cameras in order to see in 3D, as a single camera cannot perceive depths on its own (again, closing one eye deprives someone of their sense of depth). From a mathematical standpoint, the problem of getting the depth can be written as follows (see Figure 2):

- Let X be a point in space, the depth of which is unknown and is attempted to be estimated.
- Let C_1 and C_2 be the optical centers of two cameras in space, with X within their field of view.
- Let $\overrightarrow{L_1} = \frac{1}{\|\overrightarrow{C_1X}\|}\overrightarrow{C_1X}$ and $\overrightarrow{L_2} = \frac{1}{\|\overrightarrow{C_2X}\|}\overrightarrow{C_2X}$ be the unit vectors representing the direction of the lines (C_1X) and (C_2X). In reality, these vectors themselves are unknown, but some similar concept of direction can be extracted from the position of the pixels of X on a camera's image.

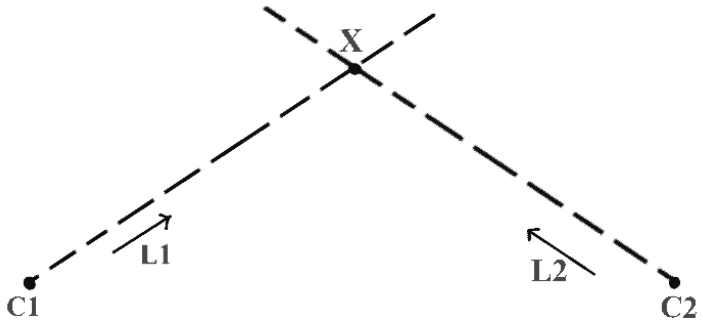


Figure 2: Diagram of the depth estimation problem (source: [2]).

If the camera C_1 acts alone, the only equation that can be written is:

$$\exists k_1 \in \mathbb{R}, \qquad \overrightarrow{OX} = \overrightarrow{OC_1} + k_1\overrightarrow{L_1} \tag{1}$$

Where $\overrightarrow{OX}$ represents 3 unknowns (the 3 cartesian coordinates of X) and k_1 is unknown. Hence, (1) is a 3-dimensional equation with 4 unknowns and cannot be solved alone. Hence, we need the presence of the second camera C_2, which allows to write:

$$\exists k_2 \in \mathbb{R}, \qquad \overrightarrow{OX} = \overrightarrow{OC_2} + k_2\overrightarrow{L_2} \tag{2}$$

Combining equations (1) and (2) yields:

$$\overrightarrow{OC_1} + k_1\overrightarrow{L_1} = \overrightarrow{OC_2} + k_2\overrightarrow{L_2}$$

$$\Leftrightarrow \overrightarrow{C_2C_1} + k_1\overrightarrow{L_1} = k_2\overrightarrow{L_2} \tag{3}$$

Which is a 3-dimensional equation with only 2 unknowns and can be solved. From acquiring the value of k_1 and k_2 this way, the cartesian coordinates of X can be deduced using (1) or (2) (for simpler

computations, the origin O is often chosen to be C_1). Notice that it is only necessary to know the position of one camera relative to the other.

II.3.B – Numerical depth problem and simplification

The mathematical problem presented above proved that it is both necessary and sufficient to have at least 2 cameras to measure the 3D coordinates of a point in space. However, a substantial numerical problem was completely overlooked there: if camera C_1 wishes to measure the position of a point X that it can see (it therefore knows the vector $\overrightarrow{L_1}$), how does camera C_2 know which pixel on its picture corresponds to the same point X (i.e. how does camera C_2 build the vector $\overrightarrow{L_2}$) ?

- At first glance, this appears as a potentially very complex issue, as both cameras can be seeing things differently from different angles and as camera C_2 has to look at every group of pixels on its picture trying to locate X. However, this problem can be simplified a lot. Let us define the following:Let
-
- Let the pictures taken by cameras C_1 and C_2 be assimilated to planes orthogonal to their optical axes and positioned a bit forward from points C_1 and C_2 (planes i_1 and i_2 on Figure 3 below).
- Let x_1 be the pixel associated to X on C_1's picture, which is known and located on (C_1X).
- Let x_2 be the pixel associated to X on C_2's picture, which is unknown and located on (C_2X).
- Let P be the plane generated by points C_1, C_2 and X. Note that points x_1 and x_2 belong to P.

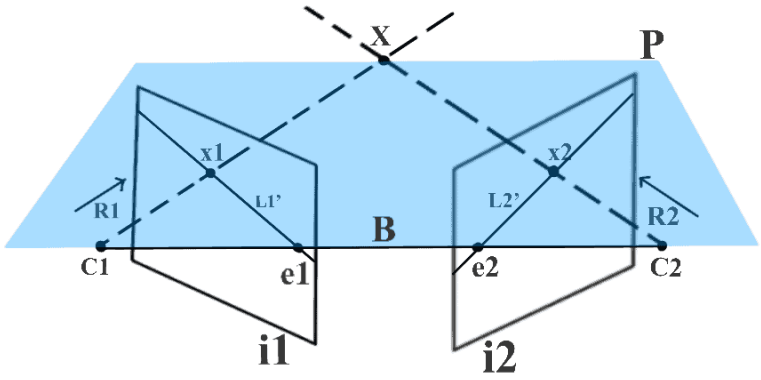


Figure 3: More precise diagram of the depth estimation problem (source: [2]).

With all these elements, the problem of looking for x_2 on C_2's picture with x_1 known becomes significantly easier. Indeed, the plane P cuts through C_2's picture to create a line, however since x_2 belongs to both the picture and P, it also belongs to that line. Hence, from a mathematical standpoint, there exists a line on which it is sufficient to search for x_2. However, finding that line in practice isn't that easy – except in a very particular case: when both cameras are facing in the same direction.

Indeed, if the two cameras have parallel optical axes, then the line created by the intersection of P and C_2's picture becomes vertical. Not only that, P cuts through C_1's picture in the same way to create another vertical line with the exact same height as the first one. Consequently, knowing x_1 indicates where to look on C_2's picture: on the only line of pixels located on the same row as x_1!

Despite these tremendous simplifications, the problem of looking for a pixel on a picture corresponding to another on a different picture remains a complex task. It can be accomplished

through looking at the pixels around the first one and trying to find a "similarly-looking" group of pixels on the second picture – more details about this can be found in [3].

II.4 – Frame change

In this section, we focus on the first main task that the Zed2 camera must tackle: perform a frame change into the robot arm's frame.

II.4.A – Zed2 camera's frame and robotic arm's frame

The frame conventions for the Zed2 camera and the robot arm can be found respectively on Figure 4 and f below:

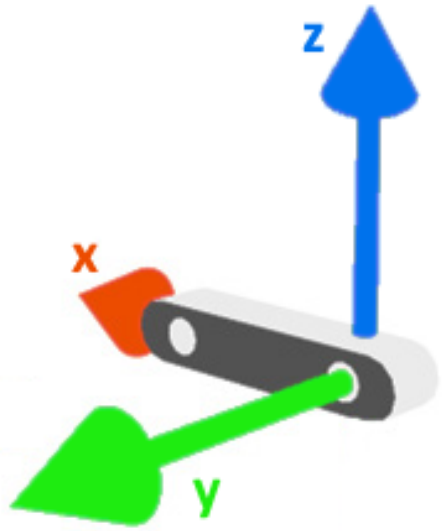


Figure 4: Chosen frame for the Zed2 camera (right-handed, z up).

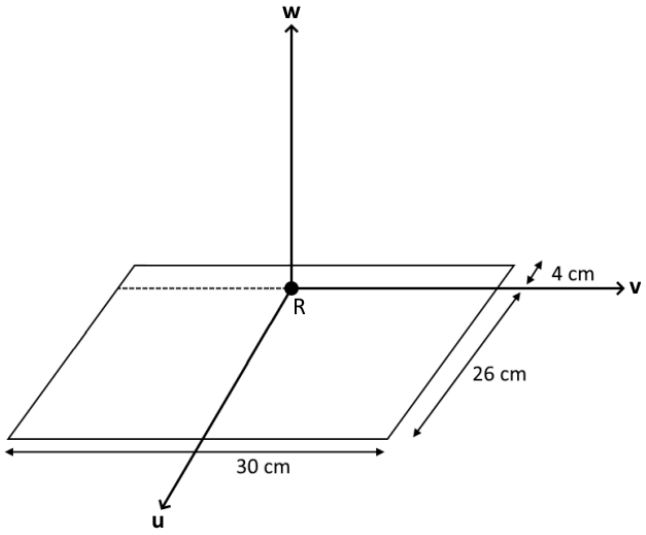


Figure 5: Robot arm's frame. Point O here acts as both the origin and the anchor point of the arm on the workbench.

It should be noted that the relative position on these two frames can be very complex as it was assumed that the Zed2 camera could be positioned in any way that allows it to observe both the robot and the target cup. The aim now becomes to build the frame change matrix, assuming that the robot's frame is known to the camera.

II.4.B – Elementary translation and rotation matrices

In order to build the frame change matrix, we will compute the product of the various elementary translation and rotation matrices corresponding to the operations to apply in sequence. These elementary matrices are the following:

The elementary translation matrix is:

$$\mathcal{M}_t(x,y,z) = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ such that } \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix} + \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \mathcal{M}_t(x,y,z) \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix}$$

The elementary rotation matrices are:

$$\mathcal{M}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ such that } \mathcal{M}_x(\theta) \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix} \text{ is } \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix} \text{ rotated by } \theta \text{ around the x-axis.}$$

$$\mathcal{M}_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ such that } \mathcal{M}_y(\theta) \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix} \text{ is } \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix} \text{ rotated by } \theta \text{ around the y-axis.}$$

$$\mathcal{M}_z(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ such that } \mathcal{M}_z(\theta) \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix} \text{ is } \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix} \text{ rotated by } \theta \text{ around the z-axis.}$$

It should be noted that, in every operation involving these matrices, 4D vectors are needed. The fourth component of these vectors is always 1 and has no signification in our case. In the following section, a 3D vector and its augmented 4D counterpart with fourth component equal to 1 will be written in the same manner.

II.4.C – Frame change algorithm

The goal of this section is to build the frame change matrix which allows to change from the Zed2 camera's frame (where the cup's coordinates will first be measured) to the robot arm's frame. In order to do this, we assume that the camera's frame is the standard reference frame, i.e.:

$$\mathcal{F}_{cam} = \left(O = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \vec{x} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \vec{y} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \vec{z} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right)$$

And we assume that the coordinates of the robot frame's origin and vectors in the camera frame are known. The robot frame is denoted by:

$$\mathcal{F}_{rob} = \left(R = \begin{bmatrix} x_R \\ y_R \\ z_R \end{bmatrix}, \vec{u} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix}, \vec{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}, \vec{w} = \begin{bmatrix} w_x \\ w_y \\ w_z \end{bmatrix} \right)$$

Under these assumptions, we first aim to build the opposite frame change matrix, i.e. the matrix allowing to change from the robot frame to the camera frame, by identifying the operations which, when applied in sequence to the robot frame, allow to move the robot frame in a way such that it becomes identical to the camera frame.

The first step to do so is to translate the robot frame's origin R onto the camera frame's origin O. This can be done by applying the translation matrix $\mathcal{M}_t(-x_R, -y_R, -z_R)$ to the point R.

Now, we wish to align the frames' vectors. Let's begin by aligning $\vec{u}$ with $\vec{x}$, and the first step in doing so is to rotate $\vec{u}$ around $\vec{x}$ into the (O, $\vec{x}$, $\vec{y}$) plane – which is the same problem as that of rotating the projection of $\vec{u}$ into the (O, $\vec{y}$, $\vec{z}$) plane, denoted by $\overrightarrow{u_{yz}}$, around $\vec{x}$ so that it matches $\vec{y}$ (see Figure 6 below).

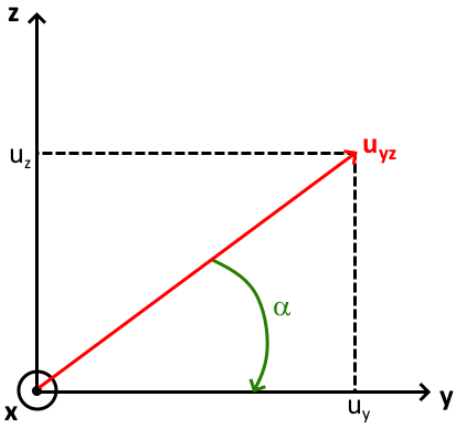


Figure 6: First rotation necessary to change the robot frame into the camera frame.

Let α denote the negative angle by which we must rotate $\overrightarrow{u_{yz}}$. Its cosine and sine can be written:

$$\begin{cases} \cos(\alpha) = \dfrac{u_y}{\sqrt{u_y^2 + u_z^2}} \\ \sin(\alpha) = -\dfrac{u_z}{\sqrt{u_y^2 + u_z^2}} \end{cases}$$

This allows to build a first elementary rotation matrix $\mathcal{M}_x(\alpha)$ such that $\mathcal{M}_x(\alpha)\vec{u} \in (O, \vec{x}, \vec{y})$.

Now, we need to rotate $\mathcal{M}_x(\alpha)\vec{u}$ around $\vec{z}$ so that it matches $\vec{x}$ (see Figure 7 below).

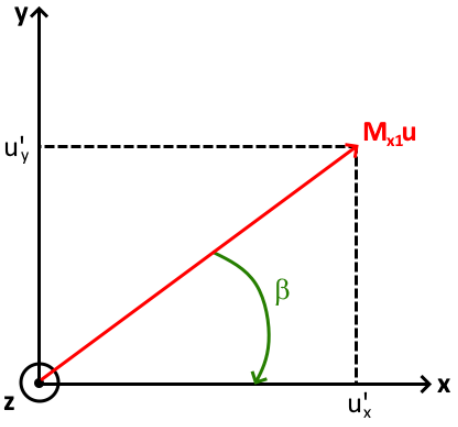


Figure 7: Second rotation necessary to change the robot frame into the camera frame.

Let β denote the negative angle by which we must rotate $\mathcal{M}_x(\alpha)\vec{u}$. Its cosine and sine can be written:

$$\begin{cases} \cos(\beta) = \dfrac{u'_x}{\sqrt{u'^2_x + u'^2_y}} = u'_x \\ \sin(\beta) = -\dfrac{u'_y}{\sqrt{u'^2_x + u'^2_y}} = -u'_y \end{cases}$$

Where $\mathcal{M}_x(\alpha)\vec{u} \coloneqq \begin{bmatrix} u'_x \\ u'_y \\ 0 \end{bmatrix}$ is of norm 1 since $\vec{u}$ is a unit vector.

This allows to build a second elementary rotation matrix $\mathcal{M}_z(\beta)$ such that $\mathcal{M}_z(\beta)\mathcal{M}_x(\alpha)\vec{u} = \vec{x}$. One can notice that $\mathcal{M}_z(\beta)\mathcal{M}_x(\alpha)\vec{v}$ and $\mathcal{M}_z(\beta)\mathcal{M}_x(\alpha)\vec{w}$ are in the (O, $\vec{y}$, $\vec{z}$) plane.

Finally, all that is missing is a rotation around $\vec{x}$ which allows $\mathcal{M}_z(\beta)\mathcal{M}_x(\alpha)\vec{v}$ to match $\vec{y}$ (see Figure 8 below).

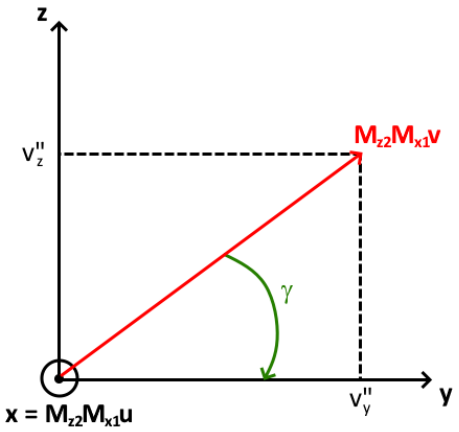


Figure 8: Third and last rotation necessary to change the robot frame into the camera frame.

Let γ denote the negative angle by which we must rotate $\mathcal{M}_z(\beta)\mathcal{M}_x(\alpha)\vec{v}$. Its cosine and sine can be written:

$$\begin{cases} \cos(\gamma) = \dfrac{v''_y}{\sqrt{v''^2_y + v''^2_z}} = v''_y \\ \sin(\gamma) = -\dfrac{v''_z}{\sqrt{v''^2_y + v''^2_z}} = -v''_z \end{cases}$$

Where $\mathcal{M}_z(\beta)\mathcal{M}_x(\alpha)\vec{v} \coloneqq \begin{bmatrix} 0 \\ v''_y \\ v''_z \end{bmatrix}$ is of norm 1 since $\vec{v}$ is a unit vector.

This allows to build a last elementary rotation matrix $\mathcal{M}_x(\gamma)$ such that:

$$\begin{cases} \mathcal{M}_x(\gamma)\mathcal{M}_z(\beta)\mathcal{M}_x(\alpha)\vec{u} = \vec{x} \\ \mathcal{M}_x(\gamma)\mathcal{M}_z(\beta)\mathcal{M}_x(\alpha)\vec{v} = \vec{y} \\ \mathcal{M}_x(\gamma)\mathcal{M}_z(\beta)\mathcal{M}_x(\alpha)\vec{w} = \vec{z} \end{cases}$$

Therefore, the complete frame change matrix going from robot frame to camera frame can be written:

$$\mathcal{M}_{rob\to cam} = \mathcal{M}_x(\gamma)\mathcal{M}_z(\beta)\mathcal{M}_x(\alpha)\mathcal{M}_t(-x_R, -y_R, -z_R)$$

And from this, we can deduce the frame change matrix going from camera frame to robot frame as it is the inverse of $\mathcal{M}_{rob\to cam}$. However, we can spare the computer some complex inversion computations by noticing that it is sufficient to reverse the order of the matrices in the product and the sign of the angles and coordinates:

$$\mathcal{M}_{cam\to rob} = \mathcal{M}_t(x_R, y_R, z_R)\mathcal{M}_x(-\alpha)\mathcal{M}_z(-\beta)\mathcal{M}_x(-\gamma)$$

The previously detailed steps constitute the algorithm implemented to build the frame change matrix $\mathcal{M}_{cam\to rob}$ (see file `geometry.py` in appendix).

II.4.D – Calibration setup and subsequent complications

There is still one component missing for the camera to properly perform the frame change. Indeed, in the previous subsection, we assumed that the robot frame was known to the camera, which requires the camera to actually locate the robot (including its orientation) in space.

We deemed that having the camera detect the robot – a task which is needed only once as the robot's workbench is immobile – was worth doing semi-manually, as considerations of orientation coupled with a need of precision render this issue very complex in a fully automated context. Consequently, we chose 3 specific corners of the robot's workbench that the camera would need to locate in space, with the help of the user, in order to learn the details about the robot's frame. These points, labelled A, B and C, are represented on Figure 9 below, would be detected by the camera by having the user position easily identifiable colored items on their locations.

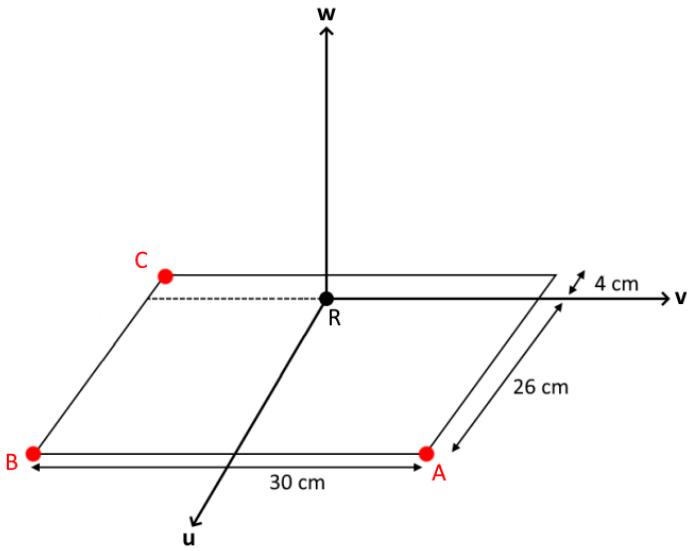


Figure 9: Position of the three corners A, B and C that the camera should detect.

Once the camera managed to detect all 3 points (i.e., measure $\overrightarrow{OA}$, $\overrightarrow{OB}$ and $\overrightarrow{OC}$ in its frame), it can deduce the robot's frame using the following relations:

$$\begin{cases} \overrightarrow{OR} = \overrightarrow{OC} + \frac{4}{30}\overrightarrow{CB} + \frac{1}{2}\overrightarrow{BA} \\ \vec{u} = \frac{1}{\left\|\overrightarrow{CB}\right\|}\overrightarrow{CB} \\ \vec{v} = \frac{1}{\left\|\overrightarrow{BA}\right\|}\overrightarrow{BA} \\ \vec{w} = \vec{u} \wedge \vec{v} \end{cases}$$

Two methods were explored for the camera to detect points A, B and C:

1. Position a colored item once and for all on the three corners of the workbench associated to the three points: a red item for A, a green item for B and a blue item for C, and have the camera look for these specific colors. While very practical on paper, this method proved trickier than expected as explained below and wasn't chosen.
2. Choose a red item and have the user manually position it on points A, B and C individually in sequence, and have the camera detect the red item each time. While more tedious for the user, this method proved simpler to implement and was chosen.

II.4.D.a – First solution to detect the points: a differently colored item for each

The first solution consited in applying masks on the image captured by the camera, so as to detect the objects that we have put next to the robot. At the first part, we were willing to detect three different objects, colored differently (Red, Green and Blue).

So the idea of the algorithm was, first of all to take an image, and set it from RGB to the HSV color system. After that, we set a range for each color, by affording an upper limit and lower one, and then for each color,we apply a mask using the predefined function in Python, in Opencv module, cv2.dialte() to the image to keep pixels that possess a color in the range afforded, finaly we detect the contours by using the predefined function cv2.findcontours(), to draw the contours of the colored objects. Finally, we show the different contours. To reduce noises, we set a lower limit of the area of the the object to be detected by the camera.

The first results were actually very great, since the algorithm was capable of detecting the colored objects on a sheet of paper reliably, as shown on Figure 10 below.

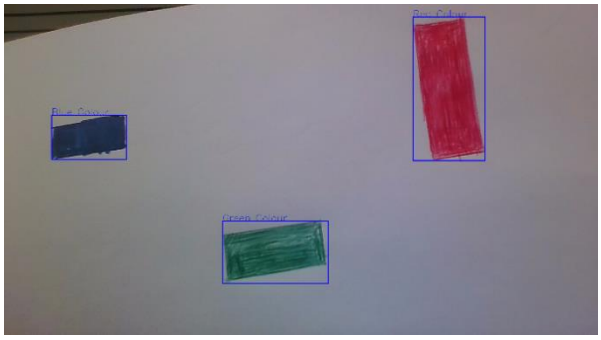


Figure 10: algorithm identifying three differently colored patches on a sheet of paper

The problem with this method is that it was actually very sensitive to noises, and to the place where we put the camera, since we noticed that if we put the camera in a very far place from the object to be detected, the camera captures other objects. In addition, if the camera is placed in a place where we find other object with the same color, the algorithm captures these other objects, as shown on Figure 11 below.

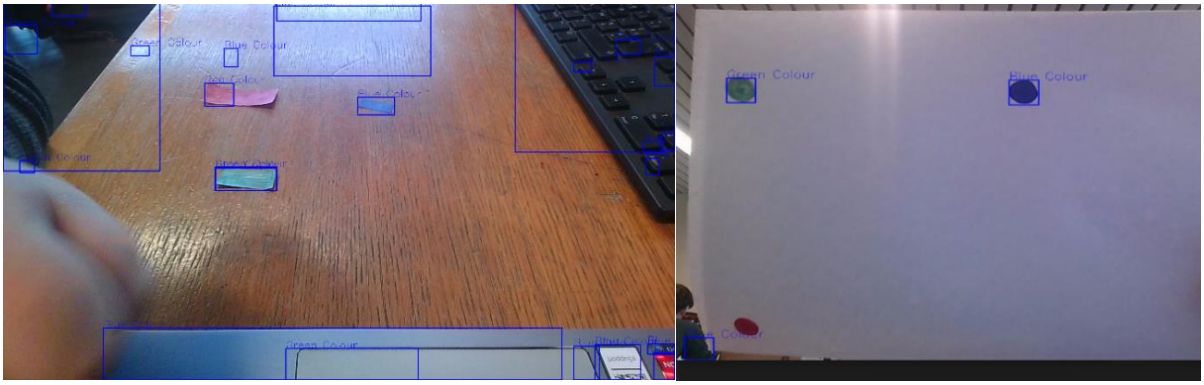


Figure 11: Instances of the algorithm producing massive errors when faced with noise.

II.4.D.b – Chosen solution to detect the points: locate a red item thrice

Instead of trying to simultaneously identify three differently colored items on the same 2D picture, we decided to simplify the task and only look for one. We chose to keep the red item as scarlet red is a rather uncommon color in usual work environments.

In order to identify the position on the image of the scarlet red object quickly and simply, we followed these two steps:

1. Apply a "red filter" to the image, i.e., remove any pixel with a red component considered "too low" (below a certain threshold) and remove any pixel with a green or blue component considered "too high" (higher than a certain threshold). To increase accuracy, we made both thresholds manually adaptable if necessary.
2. Compute the line and column of the weighed centroid of the remaining pixels.

The line and column yielded are, in practice, very reliably those of the center of the scarlet red item. In order to help the camera detect the 3 corners of the robot's workbench, a user simply has to position a scarlet red item on these corners individually and in succession.

It should be noted that, when tested, this method worked very well with the camera having a "bird's eye"-like view on the robot and the cup but failed catastrophically when the camera was positioned around the same height as the robot. This failure turned out to be caused by massive errors in depth measurements by the camera, which weren't due to calibration faults: the camera measured a distance of 3.8 meters for point B's object, actually located 1 meter from the camera, and of 3.3 meters for point C's object, actually located 1.3 meters from the camera. With a bird's eye view, such errors never manifested and the accuracy remained within a few percents away from the expected values, as advertised by StereoLabs's website (link for accuracy settings and more details: https://www.stereolabs.com/docs/depth-sensing/depth-settings/#depth-accuracy).

We have now finally reached the point where the camera's ability to see in 3D can be properly linked with the robot. Hence, all that remains to be done is to detect the target cup on a 2D image captured by the camera.

II.5 – Cup detection using AI

To detect the cup and communicate the information with the robotic arm, many options are available. To discriminate them, we want to choose a method that meets the following requirements:

a) To be efficient at detecting cups. It can seem obvious said like this, but some algorithms are suited for real-time tracking and will therefore be less successful when it comes to achieve

this task on a single picture. We deliberately chose to use a single picture instead of real-time tracking because the cup is not moving and treating one image is simpler.

b) To be precise. Of course, since the cup is small, a tiny error of a few centimeters can lead to missing the target, so our algorithm must have a maximum error of 2 or 3 centimeters.

c) A moral optional criterion: to have the least human intervention possible. We want our final result to work in the most situations possible, to be usable easily by most people and to be independent.

II.5.A – First method: using the 3D object detection module within the Zed2 SDK

After discovering the various abilities of our camera and the pyzed modules it contains, we found it already offers the possibility to detect objects in real-time.The ZED SDK uses AI and neural networks to determine which objects are present in both the left and right images. The SDK then computes the 3D position of each object, as well as their bounding box, using data from the depth module.

The 3d object detection module that comes with the ZED SDK is fast, accurate and easy to use (we only need to call a few functions). However, even if it is supposed to be able to detect objects from the "cup" category, it didn't manage to detect ours, probably because the cups considered in the training dataset for this algorithm were different from the one we use. We therefore had to abandon this lead.

II.5.B – Second method: detecting the top of a cup

We then considered using another method. After coloring the top of the cup in a specific color that is rarely present in the sight of the camera like blue or green for example, we determine the center of the considered color. After having found this point, which is where we will want to throw the ball, we retrieve its 3D coordinates and transmit it to the robot.

But this method goes against our criteria c), which, even if it's optional, reflects the way we want to achieve our task. So, as we found a better solution at the time we were developing this strategy, we decided to move on to the third technique.

II.5.C – Third method: using other object detection modules

For the problem of recognizing cups, the most stable and feasible method is to apply a neural network model. There are now several well-established neural network model-based object recognition algorithms for 2D images available on the web. However, to apply these algorithms to train a model of our own would require a large data set and time, which is almost impossible for us. We then wanted to find an existing trained neural network model on the Internet and apply it to our project. Fortunately, we found a pre-training module in PyTorch that can recognize our cups, **fasterrcnn_resnet50_fpn**, which uses an algorithm called Faster R-CNN. As it recognises a cup in the 2D image, it returns a rectangle that contains it. After selecting the point that corresponds to the middle on the x-axis and the 2 thirds on the y-axis of this rectangle, such that it corresponds to the center of the cup's hole, we call the 3D point cloud given by the camera and find the 3D coordinates of the selected point thanks to its pixel position in the image.

Here is a brief description of how this algorithm works:

The image is always complex (there may be multiple objects of different sizes and shapes). We need to find the objects that are present and then classify them. This is the object detection problem. Traditional object detection algorithms include Viola Jones Detector, HOG Detector, DPM Detector etc. In 2014, Ross Girshick proposed the R-CNN (Region-based Convolutional Neural Network) algorithm. The algorithm introduces deep learning to object detection for the first time, significantly improved efficiency. Faster R-CNN is a more efficient algorithm based on R-CNN.

II.5.C.a – R-CNN

In order to understand Faster R-CNN, we first need to understand the principle of R-CNN.

R-CNN inherits the idea of traditional target detection by first extracting a series of region proposals and then performing classes on the region proposals. Its process consists of the following four steps.

(1) Generation of Region Proposal.

The Selective Search (SS) algorithm is used to generate the region proposal. About 2000 candidate regions will be generated and scaled to a fixed size (227*227).

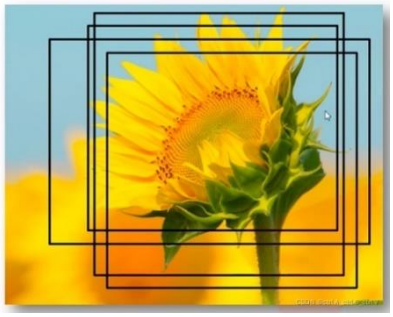


Figure 12: Many region proposals are generated on an image.

(2) For each region proposal, CNN is used to extract features.

Feed the region proposals into the pre-trained AlexNet CNN network to get a fixed dimensional feature output (4096 dimensions) and we obtain a feature matrix of size 2000×4096 (assuming there are 2000 region proposals).

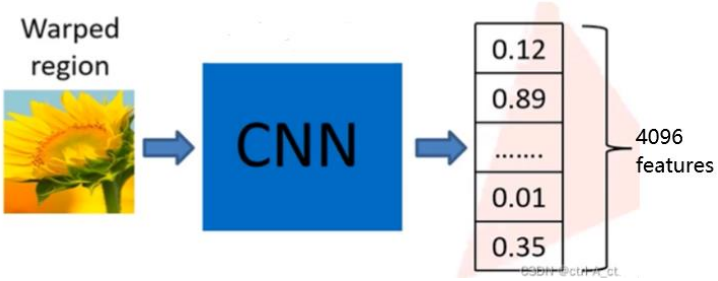


Figure 13: Each region was input into CNN to extract its features.

(3) Classification of CNN output features by SVM classifier

Suppose we have 20 pre-defined classes for the objects. For each class, we need a SVM to do the classification. The 2000×4096 feature matrix X is multiplied with a weight matrix W (4096×20) composed of 20 SVMs to obtain a 2000×20 matrix, which represents the probability that each of the 2000 candidate regions belongs to one of the 20 classes.

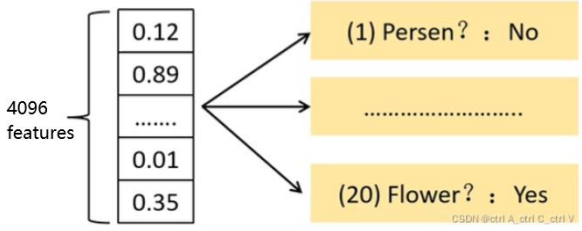


Figure 14: 20 SVMs are used to perform the classification.

Non-maximum suppression is applied to eliminate overlapping suggestion boxes and its process is as follows:

Step 1: Define the IoU index (Intersection over Union), i.e. $(A \cap B) / (A \cup B)$, which is the ratio of the intersection of AB to the union of AB. Intuitively, the IoU is the ratio of the overlap of AB, and the larger the IoU, the greater the proportion of AB overlap, i.e. the more similar A and B are.

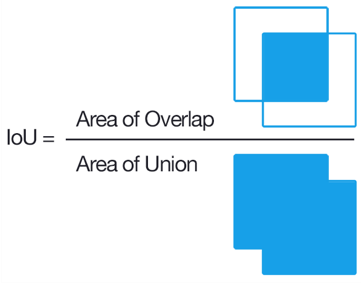


Figure 15: The definition of IoU.

Step 2: Find the region with the highest prediction score among the 2000 region proposals in each category, calculate the IoU values of other regions with it, and delete all regions with IoU values greater than the threshold. This allows only a few regions with a low overlap rate to be retained.

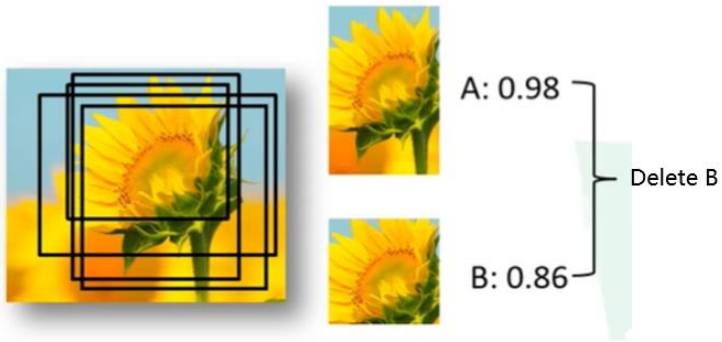


Figure 16: NMS is used to eliminate the redundant regions.

(4) Refinement of region proposal locations using regressors.

The region proposal locations obtained through the Selective Search algorithm are not necessarily accurate, so 20 regressors are used to perform regression operations on the remaining suggestion boxes in the 20 categories to finally obtain a revised target region for each category.

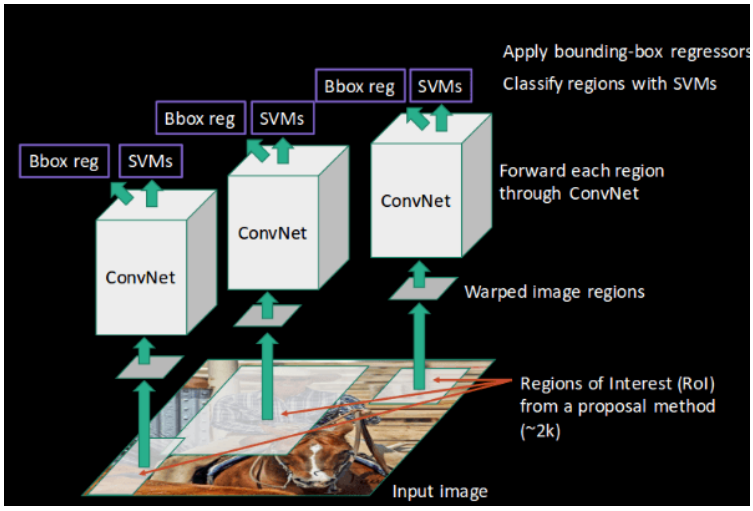


Figure 17: The overall process of R-CNN.

II.5.C.b – Fast R-CNN

The Fast R-CNN algorithm was proposed by Ross Girshick in 2015 and improves on the RCNN.

Fast R-CNN algorithm flow:

(1) An image generates 1K~2K region proposals (using the Selective Search algorithm), and we refer to a region proposal as a ROI (region of interest).

(2) Input the image into the CNN to get the feature map, and project the region proposals onto the feature map to get the corresponding feature matrix. Different from R-CNN who inputs 2000 region proposals into CNN one by one, it feeds the whole image into the network and calculates the whole image features at one time.

(3) Each feature matrix is scaled down to a 7x7 size feature matrix by ROI pooling layer.

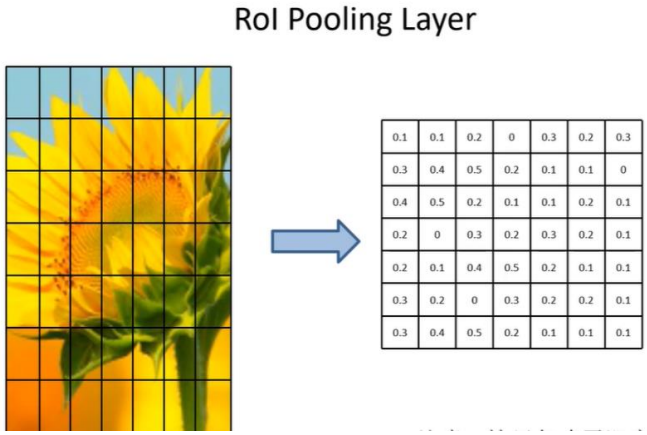


Figure 18: The RoI Pooling Layer scales down the region proposal to a 7x7size feature matrix.

As mentioned earlier, R-CNN needs to normalize the candidate regions to a fixed size (227x227), while Fast R-CNN does not need such an operation. Fast RCNN changes the feature map of each region proposal to 7x7 through the pooling layer.

(4) The feature maps are reshaped into vectors and the predictions are obtained through a series of fully connected layers and *softmax* classifier.

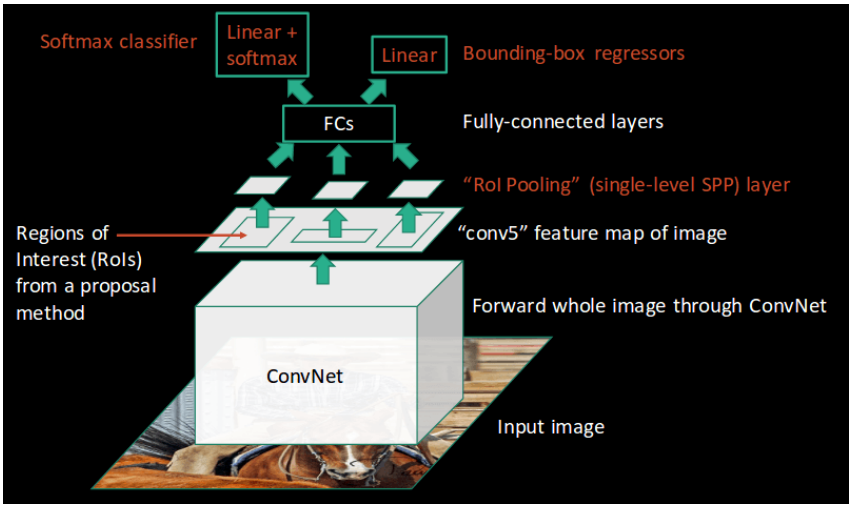


Figure 19: The overall process of Fast R-CNN.

II.5.C.c – Faster R-CNN

But we can be faster! In 2015, Faster R-CNN was proposed to further increase the speed. The biggest change of Faster RCNN is that the SS algorithm is replaced by RPN (Region Proposal Network) to generate region proposals. It projects the RPN-generated region proposals onto the feature map to obtain the feature matrices of the ROI.

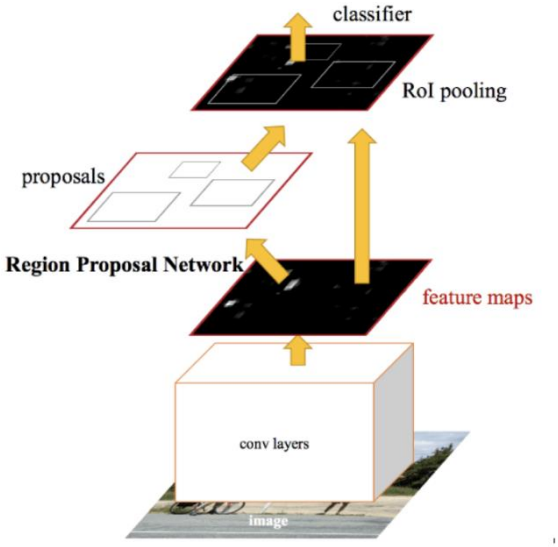


Figure 20. The overall process of Faster R-CNN.

II.5.C.d – Code implementation and results

We apply the model based on this algorithm in python. The model output is a list containing all the objects identified by the model and their positions in the image (the coordinates of the top left corner of the prediction box and the coordinates of the bottom right corner). We restrict the output to paper cups. This gives us the two-dimensional position of the cup from the RGB image. In the next step we can then use the 2D information to find the position of the corresponding points in the 3D point cloud map.

Figure 21: On this dummy picture, the cup is detected, and a box is drawn to indicate its position in 2D image.

III – Control of the robotic arm

III.1 – Stakes of the control-part

The objective of this part is to determine the optimal command to allow the robotic arm to perform the ball throw, using both physical analysis and the ROS environment provided by OpenManipulator to control the robot. In order to properly execute this performance control, we need to consider the physical constraints of the robotic arm, as well as the necessary programming constraints.

III.2 – Hypotheses and constraints

III.2.A – Throw velocity maximization

This part deals with the calculation of the maximum speed that the robot can reach, as well as the angles of the different joints at which this maximum speed is reached. This calculation will allow us to find an ideal configuration of the robot to release the ball during the throw.

The first step is to convert the angular velocities of the joints of the robotic arm into Cartesian velocities at the end of the gripper. To do this, the Jacobian that establishes this link is determined.

III.2.A.a – Angular conventions

The OpenManipulator robot is designed with its own conventions, which are shown in red in Figure 22 below. For ease of calculation, other angles are defined which are shown in green in the same Figure.

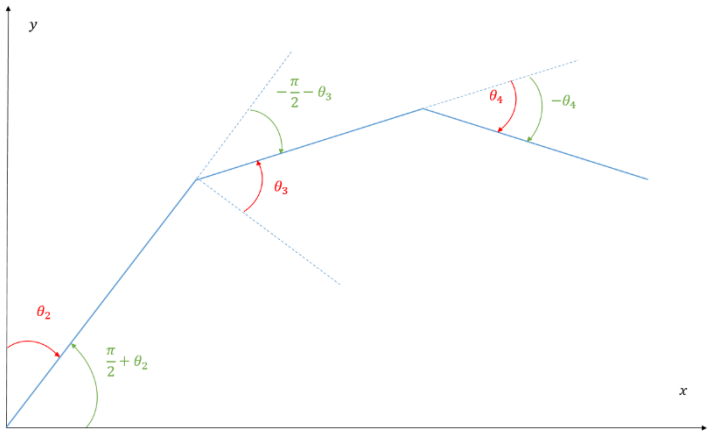


Figure 22: Angular conventions of the robot (red) and usual (green).

III.2.A.b – Determination of the Jacobian

In this section, we try to determine the Jacobian at the time of the robot's throw (which we write τ) which will maximize the norm of the ball speed. In particular, we note:

$$\alpha = \theta_2(\tau), \beta = \theta_3(\tau), \gamma = \theta_4(\tau)$$

In the Cartesian plane Oxz using trigonometric relations, we can write in the general case:

$$\begin{cases} x = l_2 \cos\left(\frac{\pi}{2}+\theta_2\right) + l_3 \cos\left(\frac{\pi}{2}+\theta_2+\left(-\frac{\pi}{2}-\theta_3\right)\right) + l_4 \cos\left(\frac{\pi}{2}+\theta_2+\left(-\frac{\pi}{2}-\theta_3\right)-\theta_4\right) \\ z = l_2 \sin\left(\frac{\pi}{2}+\theta_2\right) + l_4 \sin\left(\frac{\pi}{2}+\theta_2+\left(-\frac{\pi}{2}-\theta_3\right)\right) + l_4 \sin\left(\frac{\pi}{2}+\theta_2+\left(-\frac{\pi}{2}-\theta_3\right)-\theta_4\right) \end{cases}$$

Here, l_2, l_3 and l_4 are the respective lengths of the robot axes. These equations can be rewritten:

$$\begin{cases} x = -l_2\sin(\theta_2) + l_3\cos(\theta_2 - \theta_3) + l_4\cos(\theta_2 - \theta_3 - \theta_4) \\ z = l_2\cos(\theta_2) + l_4\sin(\theta_2 - \theta_3) + l_4\sin(\theta_2 - \theta_3 - \theta_4) \end{cases}$$

By time derivation, we obtain:

$$\begin{cases} \frac{\partial x}{\partial t} = -\frac{\partial \theta_2}{\partial t} l_2\cos(\theta_2) - \left(\frac{\partial \theta_2}{\partial t} - \frac{\partial \theta_3}{\partial t}\right) l_3\sin(\theta_2 - \theta_3) - \left(\frac{\partial \theta_2}{\partial t} - \frac{\partial \theta_3}{\partial t} - \frac{\partial \theta_4}{\partial t}\right) l_4\sin(\theta_2 - \theta_3 - \theta_4) \\ \frac{\partial z}{\partial t} = -\frac{\partial \theta_2}{\partial t} l_2\sin(\theta_2) + \left(\frac{\partial \theta_2}{\partial t} - \frac{\partial \theta_3}{\partial t}\right) l_4\cos(\theta_2 - \theta_3) + \left(\frac{\partial \theta_2}{\partial t} - \frac{\partial \theta_3}{\partial t} - \frac{\partial \theta_4}{\partial t}\right) l_4\cos(\theta_2 - \theta_3 - \theta_4) \end{cases}$$

And in matrix form:

$$\underbrace{\begin{bmatrix} \left.\frac{\partial x}{\partial t}\right|_{t=\tau} \\ \left.\frac{\partial z}{\partial t}\right|_{t=\tau} \end{bmatrix}}_{:=\boldsymbol{v}(\tau)} = \underbrace{\begin{bmatrix} -(l_2c_\alpha + l_3s_{\alpha\beta} + l_4s_{\alpha\beta\gamma}) & l_3s_{\alpha\beta} + l_4s_{\alpha\beta\gamma} & l_4s_{\alpha\beta\gamma} \\ -l_2s_\alpha + l_3c_{\alpha\beta} + l_4c_{\alpha\beta\gamma} & -(l_3c_{\alpha\beta} + l_4c_{\alpha\beta\gamma}) & -l_4c_{\alpha\beta\gamma} \end{bmatrix}}_{:=\boldsymbol{J}(\alpha,\beta,\gamma)} \underbrace{\begin{bmatrix} \left.\frac{\partial \theta_2}{\partial t}\right|_{t=\tau} \\ \left.\frac{\partial \theta_3}{\partial t}\right|_{t=\tau} \\ \left.\frac{\partial \theta_4}{\partial t}\right|_{t=\tau} \end{bmatrix}}_{:=\boldsymbol{\omega}(\tau)}$$

With:

$$c_\alpha = \cos(\alpha)\ ;\ c_{\alpha\beta} = \cos(\alpha - \beta)\ ;\ c_{\alpha\beta\gamma} = \cos(\alpha - \beta - \gamma)$$

$$s_\alpha = \sin(\alpha)\ ;\ s_{\alpha\beta} = \sin(\alpha - \beta)\ ;\ s_{\alpha\beta\gamma} = \sin(\alpha - \beta - \gamma)$$

Thus, the matrix

$$\boxed{\boldsymbol{J}(\alpha,\beta,\gamma) = \begin{bmatrix} -(l_2c_\alpha + l_3s_{\alpha\beta} + l_4s_{\alpha\beta\gamma}) & l_3s_{\alpha\beta} + l_4s_{\alpha\beta\gamma} & l_4s_{\alpha\beta\gamma} \\ -l_2s_\alpha + l_3c_{\alpha\beta} + l_4c_{\alpha\beta\gamma} & -(l_3c_{\alpha\beta} + l_4c_{\alpha\beta\gamma}) & -l_4c_{\alpha\beta\gamma} \end{bmatrix}}$$

is the Jacobian of the dynamical system (see file `Jacobian.m` in appendix for its definition in MATLAB).

III.2.A.c – Determination of optimal throwing angles and speed limits

We can therefore calculate the Euclidean norm of the speed of the ball at the instant of throw:

$$\boxed{\|\boldsymbol{v}(\tau)\|_2 = \|\boldsymbol{J}(\alpha,\beta,\gamma)\boldsymbol{\omega}(\tau)\|_2 \leq |\boldsymbol{J}(\alpha,\beta,\gamma)|_2 \times \|\boldsymbol{\omega}(\tau)\|_2}$$

Here, $|\cdot|_2$ is defined as the matrix norm induced by the Euclidean norm, which corresponds to the maximum singular value in absolute value:

$$|\cdot|_2 : \boldsymbol{N} \mapsto \max_{\lambda \in Sp(\boldsymbol{N}^T\boldsymbol{N})} \sqrt{|\lambda|}$$

Thus, in order to maximize the upper norm of the ball's throwing speed, we will try to maximize this induced norm $|\cdot|_2$ applied to the matrix $\boldsymbol{J}(\alpha,\beta,\gamma)$.

By calculation (see *Appendix 1*), it can be shown that the square matrix 3x3 $\boldsymbol{J}^T\boldsymbol{J}$ does not depend on the angle α: $\boldsymbol{J}^T\boldsymbol{J}(\alpha,\beta,\gamma) \to \boldsymbol{J}^T\boldsymbol{J}(\beta,\gamma)$. Thus, with the help of Matlab, we can draw the 3D graph of this induced norm $|\cdot|_2$ applied to the matrix $\boldsymbol{J}(\alpha,\beta,\gamma)$ as a function of β and γ only, on their domain of definition (see file `Calcul_of_max_speed2.m` in appendix): $\beta \in [0;\pi]$ and $\gamma \in [-\frac{\pi}{2};\frac{\pi}{2}]$.

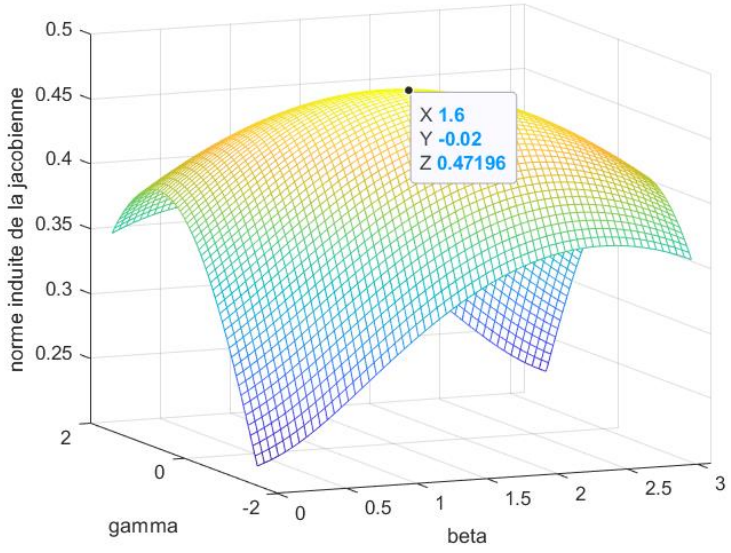


Figure 23: Induced norm graph $|\cdot|_2$ applied to the Jacobian $\boldsymbol{J}(\alpha, \beta, \gamma)$ as a function of β and γ.

We can see that the maximization of the induced norm is done at angles $\beta = \frac{\pi}{2}$ and $\gamma = 0$ which is consistent with the robotic arm reaching its most "extended" state, i.e. the distance between the ball and the robot anchor point is maximised. The rotational moment is then maximised, so the ball can reach the highest possible speed (provided that the speed motor $\omega_2 := \frac{\partial \theta_2}{\partial t}$ can withstand these dynamic constraints). An upper bound on the speed limit is therefore:

$$\frac{\|\boldsymbol{v}(\tau)\|_2}{\|\boldsymbol{\omega}(\tau)\|_2} \leq \max_{\substack{\beta \in [0;\pi] \\ \gamma \in \left[-\frac{\pi}{2};\frac{\pi}{2}\right]}} |\boldsymbol{J}(\alpha, \beta, \gamma)|_2 \approx 0{,}472\ m$$

Experimentally, we find that the motors of the OpenManipulator robot are bounded in velocity by $4{,}8\ rad/s$, so the velocity at the instant of throw is upper bounded by:

$$\boxed{\|\boldsymbol{v}(\tau)\|_2 \leq 3.92\ m/s}$$

III.2.B – Friction model during the flight

To reach a target such as the cup, a model of the dynamics of the ping-pong ball during its flight must be established, in order to determine the norm v_0 and the direction α_0 of the velocity of the robot's gripper at the moment of release. Depending on the model chosen, the velocity will not have the same norm for a given angle.

We can take into account or neglect different forces that would act on the ball to determine the dynamic model we will use for the throw. The list of forces that can be taken into account during the flight:

$$\begin{cases} \overrightarrow{\boldsymbol{f}_{\boldsymbol{frottement\ air}}} \\ \vec{\boldsymbol{P}} \\ \overrightarrow{\boldsymbol{f}_{\boldsymbol{Coriolis}}} \\ \overrightarrow{\boldsymbol{f}_{\boldsymbol{Magnus}}} \\ \overrightarrow{\boldsymbol{\pi}_{\boldsymbol{Archimède}}} \end{cases}$$

We are talking about a flight time of the order of a second and a distance of the order of a meter, so the Coriolis force is negligible. The buoyancy is negligible compared to the weight, and the clamp make the ball spin, so the Magnus effect is non-existent (the force is zero).

III.2.B.a – Frictional force in v2

This leaves weight and friction to be taken into account in the balance of forces in Newton's second law. For fluid friction on a smooth sphere, the force is expressed as:

$$\overrightarrow{f_{frottement\ air}} = -\frac{1}{2}\, C_x\, S\, \rho\, v\, \vec{v}$$

with $\begin{cases} C_x = 0.45 \text{ the aerodynamic resistance coefficient for a smooth sphere} \\ S \text{ the frontal area} \\ \rho \text{ the density of the air} \\ v \text{ the velocity of the ball} \end{cases}$

This gives the equations of motion in the Cartesian reference frame:

$$\begin{cases} m \cdot \ddot{x} = -k \cdot \dot{x}^2 \\ m \cdot \ddot{y} = -g - k \cdot \dot{y}^2 \end{cases} \quad with\ k = \frac{1}{2}\, C_x\, S\, \rho$$

This system of differential equations must be solved analytically with the parametric initial values, including v_0 the norm of the ball/clamp velocity at release. Then, thanks to the analytical expression of the trajectory, we solve the initial condition v_0 for the cup distance and the relative height of the release point. These equations are non-linear and, in fact, analytically intractable. So unless one constructs a numerical solution for each value of a list of values of v_0 , and looks at which one arrives closest to the position of the cup, one cannot trace back to v_0 from the landing position of the ball. In addition, this solution requires a fairly large computational resource for a system acting in real time. And it would still be necessary to intelligently approach the value of v_0 to be entered to reach the target, in order to reach a certain precision (if we ignore this precision, we can directly take a simplified model).

We therefore abandon this model of friction, which is fairly accurate but not adapted to our use.

III.2.B.b – Frictional force in v

A linearized friction model can be used:

$$\begin{cases} m \cdot \ddot{x} = -k \cdot \dot{x} \\ m \cdot \ddot{y} = -g - k \cdot \dot{y} \end{cases} \quad avec\ k = C_x\, S\, \rho$$

Thanks to this model, we can find the analytical expression of the solutions:

$$x(t) = \frac{m}{k} \cdot v_0 \cdot cos(\alpha_0) \cdot (1 - exp(-\frac{k \cdot t}{m}))$$

$$y(t) = \frac{m}{k} \cdot (v_0 \cdot sin(\alpha_0) + \frac{m \cdot g}{k}) \cdot (1 - exp(-\frac{k \cdot t}{m})) - \frac{mg \cdot t}{k} + H$$

We can then express the time as a function of x in the first equation and inject into the second, finally recovering the trajectory equation y(x) as a function of v_0, then isolating v_0 . The solve function in python returns a negative norm, so there must be a problem somewhere. Solving by hand returned another absurd result, despite multiple verification and then third-party verification. But this discussion can be cut short, as we will see in the next section.

III.2.B.c – Frictionless model

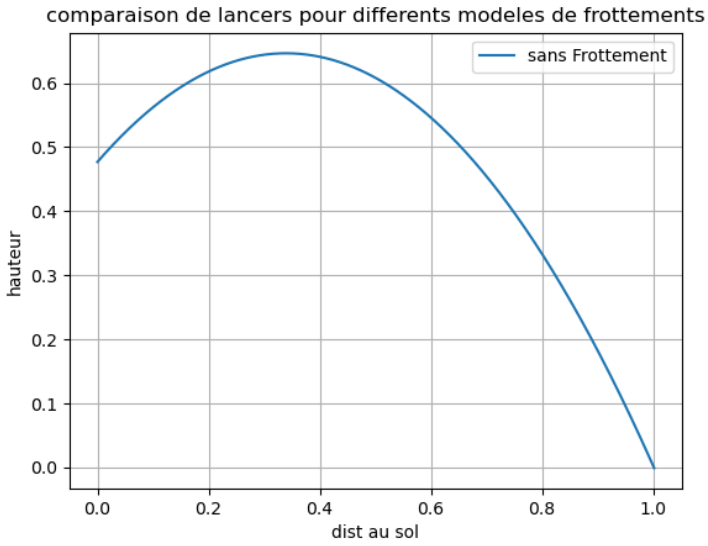
In view of these difficulties, it was preferable to have a functional code to implement in the robot first and to perfect it later. We therefore established an expression for v_0 based on a frictionless model:

$$\begin{cases} m \cdot \ddot{x} = 0 \\ m \cdot \ddot{y} = -g \end{cases}$$

After a manual resolution we find the explicit expression:

$$v_0 = \sqrt{\frac{\frac{g}{2} \cdot (\frac{D}{sin(\alpha_0)})^2}{\frac{D \cdot cos(\alpha_0)}{sin(\alpha_0)} + H - h}}$$

with $\begin{cases} D \text{ the distance on the ground between the release and the cup} \\ H \text{ the height of the release} \\ h \text{ the height of the cup} \end{cases}$



This model gives plausible orders of magnitude: for an 8cm high cup, 1m from the release point, itself 47.7cm from the ground and a launch angle of 45°, we find a standard release speed of 2.6499415857745205 m/s. And when we inject this value into the numerical trajectory simulator (built following Euler's method thanks to the odeint function of python), we obtain the following trajectory:

Figure 24: Throw trajectory for a 45° angle and a norm of 2.6499415857745205 m/s.

As for the model refinements, it is good to judge their usefulness before embarking on time-consuming debugging as announced above in the simple V model. Therefore, for the different friction models, the landing distances are compared to quantify the influence of the friction shape on the prediction performance.

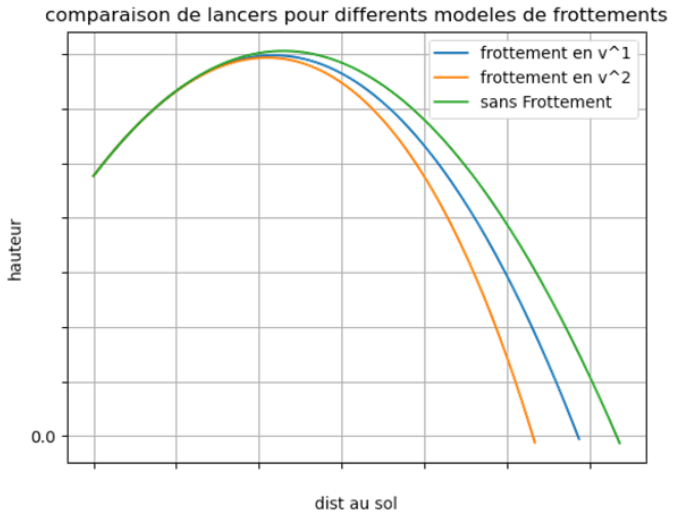


Figure 25: Zooming in on the trajectories for the different friction models.

It can already be seen on this zoom that the trajectory resulting from the v-model is between that of the v^2-model and that without friction.

But we can see that when we quantitatively compare the trajectory for the model without friction and with friction in v^2, there is almost no difference:

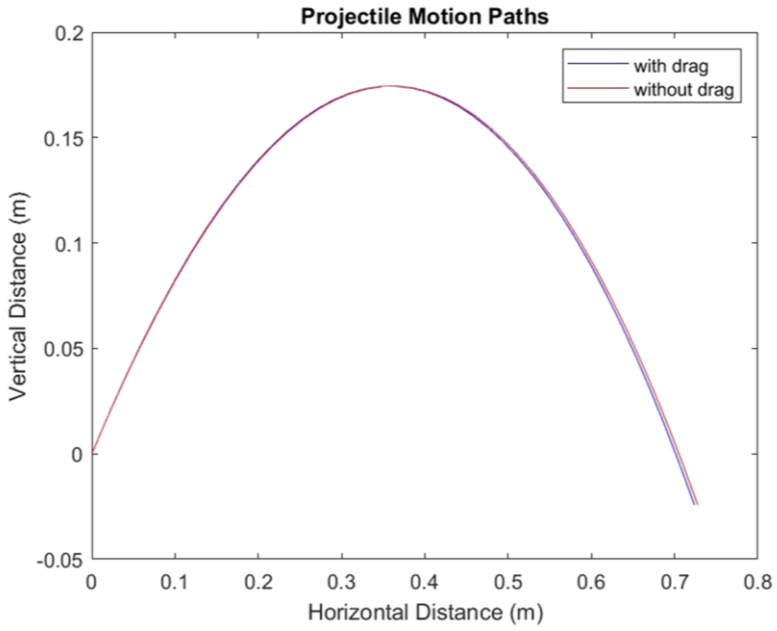


Figure 26: Trajectories for the models with friction in v^2 and without friction

In particular, compared to the uncertainties of the throw due to the opening of the clamp for example, the few millimetres to be gained with the more accurate model for friction are negligible.

And indeed, the formula for determining v_0 thanks to the frictionless model, which was finally implemented in the robot's control code, makes it possible to hit the cup with sufficient precision for the ball to enter it cleanly every time. Beyond one metre, the dynamics of the robot becomes limiting for the throw as the motors approach saturation, so despite the difficulties encountered and partially overcome to take into account the friction, the basic model without drag is more than sufficient to satisfy the specifications, thus allowing a fair and accurate throw.

III.3 – Performing a trajectory using OpenManipulator

One of the main problems that arose during this project was to be able to impose a movement on the robotic arm along a desired trajectory. We started this project with the progress of the first-year project on the unrooting task. We were therefore able to move the robotic arm into desired positions and define a sequence of movements for the arm.

We used the services offered by the ROS2 controller package provided by the manufacturer, which allows different actions, but we chose to control the angular position, requiring an inverse geometric model that we had already developed in the first year. These inverse geometrical models also allow us to define constraints on certain angles, which can offer more control over the movement. When we ask the robot to move, we also need to give a time to perform the movement.

However, performing a dynamic trajectory is quite different from simply making a succession of movements at the different points to be reached. Indeed, when we ask the arm to move, it tries to arrive at the final point in the time we ask but with a zero speed at the arrival. Thus, if we simply give a succession of points situated on the trajectory, we obtain a jerky movement totally different from what is desired, and the speed of the ball at the moment of reaching the point of release will not correspond at all to the desired speed.

So, we had to find a solution to be able to finally give the robot a trajectory so that the ball could reach a desired speed at a desired point.

A second problem with the robot was the ability to synchronize the opening of the gripper with the trajectory and whether the opening was fast enough to release the ball during a movement.

III.3.A – Study of the operation of the arm

To solve all these questions the first step was to study the exact functioning of the arm from a dynamic point of view, by studying the documentation and making experimental observations to check the consistency.

- Documentation :

The motors that make up the robotic arm are Dynamixel-XM430-W350 motors, the documentation for which is available online at https://emanual.robotis.com/docs/en/dxl/x/xm430-w350/.

Item	Specifications
MCU	ARM CORTEX-M3 (72 [MHz], 32Bit)
Position Sensor	Contactless absolute encoder (12Bit, 360 [°]) Maker : ams(www.ams.com), Part No : AS5045
Motor	Coreless
Baud Rate	9,600 [bps] ~ 4.5 [Mbps]
Control Algorithm	PID control
Resolution	4096 [pulse/rev]
Backlash	15 [arcmin] (0.25 [°])
Operating Modes	Current Control Mode Velocity Control Mode Position Control Mode (0 ~ 360 [°]) Extended Position Control Mode (Multi-turn) Current-based Position Control Mode PWM Control Mode (Voltage Control Mode)
Weight	82 [g]
Dimensions (W x H x D)	28.5 x 46.5 x 34 [mm]
Gear Ratio	353.5 : 1
Stall Torque	3.8 [N.m] (at 11.1 [V], 2.1 [A]) **4.1 [N.m] (at 12.0 [V], 2.3 [A])** 4.8 [N.m] (at 14.8 [V], 2.7 [A])
No Load Speed	43 [rev/min] (at 11.1 [V]) **46 [rev/min] (at 12.0 [V])** 57 [rev/min] (at 14.8 [V])
Radial Load	40 [N] (10 [mm] away from the horn)
Axial Load	20 [N]
Operating Temperature	-5 ~ +80 [°C]
Input Voltage	10.0 ~ 14.8 [V] (**Recommended : 12.0 [V]**)
Command Signal	Digital Packet
Physical Connection	RS485 / TTL Multidrop Bus TTL Half Duplex Asynchronous Serial Communication with 8bit, 1stop, No Parity RS485 Asynchronous Serial Communication with 8bit, 1stop, No Parity
ID	253 ID (0 ~ 252)
Feedback	Position, Velocity, Current, Realtime tick, Trajectory, Temperature, Input Voltage, etc
Case Material	Metal (Front, Middle), Engineering Plastic (Back)
Gear Material	Full Metal Gear
Standby Current	40 [mA]

Figure 27: Specifications of the motors.

This documentation gives the specifications of the motors where it is stated that the maximum speed that can be reached is 46 rev/min (4.8 rad/s) and also that several operating modes are possible.

By searching the robot documentation, we can see that to use the ROS2 controller provided by the manufacturer, the motors must be in the position control mode. We then know that the motors are controlled by a PID as described below:

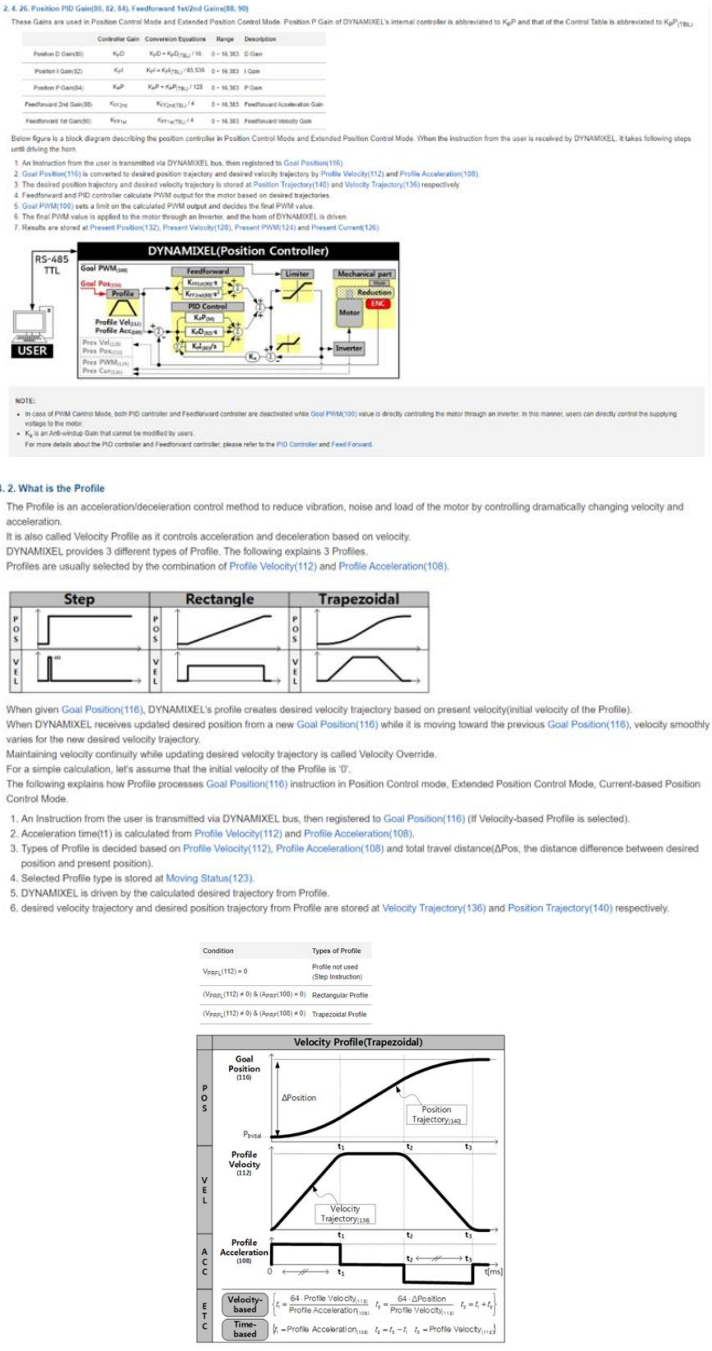


Figure 28: Commands of the motors.

Looking at the controller code, we found that the values for "*profile_velocity*" and "*profile_acceleration*" are set to 0, which means that the step speed profile is used, the motor tries to reach the target as quickly as possible.

- Experimental verification:

To check the real operation of the motors, a ROS2 node has been created which allows the state of the motors to be recorded at the same time as the robot is asked to move, by sending a message to a topic when the recording must start.

To test the reaction of a motor, we looked at the speed at the different motors when we ask the last section of the arm to make a half turn as quickly as possible. The following graph is obtained:

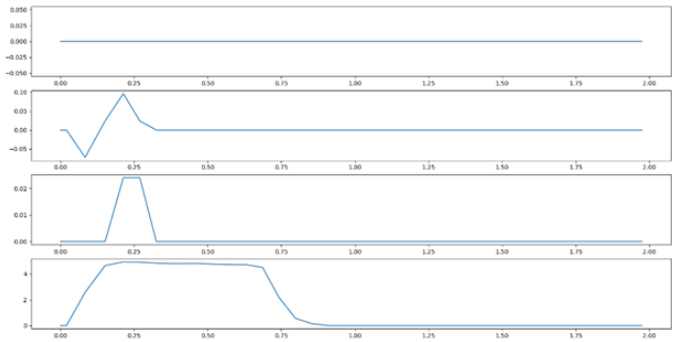


Figure 29: Speed of the motors when saturating.

The first three motors have speeds that can be considered zero, which is normal because their displacement corresponds to the force exerted by the clamp when moving. The fourth motor, which is the one to which we ask a movement, has a speed where we can see a phase of acceleration, a plateau at 4.8 rad/s which corresponds to the maximum value given in the documentation then a phase of deceleration. If we look at what happens for a slower movement, we get:

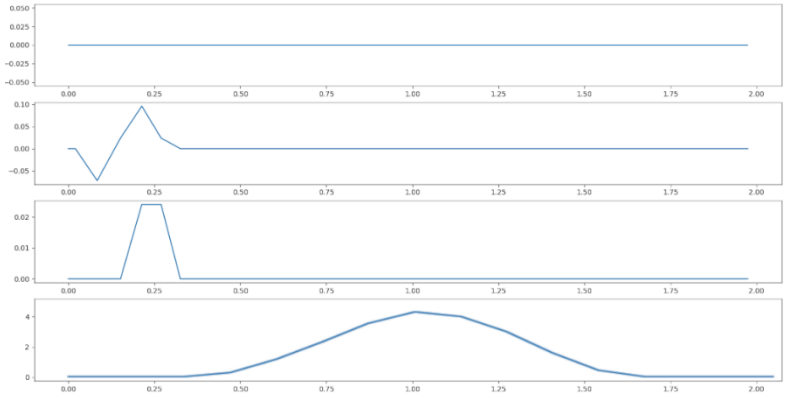


Figure 30 Speed of the motors without saturation

We can see that the speed of the motor describes a smooth curve, we understand by looking at the code that this curve is obtained by generating a trajectory between the point of origin and arrival by a polynomial interpolation of degree 5 (Hermite interpolation).

III.3.B – Explanation of the controller provided by the manufacturer:

Generating a trajectory with the robotic arm cannot be done simply by requesting successive movements of the arm to points on the desired trajectory. Indeed, by doing so, we do not control the movement between the points. In particular, the speed of the arm will not be constant between the point of origin and the point of arrival, so the movement will be jerky and unusable for throwing a ball.

One could use simple movement requests, finding the movements to be requested so that the speed of the arm follows the desired speed, considering the way the arm proceeds. This is complicated at first and does not give any assurance of success. It would be like using a positional servo to make a speed servo, so it would be better to change the mode of operation of the motors to allow for current or speed control. This would require the redesign of a controller, which is a lot of work.

We therefore chose to look for other ways of implementing a trajectory which are already allowed by the controller provided by the constructor. We therefore had to look at the source code of the controller, whose structure we will describe.

III.3.B.a – Organization of the source code of the controller node

The source code provided by the manufacturer is organised in 6 folders and provides different control methods on the robot, thanks to the definition of several ROS2 services included inside a ROS2 node:

Folders:

- ***"Dynamixel-workbench***: provides different methods to communicate with dynamixel engines.
- ***"DynamixelSDK"***: *software development kit* for dynamixel engines.
- ***"robotis_manipulator":*** In this folder are defined several classes and methods allowing to make a controller for any type of robot using **dynamixel motors**, this package is based on the two previous ones.
- ***"open_manipulator_dependencies"***: defines the dependencies between the different packages useful for the robot's operation, there is nothing to know in it to use the robot.
- ***"open_manipulator_msg"***: Defines the ROS2 messages and services used by the controller.
- ***"open_manipulator"***: controller specific to the ***OpenManipulatorX*** robot which defines the virtual classes of the **"robotis_manipulator"** package. It defines the ROS2 services accessible for the use of the robot, an inverse geometrical model, the script ***"teleop_keyboard.py"*** showing an example of use of the controller with python, defines 4 basic custom trajectories and all that is necessary to use the robot, it is this file that must be modified if we want to change the functioning of our robot.

To understand how the robot works, we need to look more closely at the contents of the folder ***"open_manipulator"***, which is the folder specific to the arm used and which must be modified if we want to add possibilities, and ***"robotis_manipulator"***, which is regularly called up by the former.

Folder Open_manipulator:

This folder contains four subfolders, ***"open_manipulator_x_description"*** which defines the gazebo and rviz simulations that we are not interested in here, the ***"open_manipulator_x_teleop"*** folder which is an example of a python package using the controller's services, and the two central folders ***"open_manipulator_x_controler"*** and ***"open_manipulator_x_libs"*** which we will detail later.

- **"open_manipulator_x_controller"**: This folder contains only the main program that must be launched to use the robot, **"open_manipulator_x_controller.cpp"**. This one defines the **ROS2 node** and all the services that will be used by the packages that we will implement later to control the robot, with the C++ class OpenManipulatorXController class.
- **"open_manipulator_x_libs"**: This folder contains the classes used directly by the controller node, which are mostly implementations of virtual classes defined in the ***"robotis_manipulator"*** package. This folder is composed of four C++ files:
    - **"open_maniulator_x.cpp"**: Defines the ***OpenManipulatorX*** class which implements the ***RobotisManiualtor*** class from the "***robotis_manipulator"*** package. This class contained in the ***OpenManipulatorXController*** class defines the geometry of the robot and the parameters. The class has four attributes (**kinematics_**, **actuator_**, **tool_** and **custom_task_trajectory**) defined in the other files of the package.

- **"kinematics.cpp"**: Defines three different inverse geometric models using different solving methods, ***SolverUsingCRAndSRPositionOnlyJacobian***, ***SolverUsingCRAndSRJacobian*** and ***SolverCustomizedForOMChain, the*** latter is used by default in the program. The solver used can be changed in the file ***open_manipulator_x.cpp*** (l. 124).
- **"dynamixel.cpp"**: Definition of three classes (***JointDynamixel***, ***JointDynamixelProfileControl*** and ***GripperDynamixel***) which implement virtual classes of robotis_manipulator and are used for communication with dynamixel engines. The code is based on the ***dynamixel_workbench*** files.
- **"custom_trajectory.cpp"**: Allows to define trajectories as a function, this is one of the possibilities we are trying for our project. There are four trajectories already implemented (**line**, **circle**, **rhombus** and **heart**), but we have only managed to use the **line** function, due to some problems such as the inverse geometric model used by default which does not always manage to converge.

Folder Robotis_manipulator:

- ***"robotis_maipulator.cpp"***: Defines the ***RobotisManipulator*** class which is the parent class of ***OpenManipulator***, the file provides templates of functions to make its own controller.
- **"robotis_manipulator_common.cpp"**: Defines some of the classes used in the other files, notably the ***"enum"*** and ***"struct"*** classes, such as ***JointWaypoint*** or ***TaskWaypoint***. The file also defines a ***Manipulator*** class which is the **manipulator_** attribute of the ***RobotisManipulator*** class.
- ***"robotis_manipulator_manager.cpp***": Defines the ***Kinematics***, ***JointActuator***, ***ToolActuator***, ***CustomJointTrajectory*** and ***CustomTaskTrajectory*** parent classes that are included in ***"kinematics.cpp"*** and ***"dynamixel.cpp"*** of ***"open_manipulator"***.
- ***"robotis_manipulator_trajectory_generator.cpp"***: Defines the ***MinimumJerk***, ***JointTrajectory***, ***TaskTrajectory***, ***Trajectory*** classes allowing the calculation of the trajectory of the different services provided by the source code.
- ***"robotis_manipulator_math.cpp"***: Defines useful mathematical objects such as **vectors**, **inertia matrices**, **quaternions**, and the **mathematical methods** for some generic calculations from these objects.
- ***"robotis_manipulator_log.cpp"***: Defines tools to display messages on the terminal and in log files.

By looking at these different files we can understand how the source code works.

III.3.B.b – Operation of the services used in this project

The list of services provided by the source code is available at the following link: https://wiki.ros.org/open_manipulator_controller. We will simply detail the operation of the two services we use ***"goal_joint_space_path"*** and ***"goal_drawing_trajectory"***.

Service goal_joint_space_path :
This service allows the robot to be controlled by providing the angular positions to be given to the four motors.

1. ***goal_joint_space_path_callback*** (OpenManipulatorXController): call ***makeJointTrajectory.***
2. ***makeJointTrajectory*** (RobotisManipulator): sets the trajectory type as **JOINT_TRAJECTORY** and calls another makeJointTrajectory located in the Trajectory class.
3. ***makeJointTrajectory*** (Trajectory): uses the ***MinimumJerk*** class to calculate the coefficients of the Hermite interpolation. The coefficients define a polynomial trajectory for each of the motors between their current position and the desired arrival position. They are stored in the **minimum_jerk_coefficient** variable of the ***Trajectory*** Class (**trajectory_** variable in the other files).

The rest of the operation is part of the process called in a loop by the node, which performs the following steps:

4. ***process_callback*** (OpenManipulatorXController): uses the node's clock to find out the time and calls ***process_open_maipulator_x*** passing the **current time** as an argument.
5. ***process_open_maipulator_x*** (OpenManipulatorX): calls the function getJointGoalValueFromTrajectory passing the **current time as** argument.
6. ***getJointGoalValueFromTrajectory*** (RobotisManipulator): uses **trajectory_** to initialize the path with ***initTrajectoryWayPoint*** if it has not been done before and to calculate the **joint_goal_way_point** variable using ***getTrajectoryJointValue***. Then send the obtained position values using ***sendAllJointActuatorValue***.
7. ***initTrajectoryWayPoin***t (Trajectory): uses the **manipulator_** attribute to initialise the angular positions of the motors to their real state (surely to keep the robot in a stable position when switched on.
8. ***getTrajectoryJointValue*** (RobotisManipulator): chooses the appropriate protocol for the type of trajectory that is requested in our case is JOINT_TRAJECTORY. Calls the ***getJointWayPoint*** function of the ***JointActuator*** attribute of **trajectory_** and checks that the result does not exceed the limits of the engine, in which case the trajectory is interrupted with the ***removeWayPointDynamicData*** function.
9. ***getJointWayPoint*** (JointTrajectory): calculates the angular position to give to the motors using the coefficients calculated earlier (simple polynomial evaluation).
10. ***sendAllJointActuatorValue*** (RobotisManipulator): sends the desired positions to the motors by calling functions from the ***"dynamixel-workbench"*** folder.

The saturation of the motor at 4.8 rad/s which appears cannot really be explained by the algorithm, perhaps the power supply to the motors saturates and the position control of the motors plays its role, which allows the correct position to be reached but with a certain delay.

Service goal_drawing_trajectory:
This service allows the use of custom trajectories, the initial code provides 4 trajectories: line, circle, rhombus and heart.

1. ***goal_drawing_trajectory_callback*** (OpenManipulatorXController): takes the parameters provided in the message sent to the service and formats them according to the type of trajectory requested. Then call ***makeCustomTrajectory***.
2. ***makeCustomTrajectory*** (RobotisManipulator): sets the path type as **CUSTOM_TASK_TRAJECTORY** (resp. **CUSTOM_JOINT_TRAJECTORY** if the requested trajectory is an angular trajectory, i.e. the function used to describe it describes the angular positions of the motors as a function of time and not the Cartesian positions reached by the gripper), then calls another ***makeCustomTrajectory*** located in the ***Trajectory*** class by passing a ***TaskWayoint*** (resp. ***JointWaypoint***) as argument.
3. ***makeCustomTrajectory*** (Trajectory): uses the **cus_joint_** attribute to call a ***makeTaskTrajectory*** (resp. ***makeJointTrajectory***) function specific to the type of trajectory requested. **cus_joint_** contains information about the available custom trajectories, associating the names of the trajectories with the corresponding class in the ***"custom_trajectory.cpp"*** file.
4. ***makeTaskTrajectory*** (Line, Curve, ...): launches the initialization function specific to the desired trajectory.
5. ***init_curve*** (Line, Curve, ...): calculates the variables needed to define the desired trajectory and depending on the parameters passed to the service or the initial position of the robot.

The rest of the operation is again part of the process called looping by the node, the first 4 steps are similar to what is described above.

- ***getTrajectoryJointValue*** (RobotisManipulator): selects the appropriate protocol for the type of trajectory that is requested in our case it is **CUSTOM_TASK_TRAJECTORY**. Calls the ***getTaskWaypoint*** (resp. ***getJointWaypoint***) function of the class corresponding to the desired trajectory using the information contained in the trajectory_ attribute and checks that the result does not exceed the **limits of the engine**, in which case the trajectory is interrupted with the ***removeWayPointDynamicData*** function.
- ***getTaskWaypoint*** (Line, Curve, ...): (resp. ***getJointWaypoint***) returns the Cartesian position to which the robot must go at **the present time in the** form of a ***TaskWaypoint***. This **TaskWaypoint is** converted into a **JointWaypoint** by the inverse geometrical model given in **kinematics_** (it is usually at this stage that the program stops because it cannot find the solution). In the case of an angular trajectory, the conversion is not necessary as the function returns a ***JointWaypoint*** directly.

III.3.C – Trajectory calculation by polynomial interpolation

This part deals with the calculation of the trajectory taken by the robot arm according to the kinematic and physical constraints of the robot. Now that we know the kinematic limits of the robot, we will try to give a control trajectory at the robot's input.

It should be noted that in the application case of this robotic arm, whose function is to throw a ball at a given angle and speed to aim at a target, the general shape of the trajectory does not fundamentally matter to us, as long as it respects the constraints for a correct throw. Thus, for reasons of computational ease, we propose trajectory generation by Hermite polynomial interpolation. Similar to the Lagrangian polynomial interpolation technique, we will generate a polynomial of degree n from $n + 1$ constraints.

The trajectory will be divided into two phases: an acceleration phase, before the ball is thrown, to reach the imposed speed and throwing position and then release the ball; a deceleration phase, after the ball is thrown, to bring the robot back to a static state while respecting the physical constraints of the robot and the continuity of the physical quantities. Thus, in order to control the variation of position, speed and acceleration, the following constraints will be imposed:

- In the acceleration phase: an initial x_{01} and final x_{f1}, an initial and final velocity $v_{01} = 0$ and final v_{f1}, an acceleration $a_{01} = 0$ and final a_{f1}.
- In the deceleration phase: an initial position $x_{02} = x_{f1}$, an initial speed $v_{02} = v_{f1}$ and final $v_{f2} = 0$, an initial and final acceleration $a_{02} = a_{f1}$ and final $a_{f2} = 0$ and an initial jerk (derivative of the acceleration) $j_{02} = j_{f1}$.

The details of the calculations will not be developed here, the file `Interpolation_Hermite_v3.m` contains the expressions of the various coefficients of the generated polynomials and makes it possible to draw the various curves in position, speed and acceleration according to the chosen parameters. It should be noted that, in the following example, the chosen parameters are not necessarily applicable to the real system: they only serve to illustrate the functioning of the method and the considerations that were made in its construction.

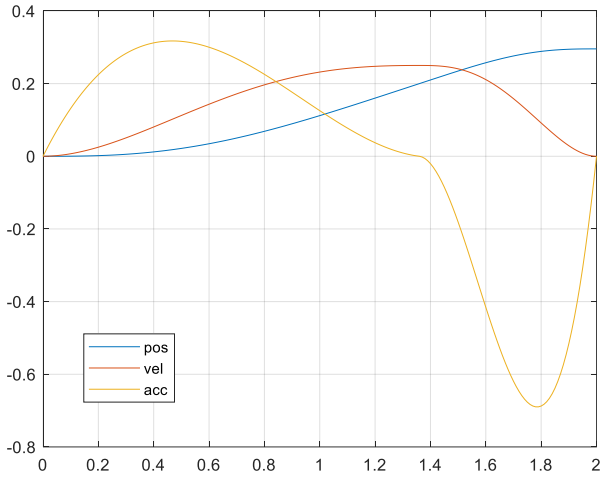


Figure 31: Plot of position (in x), velocity and acceleration according to the set of parameters defined in the file `Interpolation_Hermite_v3.m` (see Appendix 2).

It may be noted that classical robot trajectory commands are "trapezoidal" speed commands, i.e. composed of an affine function increasing over a first period τ_1, a constant function over a second period τ_2, then the symmetrical of the first affine function on a third time period τ_1 also.

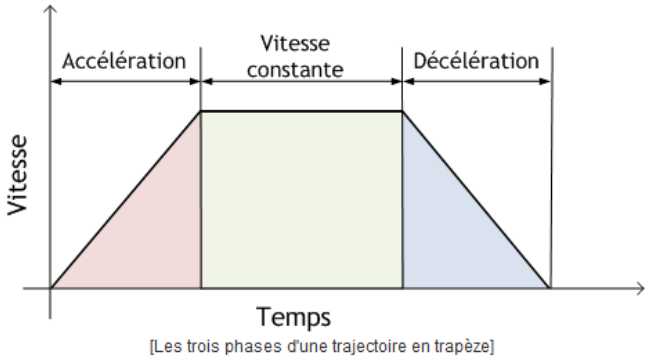


Figure 32: Principle of trapezoidal speed control.

Here we can see that with Hermite interpolation, in addition to respecting our control and dynamic constraints, the velocity curve also closely adopts a "trapezoidal control" curve shape. Thus, we will use this method to generate the trajectories that the robot arm will follow during its application.

III.3.D – Implemented trajectories

III.3.D.a - Implementation

Two types of trajectories were implemented directly in the source code during this project. The first one called ***"courbe"*** theoretically allows to throw the ball with a speed having both a vertical and a horizontal component. The second one called ***"depliage"*** throws the ball horizontally, when the arm is at maximum extension in the vertical direction, this theoretically allows to reach the highest possible speeds but requires more effort from the motors.

Two types of ties were implemented direct in the source code during this project. The first one called ***"courbe"*** theoretically allows to throw the ball with a speed having both a vertical and a horizontal coonent. The second one cald ***"depliage"*** throws the ball horizontally, when the ar is at maximum xtesion in h vertical irection, this theoretically allows to reach the highest possible speeds but requires more effort from the motors.

Both trajectories can be requested with a python program using the ROS2 service ***"goal_drawing_trajectory"***.

- "courbe":

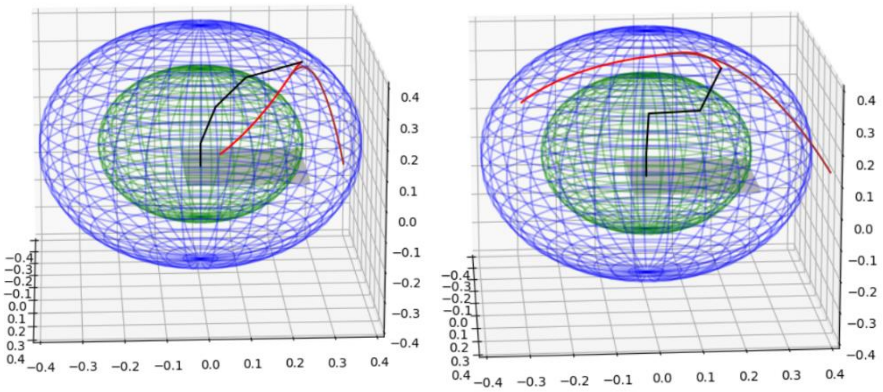


Figure 33 Visualisation of the trajectory

This trajectory includes an acceleration phase that ends when the ball is released and then a deceleration phase to stop the movement of the arm. The trajectory of these two phases is calculated using the Hermite interpolations that were presented earlier and allow the movement to be smoothed.

It is set up with 8 parameters:

- the acceleration time

- deceleration time

- the vertical speed at the moment of release

- the horizontal speed at the moment of release

- the height of the release point

- the distance from the release point to the robot's origin

- the height of the point where the clamp stops

- the distance from the point where the gripper stops to the robot origin

The release velocities depend on the point of launch and the position of the cup, so it can be calculated if you want to hit a target. The time of the acceleration and deceleration phases will have an impact on the trajectory obtained, they can be chosen by trying to minimize the maximum speeds reached by the motors during the trajectory.

However, the key points of the trajectory must be chosen. The release and stop points of the gripper can theoretically be chosen freely in the space reachable by the robot. However, the speeds required for the motors to reach will depend greatly on these parameters. It is thus difficult to launch the ball with a speed higher than 1 m/s with such a trajectory.

In addition, the controller transforms the Cartesian trajectory into an angular trajectory using an inverse geometric model, coded in the source code in C++. In order to get this trajectory to work on the robot, we had to replace it with the one we had developed in last year's project.

So we have not implemented anything that would allow us to calculate the parameters to be given to this type of trajectory depending on the position of a cup. This could certainly be done with the help of an AI that would find the best possible set of parameters for a particular target to hit. This work can be considered in the future.

- "depliage":

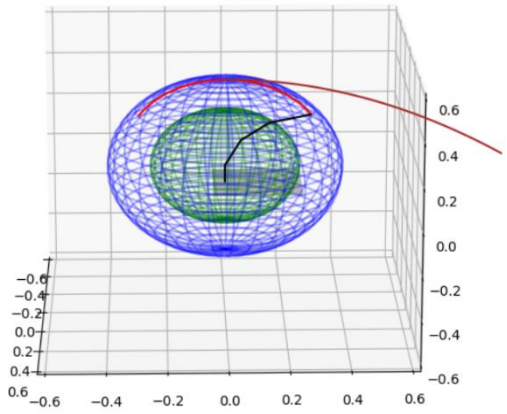


Figure 34 Visualisation of the trajectory

This trajectory is described directly using the angular positions of the motors. The principle is to overcome the various problems of the first trajectory (dependence on the inverse geometric model, non-complementary use of the different motors). For this purpose, in this trajectory, the different sections of the arm have at each moment an equal angle relative to the previous section. This relative angle is varied from -α to +α (for α up to 30°).

The variation of α over time is calculated by Hermite interpolation, to obtain smooth variations. This trajectory can also be requested from a python script using "goal_drawing_trajectory".

The parameters of this trajectory are:

- the angle α, of amplitude
- the execution time of the trajectory
- the speed to be reached for the throw.

With this trajectory, the ball can be thrown with a horizontal velocity of up to 3.5m, allowing a throw of up to about 1m. Given a position to be reached, the speed to be given to the ball at the moment of release is calculated with the frictionless free fall model presented earlier. The execution time of the trajectory is calculated by simulating the trajectory produced with different execution times, trying to maximize this time without the velocity being negative at any point in the trajectory.

We can therefore perform a throw by choosing the amplitude α (we use 25°), and simply giving the position of the cup to be reached as a parameter. This trajectory is the only functional one currently available for reaching a cup.

III.3.D – Implemented trajectory

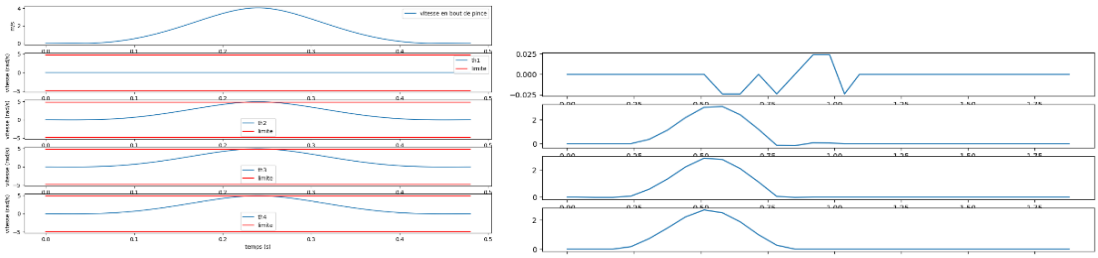


Figure 35: Theoretical speed of the motors (left) and measured speed (right)

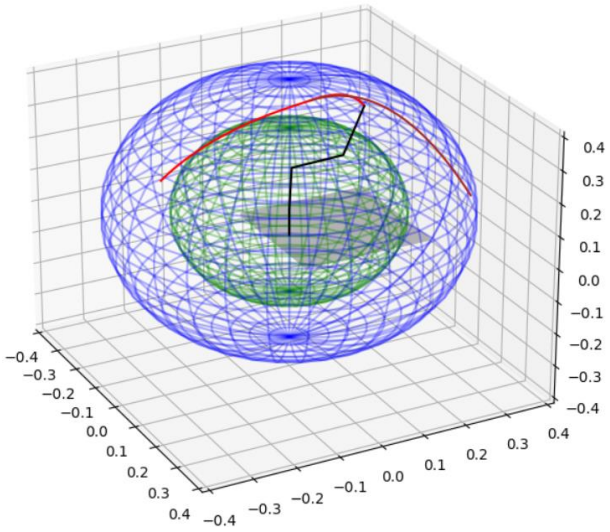
We can see that measured speed from experiences is really close to theorical one. Thus, we can consider that the robot is able to perform this kind of trajectory well for the throw.

III.3.D.b – Visual simulation

We developed python simulations of the two custom trajectories, allowing to visualise the spatial path of the robot and the movement of the ball in the air. This makes it possible to check that the trajectories generated with the "curve" function remain within the range of possible robot movements before asking the robot to do so, and thus to avoid collisions with the support.

The simulation of the "courbe" trajectory compares two inverse geometric models: the first uses only trigonometry calculations but requires the orientation of the arm to be fixed with respect to the horizontal, the second uses a gradient descent algorithm with a Jacobian matrix linking angular position and Cartesian position of the arm.

Here are the kind of results shown by these simulations:



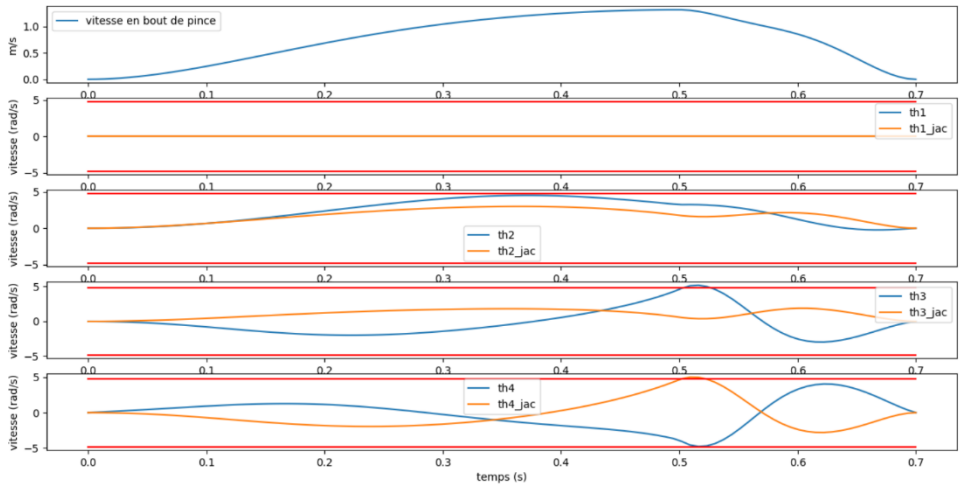


Figure 36: Results of simulations.

The blue and green spheres represent respectively the space accessible by the gripper and the space accessible by the second section of the arm. The red line is the trajectory of the gripper, the brown line is the free fall of the ball in the air.

III.3.E – Writing a new trajectory

To write a new trajectory using ***custom_trajectory***, there are two options:

III.3.E.a – Option 1: Cartesian trajectory

The Line, Circle, Rhombus, Heart and Curve trajectories are examples.

"open_manipulator_x_controller.cpp":

Add an "*else if*" to ***goal_drawing_trajectory_callback*** with the **name of the** desired **path** and the **number of parameters** to pass to the function.

Call the ***makeCustomTrajectory*** function with the argument ***req→end_effector_name***, this is what will define the trajectory as Cartesian.

"custom_trajectory.cpp/.hpp":

Add a class corresponding to the new path (copy the functions present in an example) taking care that the functions use ***TaskWaypoint***. Don't forget to make consistent changes between the .cpp and the .hpp.

"open_manipulator_x .cpp/.hpp":

In the .hpp, change the **CUSTOM_TRAJECTORY_SIZE** and add a line to define its new path as those already present.

In the .cpp add the lines to define the new trajectory in the ***Initialize Custom Trajectory*** **section** at the end of the file, following the example of the trajectories already defined.

III.3.E.b – Option 2: Angular trajectory

The Depliage trajectory is an example.

"open_manipulator_x_controller.cpp":

Add an "*else if*" to ***goal_drawing_trajectory_callback*** with the **name of the** desired **path** and the **number of parameters** to pass to the function.

Call the ***makeCustomTrajectory*** function without passing **req→end_effector_name** as an argument, this is what will define the trajectory as angular.

"custom_trajectory.cpp/.hpp":

Add a class corresponding to the new path (copy the functions present in an example) taking care that the functions use and return ***JointWaypoint*** and not ***TaskWaypoint***. Do not forget to make consistent changes between the .cpp and .hpp.

"open_manipulator_x .cpp/.hpp" :

In the .hpp, change the **CUSTOM_TRAJECTORY_SIZE** and add a line to define its new path as those already present.

In the .cpp add the line to define the new trajectory in the ***Initialize Custom Trajectory*** **section** at the end of the file, following the example of the angular trajectories already defined. Be careful not to add the trajectory to the **custom_trajectory_** array as this would cause a compilation error as this array can only contain ***CustomTaskTrajectories*** and not ***CustomJointTrajectories.*** This will not cause any problems, as this array is not used anywhere.

III.4 – Further study: dynamic model and simulation

In the case the next group would like to improve the performances of the robot arm, we tried to establish a Simulink model of the arm. This should allow to simulate the output and thereby be able to identify the parameters that are most important for the improvement of one characteristic of the robot. For example, to widen the throwing-range, one can wonder if it is better to replace the motors by more powerful ones or to lighten the structure for example. In order to establish the Simulink model, it is necessary to first have a dynamic model of the robot.

III.4.A – Dynamic model

III.4.A.a – Geometric and kinematic models

One of the first steps to do is to calculate the kinematic model of the system, to link the angular velocities of the joints to the Cartesian velocities of the robot's gripper.

To do this, we start by making the geometric model of the system, which consists of making simple projections along the axes of reference.

The system is modeled by the following diagram:

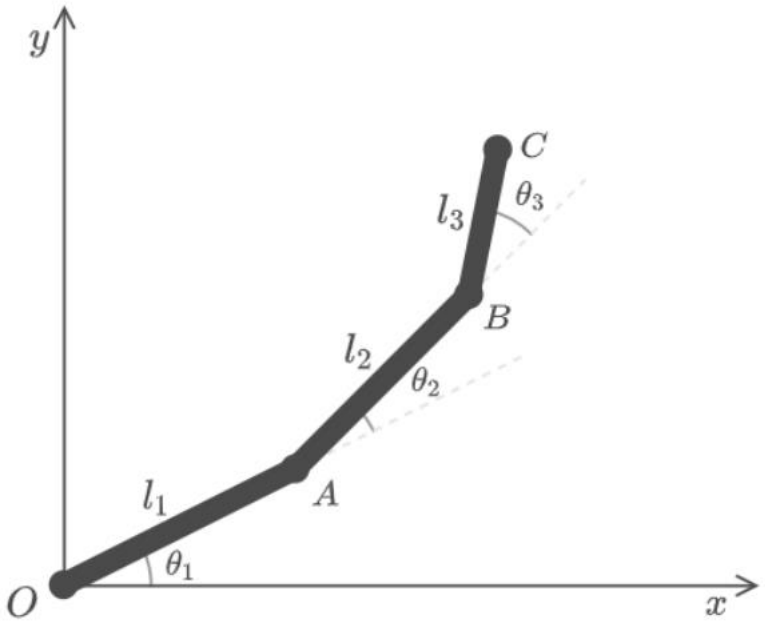


Figure 37: Robotic arm diagram model.

By projecting the robot segments on x and y, we end up with the direct geometrical model of the system.

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} l_1 \cos(\theta_1) \\ l_1 \sin(\theta_1) \end{pmatrix} + \begin{pmatrix} l_2 \cos(\theta_1 + \theta_2) \\ l_2 \sin(\theta_1 + \theta_2) \end{pmatrix} + \begin{pmatrix} l_3 \cos(\theta_1 + \theta_2 + \theta_3) \\ l_3 \sin(\theta_1 + \theta_2 + \theta_3) \end{pmatrix}$$

$$x = l_1 \cos(\vartheta_1) + l_2 \cos(\vartheta_1 + \vartheta_2) + l_3 \cos(\vartheta_1 + \vartheta_2 + \vartheta_3)$$

$$y = l_1 \sin(\vartheta_1) + l_2 \sin(\vartheta_1 + \vartheta_2) + l_3 \sin(\vartheta_1 + \vartheta_2 + \vartheta_3)$$

$$\phi = \vartheta_1 + \vartheta_2 + \vartheta_3$$

An inverse geometric model is also useful to know the angles of the joints in function of our desired position of the robot clamp.

A trigonometric manipulation gives us:

$$\begin{cases} \theta_2 = \pm \arccos\left(\frac{\bar{x}^2 + \bar{y}^2 - l_1^2 - l_2^2}{2 l_1 l_2}\right) \\ \theta_1 = \arctan 2(\bar{y},\ \bar{x}) \mp \arctan 2\left(l_2 \sin(\theta_2),\ l_1 + l_2 \cos(\theta_2)\right) \\ \theta_3 = \varphi - \theta_1 - \theta_2 \end{cases}$$

The next step is to represent the kinematic model, which is calculated simply by deriving the relations obtained in the geometric model with respect to time.

$$\dot{p} = \frac{\partial p}{\partial q}\dot{q} = J_p(q)\dot{q}$$

$$\dot{x} = -\dot{\theta}_1(l_1 s\theta_1 + l_2 s\theta_{12} + l3 s\theta_{123}) - \dot{\theta}_2(l_2 s\theta_{12} + l_3 s\theta_{123}) - \dot{\theta}_3(l_3 s\theta_{123})$$
$$\dot{y} = \dot{\theta}_1(l_1 c\theta_1 + l_2 c\theta_{12} + l3 c\theta_{123}) + \dot{\theta}_2(l_2 c\theta_{12} + l_3 c\theta_{123}) + \dot{\theta}_3(l_3 c\theta_{123})$$

With $l_3 s\vartheta_{123} = l_3 * \sin(\vartheta_1 + \vartheta_2 + \vartheta_3)$

By deriving the inverse geometric model, we obtain the inverse kinematic model, which can also be represented in matrix form:

$$\begin{pmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \dot{\theta}_3 \end{pmatrix} = \begin{bmatrix} \frac{c\theta_{12}}{l_1 s\theta_2} & \frac{s\theta_{12}}{l_1 s\theta_2} \\ \frac{-l_2 c\theta_{12} - l_1 c\theta_1}{l_1 l_2 s\theta_2} & \frac{-l_2 s\theta_{12} - l_1 s\theta_1}{l_1 l_2 s\theta_2} \\ \frac{c\theta_1}{l_2 s\theta_2} & \frac{s\theta_1}{l_2 s\theta_2} \end{bmatrix} \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix}$$

III.4.A.b – Dynamic model

This part consists of modelling the dynamics of the robot. This dynamic model can be used later to simulate the operation of the robot as well as to simulate a control by current.

To build this model, we will use the Lagrangian dynamics to establish all our relations.

This method is based on the energetic study of the system, and on the term called the Lagrangian.

$L(q,\dot{q}) = T(q,\dot{q}) - U(q)$

The Euler-Lagrange formulation will then be applied to find a relation between the kinematics of the system (angular) and the moments applied on the joints. Thus the dynamics of the system, which will then be useful to control the moments applied on the joints to achieve a precise kinematics desired.

We start by calculating the kinetic energy of the system, which is the sum of the kinetic energies of the arm segments.

The kinetic energy of the segments is represented by these relations:

$$T_1 = \frac{1}{2} m_1 v_1^2 + \frac{1}{2} I_1 \dot{\theta_1}^2$$
$$T_2 = \frac{1}{2} m_2 v_2^2 + \frac{1}{2} I_2 \left(\dot{\theta_1}^2 + \dot{\theta_1}^2\right)$$
$$T_3 = \frac{1}{2} I_3 \left(\dot{\theta_1}^2 + \dot{\theta_3}^2 + \dot{\theta_3}^2\right) + \frac{1}{2} m_3 \left(\dot{x_3}^2 + \dot{y_3}^2\right)$$

And so, the kinetic energy of the whole system is the sum of all kinetic energies:

$$T = \frac{1}{2}m_1 v_1^2 + \frac{1}{2}I_1\dot{\theta}_1^2 + \frac{1}{2}m_2 v_2^2 + \frac{1}{2}I_2\left(\dot{\theta}_1^2 + \dot{\theta}_2^2\right) + \frac{1}{2}m_3 v_3^2 + \frac{1}{2}I_3(\dot{\theta}_1^2 + \dot{\theta}_2^2 + \dot{\theta}_3^2)$$

To continue, it is necessary to calculate the potential energy of gravity of the system which amounts to summing the potential energy of each segment.

U1 = m1gy1 = m1g l_{g1} sin(θ1)

U2 = m2gy2 = m_2g (l_{g1} sin(θ1) + l_{g2} sin(θ1 + θ2))

U3 = m3gy3 = m_3g(l_{g1} sin(θ1) + l_{g2} sin(θ1 + θ2)+l_{g3} sin(θ1 + θ2 + θ3)

And so, the potential energy of the system is written:

U = U1 + U2 + U3

Having calculated the two energies, the Lagrangian of the system is written :

$L(q,\dot{q}) = T(q,\dot{q}) - U(q)$

We can now apply the Euler-Lagrange relation.

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{q}}\right) - \frac{\partial \mathcal{L}}{\partial q} = \mathcal{T} - \frac{\partial \mathcal{D}}{\partial \dot{q}}$$

We have to compute each term alone by doing simple derivatives. The calculation is long but simple.

Having computed all the terms, we can rewrite the Euler-Lagrange relation in matrix form.

$$\mathrm{T} = M(\theta)\ddot{\theta} + C(\theta,\dot{\theta}) + G(\theta)$$

With the symetric mass matrix $M(\vartheta)$:

$$M(\theta) = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ * & M_{22} & M_{23} \\ * & * & M_{33} \end{bmatrix}$$

The terms being:

$M_{11} = m_{123}l_{12} + m_{23}l_{22} + m_3l_{32} + 2m_3l_{13}\cos(\vartheta_{23}) + 2m_{23}l_{12}\cos(\vartheta_2) + 2m_3l_{23}cos(\vartheta_3)$

$M_{12} = m_{23}l_{22} + m_3l_{32} + m_3l_{13}\cos(\vartheta_{23}) + m_{23}l_{12}\cos(\vartheta_2) + 2m_3l_{23}\cos(\vartheta_3)$

$M_{13} = m_3l_{32} + m_3l_{13}\cos(\vartheta_{23}) + m_3l_{23}\cos(\vartheta_3)$

$M_{22} = m_{23}l_{22} + m_3l_{32} + 2m_3l_{23}\cos(\vartheta_3)$

$M_{23} = m_3l_{32} +$

$m_3l_{23}\cos(\vartheta_3)$ M_{33}

$=m_3l_{32}$

with

$m_{12\ldots n} = m_a + m_b + \cdots + m_n$

$I_{12\ldots n} = I_a \times I_b \times \cdots \times I_n$

To simplify the long expression

And then the velocity product term:

$$C(\theta,\dot{\theta}) = \begin{bmatrix} C_1 \\ C_2 \\ C_3 \end{bmatrix} \qquad (4.62)$$

With:

$$\begin{aligned} C_1 = & -m_{23}l_{12}\sin(\theta_2)\dot{\theta_2}^2 - m_3 l_{23}\sin(\theta_3)\dot{\theta_3}^2 - 2m_{23}l_{12}\sin(\theta_2)\dot{\theta}_1\dot{\theta}_2 - 2m_3 l_{23}\sin(\theta_3)\dot{\theta}_2\dot{\theta}_3 \\ & - 2m_3 l_{23}\sin(\theta_3)\dot{\theta}_1\dot{\theta}_3 - 2m_3 l_{13}\sin(\theta_{23})(\dot{\theta}_2+\dot{\theta}_3)(\dot{\theta}_1+\dot{\theta}_2+\dot{\theta}_3) \\ C_2 = & \left(m_{23}l_{12}\sin(\theta_2) + m_3 l_{13}\sin(\theta_{23})\right)\dot{\theta_1}^2 - m_3 l_{23}\sin(\theta_3)\dot{\theta_3}^2 - 2m_3 l_{23}\sin(\theta_3)\dot{\theta}_3(\dot{\theta}_1+\dot{\theta}_2) \\ C_3 = & \left(m_3 l_{23}\sin(\theta_3) + m_3 l_{13}\sin(\theta_{23})\right)\dot{\theta_1}^2 + m_3 l_{23}\sin(\theta_3)\dot{\theta_2}^2 + 2m_3 l_{23}\sin(\theta_3)\dot{\theta}_1\dot{\theta}_2 \end{aligned}$$

And finally, the gravity term:

$$G(\theta) = \begin{bmatrix} m_{123}g\,l_1\cos(\theta_1) + m_{23}g\,l_2\cos(\theta_{12}) + m_3 g\,l_3\cos(\theta_{123}) \\ m_{23}g\,l_2\cos(\theta_{12}) + m_3 g\,l_3\cos(\theta_{123}) \\ m_3 g\,l_3\cos(\theta_{123}) \end{bmatrix}$$

It is therefore our direct dynamic model that relates the torques on the joints to the kinematics of motion.

The inverse dynamic model is also useful and has the following form:

$$\ddot{\theta} = M^{-1}(\theta)\left(\mathcal{T} - C(\theta,\dot{\theta}) - G(\theta)\right)$$

M being a symmetric matrix, we can calculate its inverse which will be:

$$M^{-1}(\theta) = \begin{bmatrix} \dfrac{1}{m_1 l_1^2 + m_2\left(l1^2 - l_1^2\cos^2(\theta_2)\right)} & \dfrac{-l_2 - l_1\cos(\theta_2)}{l_1^2 l_2\left(m_1+m_2-m_2\cos^2(\theta_2)\right)} \\ \dfrac{-l_2 - l_1\cos(\theta_2)}{l_1^2 l_2\left(m_1+m_2-m_2\cos^2(\theta_2)\right)} & \dfrac{(m_1+m_2)l_1^2 + m_2 l_2^2 + 2\,m_2 l_1 l_2\cos(\theta_2)}{m_2 l_1^2 l_2^2\left(m_1+m_2-m_2\cos^2(\theta_2)\right)} \end{bmatrix}$$

Now, we can implement this matrix in the Simulink model for the robot arm, to be able to simulate its behaviour, compare it to the real one and then predict the behaviour after changing some parameters

III.5.B – Simulation

This section deals with the simulation of the Cartesian position-controlled system in Simulink. The system is assumed to be torque-controllable, so the control loop must convert a Cartesian input trajectory into a torque control of the three actuators on the joints of the robotic arm.

The control loop must perform the following tasks:

- Convert an input trajectory (in the plane of the ball throw) into a set of rotational speeds to be applied to the three motors
- Correct the control to optimize the tracking of motor torque commands (speed/torque conversion is implicit in this corrector, and a constant ratio is assumed)
- Recover (via sensors) the quantities that allow the determination of the control blocks

A control loop of the following form is therefore proposed:

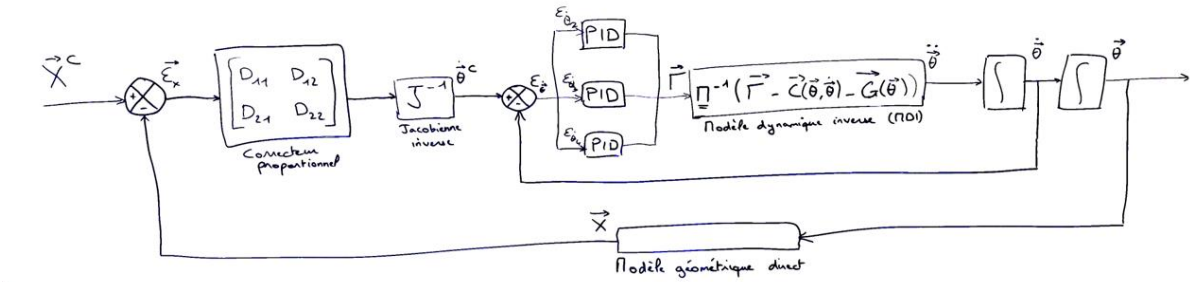


Figure 38: Block diagram chosen for the robot control (in simulation phase)

In this control loop, there are therefore two correction steps:

- A first correction step after the return of the real trajectory: here, we apply a proportional matrix corrector of size 2x2 $D = \begin{bmatrix} D_{xx} & D_{xz} \\ D_{zx} & D_{zz} \end{bmatrix}$, so 4 components must be set.
- A second correction step after the return of the angular velocity of the different robot axes: here, three independent PID $C_i(s) = P_i + D_i s + \frac{I_i}{s}, i \in [\![2;4]\!]$ are applied to the three different angular speeds $\omega_2 \coloneqq \frac{\partial \theta_2}{\partial t}$, $\omega_3 \coloneqq \frac{\partial \theta_3}{\partial t}$ and $\omega_4 \coloneqq \frac{\partial \theta_4}{\partial t}$, which corresponds to 9 parameters to be set.

For the simulation, the inverse dynamic model of the robot fits into the block named "MDI", given by the formula:

$$\frac{\partial^2 \boldsymbol{\theta}}{\partial t^2} \coloneqq \ddot{\boldsymbol{\theta}} = \boldsymbol{M}^{-1}(\boldsymbol{\Gamma} - \boldsymbol{C}(\boldsymbol{\theta}, \dot{\boldsymbol{\theta}}) - \boldsymbol{G}(\boldsymbol{\theta}))$$

Where the matrices M, $\boldsymbol{C}(\boldsymbol{\theta}, \dot{\boldsymbol{\theta}})$ and $\boldsymbol{G}(\boldsymbol{\theta})$ are calculated from the Lagrange formalism. A first proposal for the expression of these matrices can be found in the file "boucle_de_commande.slx", but the calculations have not been verified (take simple values to test the coherence of the model, test in "opposition of gravity" mode to ensure that the model works in static).

IV – Appendix

Appendix 1

We calculate the square matrix 3x3 $\boldsymbol{J}^T\boldsymbol{J}(\alpha, \beta, \gamma) \coloneqq [m_{i,j}]_{1 \le i,j \le 3}$:

$$\boldsymbol{J}^T\boldsymbol{J}(\beta, \gamma) = \begin{bmatrix} -(l_2 c_\alpha + l_3 s_{\alpha\beta} + l_4 s_{\alpha\beta\gamma}) & -l_2 s_\alpha + l_3 c_{\alpha\beta} + l_4 c_{\alpha\beta\gamma} \\ l_3 s_{\alpha\beta} + l_4 s_{\alpha\beta\gamma} & -(l_3 c_{\alpha\beta} + l_4 c_{\alpha\beta\gamma}) \\ l_4 s_{\alpha\beta\gamma} & -l_4 c_{\alpha\beta\gamma} \end{bmatrix} \begin{bmatrix} -(l_2 c_\alpha + l_3 s_{\alpha\beta} + l_4 s_{\alpha\beta\gamma}) & l_3 s_{\alpha\beta} + l_4 s_{\alpha\beta\gamma} & l_4 s_{\alpha\beta\gamma} \\ -l_2 s_\alpha + l_3 c_{\alpha\beta} + l_4 c_{\alpha\beta\gamma} & -(l_3 c_{\alpha\beta} + l_4 c_{\alpha\beta\gamma}) & -l_4 c_{\alpha\beta\gamma} \end{bmatrix}$$

The result is:

- $m_{1,1} = (l_2 c_\alpha + l_3 s_{\alpha\beta} + l_4 s_{\alpha\beta\gamma})^2 + (-l_2 s_\alpha + l_3 c_{\alpha\beta} + l_4 c_{\alpha\beta\gamma})^2$

$$\boxed{m_{1,1} = {l_2}^2 + {l_3}^2 + {l_4}^2 - 2l_2l_3\sin(\beta) - 2l_2l_4\sin(\beta+\gamma) + 2l_3l_4\cos(\gamma)}$$

- $m_{1,2} = -(l_2c_\alpha + l_3s_{\alpha\beta} + l_4s_{\alpha\beta\gamma})(l_3s_{\alpha\beta} + l_4s_{\alpha\beta\gamma}) - (-l_2s_\alpha + l_3c_{\alpha\beta} + l_4c_{\alpha\beta\gamma})(l_3c_{\alpha\beta} + l_4c_{\alpha\beta\gamma})$

$$\boxed{m_{1,2} = m_{2,1} = -{l_3}^2 - {l_4}^2 + l_2l_3\sin(\beta) + l_2l_4\sin(\beta+\gamma) - 2l_3l_4\cos(\gamma)}$$

- $m_{1,3} = -(l_2c_\alpha + l_3s_{\alpha\beta} + l_4s_{\alpha\beta\gamma})l_4s_{\alpha\beta\gamma} - (-l_2s_\alpha + l_3c_{\alpha\beta} + l_4c_{\alpha\beta\gamma})l_4c_{\alpha\beta\gamma}$

$$\boxed{m_{1,3} = m_{3,1} = l_2l_4\sin(\beta+\gamma) - l_3l_4\cos(\gamma) - {l_4}^2}$$

- $m_{2,2} = (l_3s_{\alpha\beta} + l_4s_{\alpha\beta\gamma})^2 + (l_3c_{\alpha\beta} + l_4c_{\alpha\beta\gamma})^2$

$$\boxed{m_{2,2} = {l_3}^2 + {l_4}^2 + 2l_3l_4\cos(\gamma)}$$

- $m_{2,3} = (l_3s_{\alpha\beta} + l_4s_{\alpha\beta\gamma})l_4s_{\alpha\beta\gamma} + (l_3c_{\alpha\beta} + l_4c_{\alpha\beta\gamma})l_4c_{\alpha\beta\gamma}$

$$\boxed{m_{2,3} = m_{3,2} = l_3l_4\cos(\gamma) + {l_4}^2}$$

- $m_{3,3} = (l_4s_{\alpha\beta\gamma})^2 + (l_4c_{\alpha\beta\gamma})^2$

$$\boxed{m_{3,3} = {l_4}^2}$$

It can therefore be seen that the angle α is not involved in the expression of $\boldsymbol{M}^T\boldsymbol{M}$:

$$\boxed{\boldsymbol{J}^T\boldsymbol{J}(\alpha,\beta,\gamma) \rightarrow \boldsymbol{J}^T\boldsymbol{J}(\beta,\gamma)}$$

Appendix 2

The set of parameters used is shown below:

- $x_{01} = 0$
- $x_{f1} = x_{02} = 0{,}2$
- $v_{01} = 0$
- $v_{f1} = v_{02} = 0{,}25$
- $v_{f2} = 0$
- $a_{01} = 0$
- $a_{f1} = a_{02} = 0$
- $a_{f2} = 0$

It should be noted that the time instants allowing to define the duration of the two phases (and thus the abscissae of the various curves in which the quantities must reach their imposed values) are defined in the following way: one considers a total execution time $t = 2$, during which the two phases will be executed in sequence, and a ratio $r \in [0; 1]$ (here $r = 0{,}68$) which subdivides this time into two execution periods for each of the two phases (phase 1 takes place over a time rt, phase 2 takes place over a time of $(1-r)t$).

V - References

[1] **Zeng A, Song S, Lee J, Rodriguez A, Funkhouser T**. 2020. TossingBot: learning to throw arbitrary objects with residual physics. *IEEE Trans. Robot.* 36(4):1307-1319. https://doi.org/10.1109/TRO.2020.2988642

[2] **Sadekar K**. 2020 Dec 28. Introduction to Epipolar Geometry and Stereo Vision. *LearnOpenCV* ; [accessed 2023 Jan 28]. https://learnopencv.com/introduction-to-epipolar-geometry-and-stereo-vision/

[3] **Sadekar K**. 2021 Apr 5. Stereo Camera Depth Estimation With OpenCV (Python/C++). *LearnOpenCV* ; [accessed 2023 Jan 28]. https://learnopencv.com/depth-perception-using-stereo-camera-python-c/