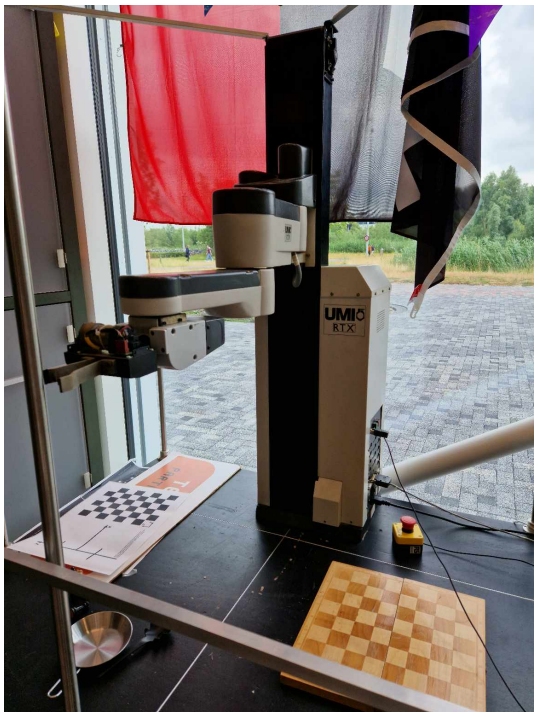

Mid-Term report
A ROS 2 Interface for the UMI-RTX robotic arm

Guillaume GARDE and Théo MASSA
Under the supervision of Arnoud Visser
Intelligent Robotics Lab, Universiteit van Amsterdam, The Netherlands
ENSTA Bretagne, France

June 28, 2023

Abstract

The aim of this project is to work with an old robotic arm called the UMI-RTX (created in the 1980's) and make it grab objects on a plane with its gripper. Some work has already been done on this robot by students, but mainly with old tools. The idea of our project is to implement a new way of making it work and to use more recent tools. More specifically, our goal is to set up a ROS 2 environment and build an interface that will allow us to perform image analysis, trajectory planning, and target grabbing.

We have chosen a plush banana as a target. With the computer vision library OpenCV, we managed to detect this banana with our camera, in a dedicated ROS 2 node, and to get the coordinates of its centroid. Besides, we managed to calibrate our stereo camera, the ZED M module, and begin the depth estimation needed to locate the target in a 3D space.

Once this is done, this information is sent into our ROS 2 architecture, more specifically to a node dedicated to inverse kinematics. In this node, the joints' states required to reach the aimed pose are processed and sent both to a simulation and the real arm. For this, we have two nodes, each dedicated to its own part, one for the simulation, the other for the real arm.

Finally, we designed a custom Graphic User Interface (GUI) in which the simulation and the processed image are integrated, and in which we are able to choose between automatic control of the arm and a manual mode, where we can choose our own target's position.

Contents

Chapter 1

Introduction

1.1 Context

This project is led in an internship context, mainly for educational purposes. We worked on an old industrial arm, the UMI-RTX, which was created in the 1980s. Despite his age, the arm is still compatible with recent software and hardware, so it is interesting to work on it. Its educational appeal is obvious when we consider that a lot of work has already been done on it, especially by Dooms [4] and Van Der Borght [2] who worked on a ROS interface in order to control it.

1.2 Objectives

Our objectives are plural. First, it is to learn new knowledge. As students, this kind of internship is strong in apprenticeships, and this is a good opportunity. But more importantly, we have the objective of creating a ROS 2 interface in order to be able to control the arm. The robot should be able to detect a recognisable object, a yellow banana plush in our case, move to this object, and take it. To this extent, we have to rely on previous works and obviously create, from not much except drivers, a full ROS 2 interface.

Chapter 2

Arm manipulation

2.1 Description

This project uses the UMI-RTX arm, which is quite simple in its composition. Indeed, it is composed of an axis to translate on the z-axis and a three-part arm, where each part is connected to another through revolute joints. Those joints can be controlled through both position and velocity, but in this project, we only control them through position, as it is more adequate to our project, which is to grab a target, a mission that requires to go to a specific position. Our method is also more adapted to a position control. Each motor has encoders that allow it to be controlled and know its state.

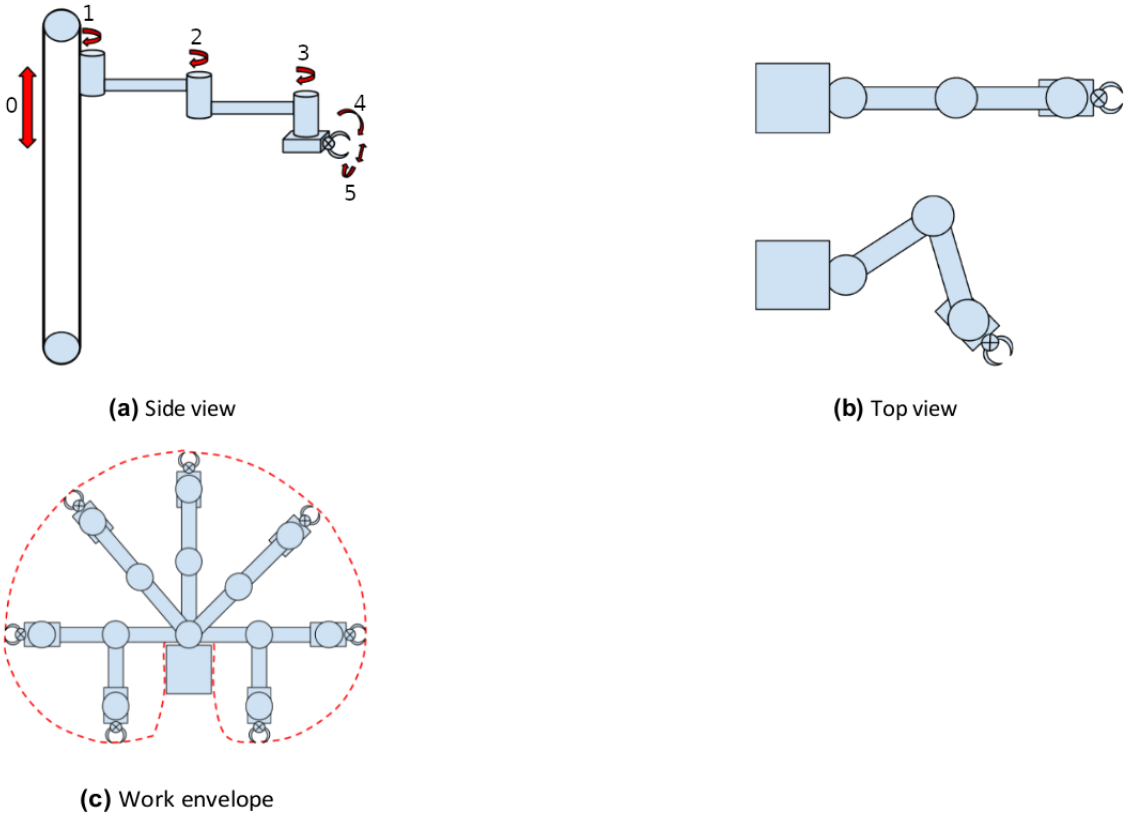

Figure 2.1: Model of the arm

As we can see in the Figure, this arm can be compared to a human arm, at least in a conception way. Joint 1 corresponds to the shoulder, joint 2 to the elbow, and ensembles 3-4-5 to the wrist. For the rest of this document, they will be referred to as we can see in the following table:

Table 2.1: Description of the joints ID

Joint number	Joint ID
0	**ZED**
1	**SHOULDER**
2	**ELBOW**
3	**YAW**

One characteristic of this arm is how the roll and pitch of the hand work. They are not controlled separately but together by two motors, one on each side. A view of this system can be seen in the following Figure:

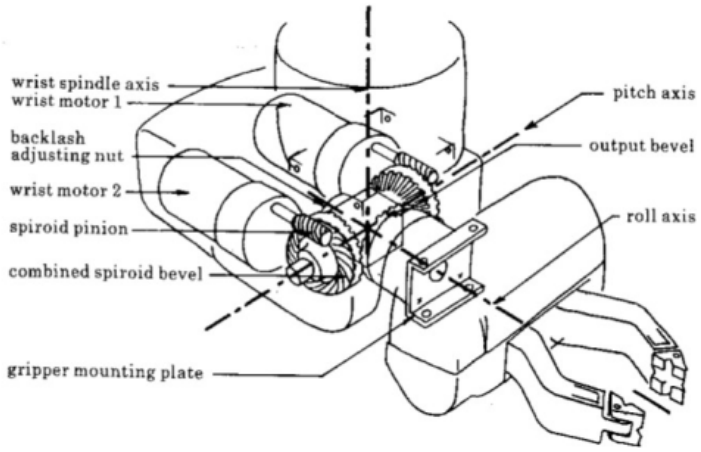

Figure 2.2: Wrist system

This particular system has to be taken into account when controlling the arm, and the two motors will be referred to as **WRIST1** and **WRIST2**.
The only part that we haven't managed to control yet is the grip.
Once we have understood how this arm is assembled and what joints to handle, we are able to begin manipulating it.

2.2 URDF description

For practical purposes, it is really useful, even mandatory, to have a digital twin of the arm. To this extend, it seems appropriate to use an URDF description of the arm. This URDF (Unified Robotics Description Format) allows one to manipulate virtually the arm and previsualize what effects the commands would have on the arm. Particularly in

robotics, having a virtual clone of our system is always something important, and each time it is more researched.

This description consists of a description of every part and joint, describing the geometry of the blocks, the joints between them, their type, limits, etc. One can see below an extract of the description of the arm. The entire description can be found in the annex.

Listing 2.1: URDF Description of the arm

```
<robot name="umi-rtx">

  <link name="base_link">
    <visual>
      <geometry>
        <box size="1.252 0.132 0.091"/>
      </geometry>
      <origin rpy="0 -1.57 1.57" xyz="0 -0.0455 0"/>
      <material name="blue">
        <color rgba="0 0 .8 1"/>
      </material>
    </visual>
  </link>

  <joint name="shoulder_updown" type="prismatic">
    <parent link="base_link"/>
    <child link="shoulder_link"/>
    <origin xyz="0 0.0445 -0.3" rpy="0 0 1.57"/>
    <!-- xyz="0.0445 0 0.134" -->
    <axis xyz="0 0 1"/>
    <limit lower="0.033" upper="0.948" effort="1" velocity="1"/>
  </joint>

  <link name="shoulder_link">
    <visual>
      <geometry>
        <box size="0.278 0.132 0.091"/>
      </geometry>
      <origin rpy="0 -1.57 0" xyz="0 0 0"/>
      <material name="white">
        <color rgba="1 1 1 1"/>
      </material>
    </visual>
  </link>
```

This description will be particularly useful when it comes to seeing the virtual model in our simulation and processing the inverse kinematics (see *2.4*).

There is only one main difference between this description and reality, which is the wrist, particularly the pitch and roll. In this description, there are two independent joints dedicated to pitch and roll, whereas in reality, we saw before that two motors worked together to handle those angles. Therefore, we have to be careful when converting this

description into reality. We have:

$$\mathbf{WRIST1} = \frac{roll + pitch}{2}$$
$$\mathbf{WRIST2} = \frac{pitch - roll}{2}$$

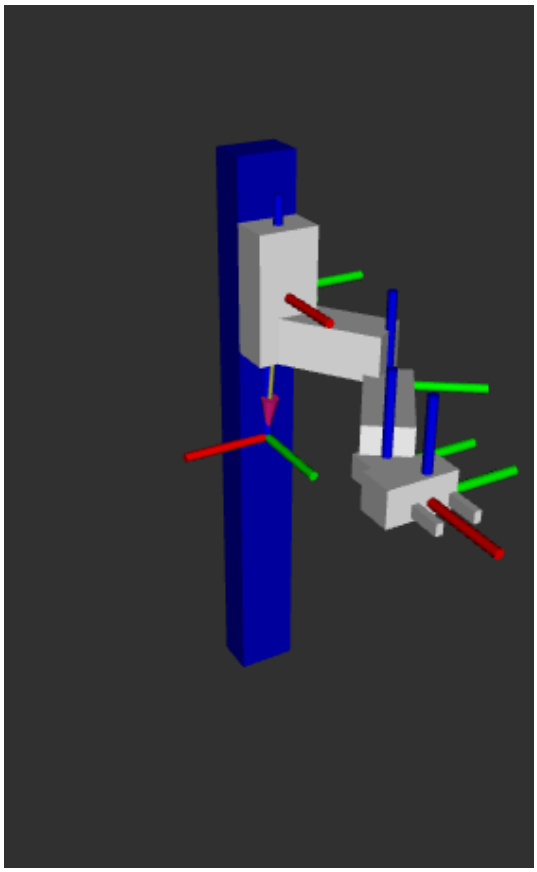

Figure 2.3: Virtual model of the arm

On this model, every frames is attached to its part and represented in red, green and blue, for the x, y and z axis.

2.3 Communication with the arm

As the arm is old, the communication is not direct. The documentation is limited, and there are no ready-to-use drivers or software furnished by the creator of the arm. It is necessary to use a TCP/IP connection through the RS232 bus between the computer and the arm to send commands or acquire data from the arm. Doing this is already a lot of work, but thanksfully, we had drivers developed by previous students at our disposal. Thanks to our supervisor A. Visser, we have a fully developed driver that communicates with the arm, that was furnished by him.

One particularity of the arm, is that the motors are controlled by two 8031 chips, IPs[1] called. Each IP ensures the proper operation of a selection of motors. Table 2.2 shows

[1]Intelligent Peripheral

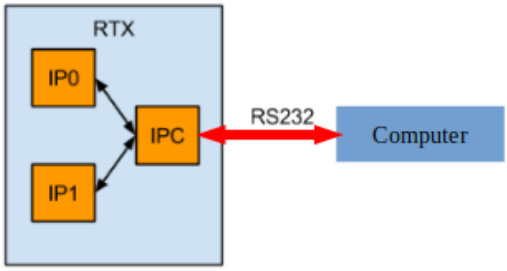

Figure 2.4: Communication between the arm and the computer

which motors there are with their corresponding IP. So, for example, to move the arm up and down (zed), it must first be switched to IP1. Once switched, the new command of the motor can be entered [4]. IPC stands for Intelligent Peripheral Communication, and it uses three possible ways of communication, relying on a request from the computer and a response from the arm (see Figure 2.5).

ZED	SHOULDER	ELBOW	YAW	WRIST1	WRIST2	GRIPPER
IP1	IP1	IP1	IP1	IP0	IP0	IP1

Table 2.2: Overview of motors with corresponding IP

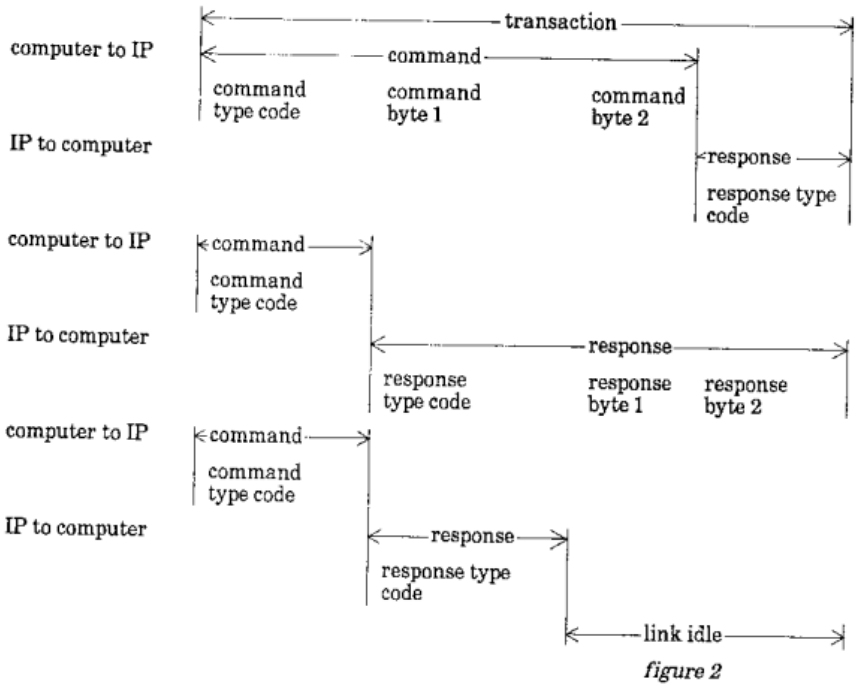

Figure 2.5: 3 ways of communication

However, this driver is one that allows to control the arm only via the terminal, when we want to build an interface that does not just happen via the terminal. For this, we had to look into the source code of the drivers, understand how they manage to communicate with the arm, and reuse their functions in our own code.

In order to use the arm, we have to follow a precise procedure. First, we have to start communication with the arm by launching the daemon created by Dooms and Van Der Borght and specifying which USB port is used by the arm. Then, in our codes, we have to initialise the communications by sending a certain code to the arm, and every time we use the arm, an initialization procedure has to be processed for the arm to know where the encoder's limits are. Indeed, there is no real memory of the encoders' limits and parameters, so those have to be initialised before every use of the arm. Fortunately, the drivers contain everything necessary to do so.

Now, we have everything ready to command the arm. To do so, we first have to write in the arm what we want to do, then tell it to go in the state written before.

2.4 Inverse kinematics

Inverse kinematics is one of the main parts of this project. It consists of processing the state of each joint given the desired pose of the end-effector. Unlike forward kinematics, where we process the end-point pose given the state of every joint, we do the opposite here.

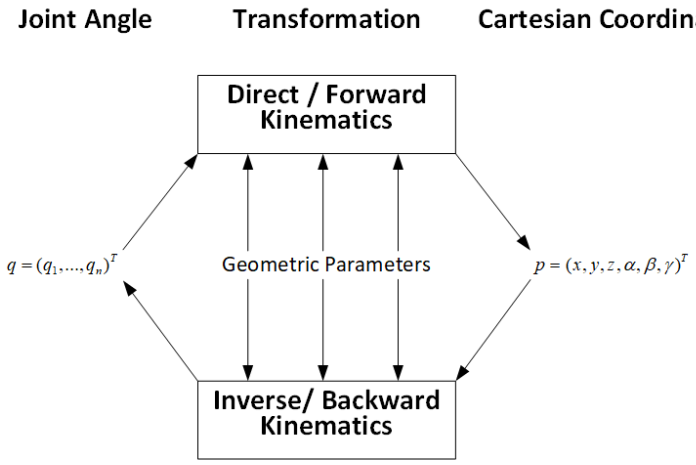

Figure 2.6: Inverse and forward kinematics

The inverse kinematics process is way more complicated than forward kinematics because there are none, one, or multiple solutions, and the difficulty increases with the number of joints or degrees of freedom. Fortunately, each joint has only one degree of freedom, so computing is a bit simplified.

To compute those inverse kinematics, we chose to use a C++ library named Pinocchio [3], which allows us to create algorithms that will process the inverse kinematics. This library was selected due to its versatility and efficiency, but also and mainly because of its integration into ROS 2 packages.

To process the inverse kinematics, we use a method called CLIK, for Closed-Loop Inverse Kinematics [5], using methods and object from Pinocchio library. This iterative algorithm allows us to find the best state of each joint in order to be as close as possible to an objective defined by a position (x, y, z) and an orientation $(yaw, pitch, roll)$.

Let's explain this algorithm:

Let be (x, y, z) the desired position and $(\phi, \theta\, \psi) = (yaw, pitch, roll)$ the desired orientation.
We define the different rotation matrices by :

$$R_\phi = \begin{bmatrix} cos(\phi) & -sin(\phi) & 0 \\ sin(\phi) & cos(\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$
$$R_\theta = \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix}$$
$$R_\psi = \begin{bmatrix} 0 & cos(\psi) & -sin(\psi) \\ 0 & sin(\psi) & cos(\psi) \\ 1 & 0 & 0 \end{bmatrix}$$

And the desired rotation matrix by :

$$R = R_\phi.R_\theta.R_\psi$$

Finally, the desired pose lies in $SE3$ space, defined by the desired position and R, the desired rotation.

Then we define q, a vector defining the initial state of the arm. Each value of q corresponds to a joint "value". For example, the joint value for **ZED** will be in metres, whereas **SHOULDER**, **ELBOW**... are in radians.

$$q = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \end{bmatrix} = \begin{bmatrix} ZED \\ SHOULDER \\ ELBOW \\ YAW \\ PITCH \\ ROLL \end{bmatrix}$$

This vector is initialised, whether at the neutral position of the arm or at the last known state. Initialising this vector at the last state known allows a sort of continuity in the solutions, because of the iterative method that is used after that.
Once we have all that, we can initiate the iterative process.

- First, we compute the forward kinematics with the current configuration defined by the vector q

- Then, we get the transformation $T \in SE3$ between the current pose and the desired one and the logarithmic error defined by :

$$err = log(T)$$

- if $||err|| \leq \epsilon$ with ϵ a defined coefficient that characterises the precision we want, or if we did a certain amount of iteration, we stop the iterative process

- Else, we compute the Jacobian J of the current configuration.

- We define the vector v thanks to the damped pseudo-inverse of J in order to avoid problems at singularities:

$$v = -J^T(JJ^T + \lambda.I)^{-1}.e$$

  v can be considered the speed vector that will get the configuration closer to the desired one.

- Then we can integrate $q = q + v.dt$ and reiterate this process.

Once this iterative process is over, we have the required configuration stored in q in order to reach the desired pose. Finally, we transform our angles from $[0, 2\pi]$ to $[-180, 180]$ for practical purposes and send the required state on the corresponding ROS topic */motor_commands*.

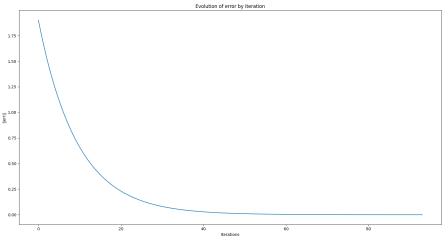

Figure 2.7: Evolution of the error according to iterations

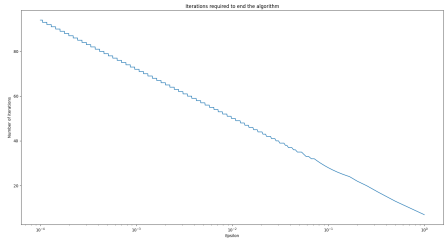

Figure 2.8: Evolution of the number of iterations according to epsilon in logarithmic scale

As we can see in the two Figures above, this method is quite efficient as the error decreases exponentially. It also confirms the fact that our algorithm converges towards a solution. On the other side of the coin, the smaller the epsilon, the greater the number of iterations required, up to a point where it is no longer possible to converge within a reasonable time or even to converge at all.
Look forward to *Chapter 4* to see how the inverse kinematic calculation is integrated into our project.

Chapter 3

Computer vision

To make the robotic arm grab the target, one has to rely on one key element: computer vision. This element is a set of several techniques to see the scene of interest with an optical device and extract valuable information from it. In this project, these techniques are used to detect the target and get its 3D position in the camera's frame. The language of programming that will be used here is C++, and the OpenCV[1] methods that will be cited here will be written accordingly with the C++ syntax.

3.1 Detection of the target in a horizontal plane

The target is a yellow banana plush. It will be put on a dark horizontal plane on which the UMI-RTX is fixed.

Figure 3.1: The banana plush on the dark horizontal plane that supports the arm

The first task of the computer vision part is to detect this banana. The banana was chosen because it is a convenient target. It is a standard object easy to find on Ikea; its colour is convenient to detect; its softness makes it easy for a gripper to grab it; and it

[1]OpenCV documentation: https://docs.opencv.org/4.7.0/

is coherent with the fact that most objects are not rectangular but have curves, therefore the approach is more general. The image process is made with OpenCV which includes build in methods for computer vision. To extract the banana from the scene, one works in a specific colour space: the HSV colour space (Hue, Saturation, Value) [10]. It is more common to hear about the RGB [10] colour space (Red, Green, Blue), in which each colour is represented by a set of three values between 0 and 255 corresponding to a proportion of the associated colour. This is the colour space used to associate a colour to screen's pixels. However, the HSV space is an appropriate colour space to perform colour detection. Each colour is represented by a triplet of values between 0 and 255 corresponding to its values of hue, saturation, and value [10].

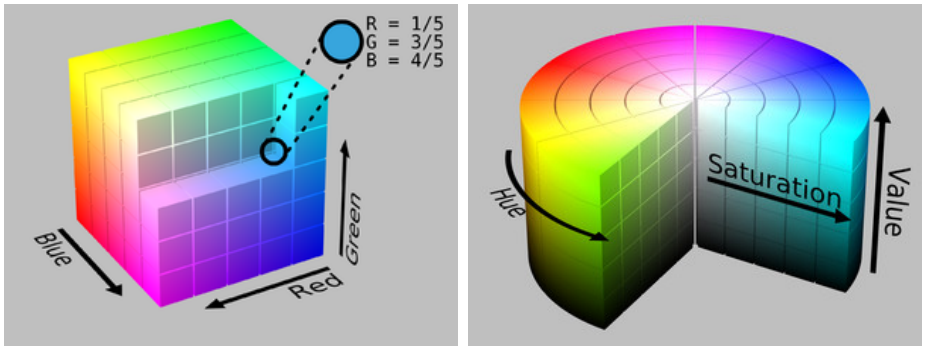

Figure 3.2: Representation of the RGB and HSV colour spaces

The extraction of an object from an image is based on contour detection once the image has been binarized according to a specific strategy. In this case, two HSV thresholds have been selected, (20,100,100) and (60,255,255), to extract objects in between. These values have been chosen to binarize the image with *cv::inRange()* and extract yellow objects that have similar HSV values. The result is a binarized image with white objects on a black background. Then one can perform contour detection on these objects. A hypothesis made for the project is that the only visible yellow object in the scene would be the banana target. This ensures that when performing contour detection, only the contour of the banana is found. The method *cv::findContours()* gathers all the contours detected, in this case only the contour of the target. Then one can access the moments of the contour with *cv::moments()* and compute the coordinates of its centroid in the reference frame of the image.

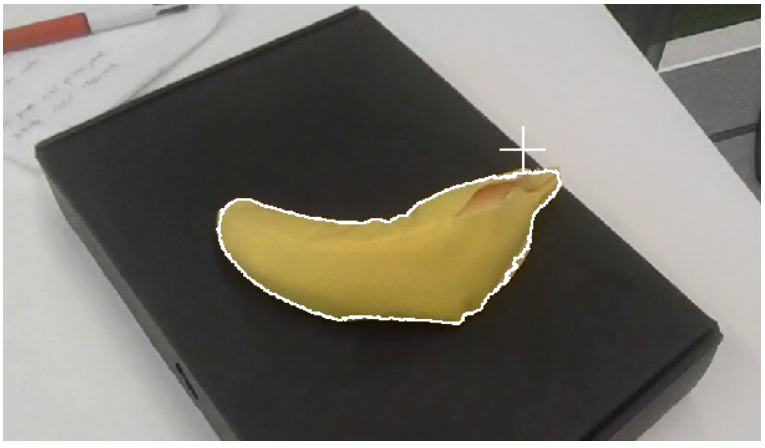

Figure 3.3: Example of detected banana and its contour with OpenCV

With the coordinates of the centroid, one can locate the banana in the horizontal plane. The next step is to access its depth with respect to the camera, to grab it with the UMI-RTX's gripper.

3.2 Generating depth with stereo vision

To allow the arm to grab the target, it needs to know where it is. The first step of detecting the banana in a horizontal plane can be done with a single camera, but getting its depth is more complex and requires a second one [8]. This is called stereo vision. The stereo device used in this project is the ZED M camera device from *StereoLabs*.

Figure 3.4: The ZED M stereo device

Note that this is a specific type of device. Both lenses are on the same support and have parallel optical axes and coplanar image planes. Some stereo installations use two distinct cameras that can be separated from each other according to need.

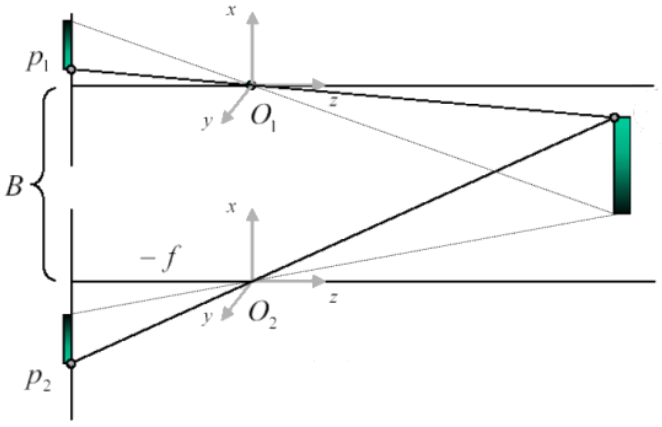

Figure 3.5: Vision of an object (on the right) with a stereo camera (on the left). O_1 and O_2 are the optical centers and B the distance between the optical axes. [10]

The main idea behind stereo vision is to reproduce human vision [11]. One will use the difference in perception of the scene to extract depth information. OpenCV provides methods and algorithms to get to that point step by step [1]. The theory behind these

methods belongs to computer science and vision. The calculations made to extract information from the views fall within the framework of *projective geometry* and *epipolar geometry* [6][8].

3.2.1 A bit of geometry

Projective space is an extension of Euclidean space where parallel lines meet at infinity [8]. To work in projective space, one has to use *homogeneous coordinates* [11][8]. These coordinates are used to characterise changes in space and allow to consider points at infinity and to calculate with points that are not at infinity with matrices as in Euclidean space. Projective geometry is used in computer science to manipulate coordinates, but it is not the only one used. The other mathematical aspect is epipolar geometry. It describes the relationship between two views of the same object [11].

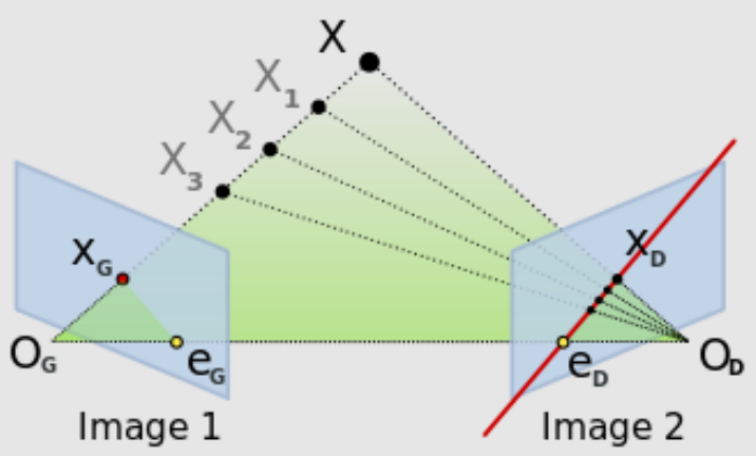

Figure 3.6: Representation of the epipolar plane [11]

Characterising the links and differences between the two views of a stereo device is essential to performing any type of scene reconstruction.

3.2.2 Parameters

One needs to know some parameters associated with the scene and the camera in order to access depth information. The first type of parameter is the *intrinsic* parameters [11], which are internal to the camera, such as the focal length f or the baseline B, which is the distance between the optical axes. The second type of parameter is the *extrinsic* parameters [11], which are a rotation matrix R that links the scene reference frame to that of the camera and a vector T corresponding to a translation that links one reference frame to the other. The third type of parameter is the *fundamental matrix* F [8] [11] that contains all the epipolar information of the views, and the *camera matrices* that describe the mapping of 3D points in the world to 2D points in the images. Accessing depth information can only be done through the determination of these parameters, thanks to a well-thought-out strategy.

3.2.3 Calibrating the stereo camera

The first step of the process is to calibrate the ZED M device in order to compute the extrinsic parameters, the fundamental matrix and the camera matrices [8][11]. To do

so, one has to use stereo images with easy-to-detect points and apply correspondence algorithms to compute the results. In this project, a black and white chessboard was photographed five times[2] in different poses.

Figure 3.7: One of the views of the chessboard used to calibrate the ZED M device

One has to declare the inner pattern that will be searched by the algorithm. In this case, it is the inner part of the chessboard that has 7 by 5 corners[3]. It is also important to provide the algorithm with the size of a square (3.1 cm here). Then, for each pair of images, one has to use *cv::findChessboardCorners()* for the left and the right views. This method will detect the declared pattern in the images and the associated corners. Then one has to use *cv::cornerSubPix()* to refine the positions of the detected corners.

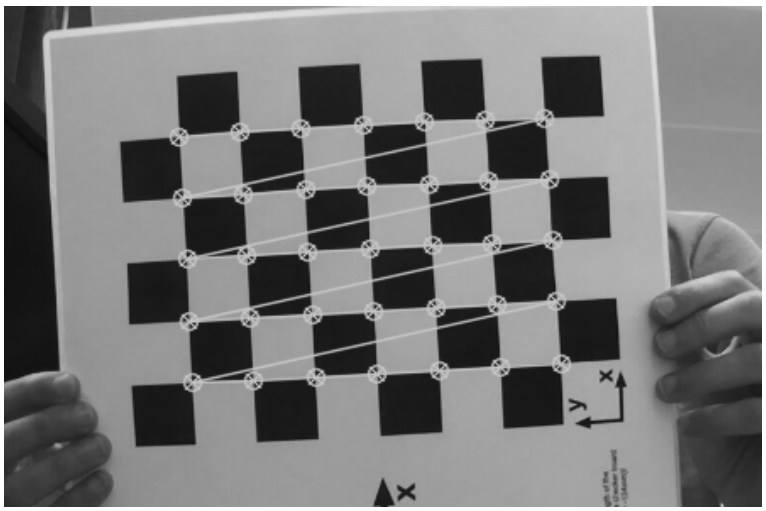

Figure 3.8: Detected corners on the precedent left view, associated to the declared pattern

Once this is done, the 2D positions of the corners in the left and right images are saved in associated vectors, and, for each pair of images, the 3D coordinates[4] of the corners[5] with respect to the top left corner are saved in a dedicated vector. Which means that one has a vector whose five components are the same and two other vectors whose components are relative to the images. Finally, one uses *cv::stereoCalibrate()* to compute the following parameters: calibration RMS[6] error, left camera matrix, right camera matrix, left distortion coefficients, right distortion coefficients, rotation matrix, translation vector,

[2]To guarantee robustness.
[3]It is very important to correctly count the inner corners for the algorithm to work.
[4]With third coordinate set to 0 for now.
[5]Using real dimensions.
[6]Root-Mean-Squarre.

essential matrix, and fundamental matrix. The algorithm that summarises the process is in the annex.

3.2.4 Stereo rectification parameters computation

After calibrating, one must rectify some things. The goal here is to "*compute the rotation matrices for each camera that (virtually) make both camera image planes the same plane. Consequently, this makes all the epipolar lines parallel and thus simplifies the dense stereo correspondence problem*". For this part, one uses the camera matrices and distortion coefficients that were computed during the previous step. One must use *cv::stereoRectify()*, and it will compute rectification transforms (rotation matrices) for the cameras, projection matrices in the new rectified coordinate systems for the cameras, and a disparity-to-depth mapping matrix.

3.2.5 Stereo rectification

The next part is to "*computes the joint undistortion and rectification transformation and represent the result in the form of maps for remap*" by using *cv::initUndistortRectifyMap()* To do so, one must use the parameters found during the previous step. Then one can remap the images to be rectified (in this case, the work scene) with *cv::remap()*. This must be applied once for each view. On the following images is an example of an image taken by the left camera and its associated rectified image.

Figure 3.9: An example of the left view and the rectified image

It is complex to say why the second image is better to use by just looking at it, but it is the result of the rectification process, and now everything is set to work on disparity.

3.2.6 Disparity map computation

Stereo vision reproduces human vision [11][8]. Each camera of the stereo device will perceive the scene in its own way, just like the human eye. Therefore, the scene seems to be shifted to the right or left, depending on the considered view. This is where in-depth information can be found. Between the views, there will be a difference in horizontal positioning for each object. This difference can be measured in pixels, for instance, and is called *disparity* [11][8]. The principle is, then, simple. When considering an object in a stereo image of a scene, the bigger the disparity, the closer the object. Computing disparity for every object in the scene allows for the creation of a disparity map. Moreover, considering that depth (Z) and disparity (d) are proportionate following the equation $Z = \frac{fB}{d}$, one can access a depth map with a disparity map.

OpenCV provides two algorithms to compute disparity between two stereo-associated views: *cv::stereoBM()* and *cv::stereoSGBM()*, which is a modified version of the first one. These two algorithms perform horizontal block matching between the views. One has to set some parameters[7] of the constructor before computing the disparity map. Each parameter has a precise influence on the results, and it may be difficult to set them correctly. A brief description can be found in the annex. One thing important to remember, though, is that these algorithms are sensitive to texture. A lack of texture in the images can result in poor results. In this project, many configurations (different images and parameter values) have been tried and given various results. On the following pictures, one can see an example of a "common perception image"[8] and the associated disparity map.

Figure 3.10: The "common perception image".

[7] `http://wiki.ros.org/stereo_image_proc/Tutorials/ChoosingGoodStereoParameters`

[8] An extraction from the left view, on which were kept objects that are seen by both cameras.

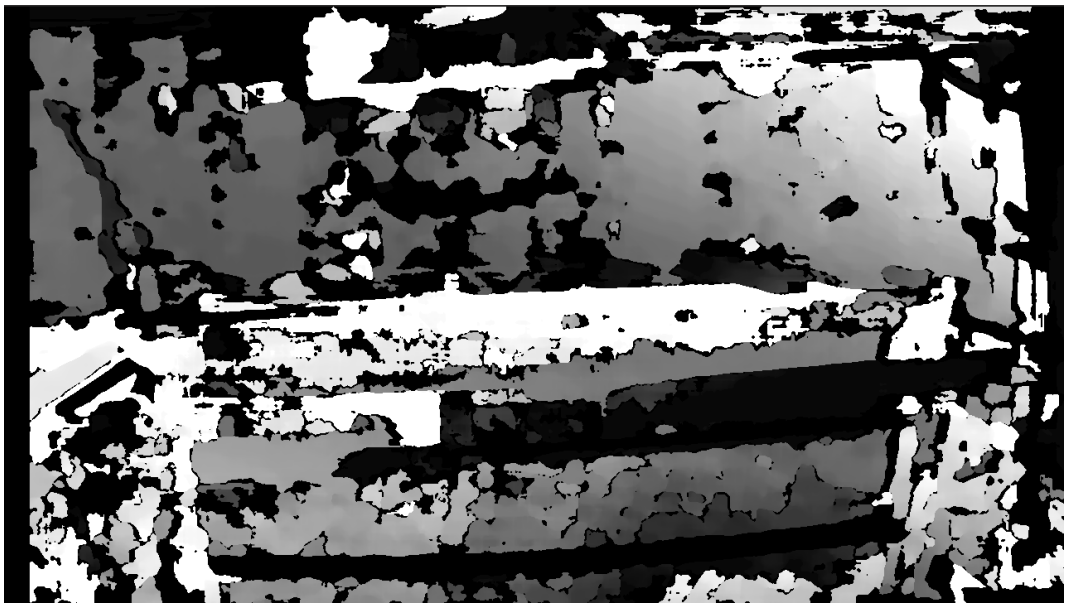

Figure 3.11: The associated disparity map.

On the disparity map, one notices that there are many levels of grey. The brighter, the closer. This map is still noisy, but it is one of the cleanest made during the project. One has to keep in mind that finding convenient parameter values is difficult and that it is possible to scale down the disparity or normalise it. To easily get better results, the disparity map has been adjusted with trackbars linked to each parameter. However, changing one parameter often strongly disturbs the harmony of the map, making it difficult to get a good result. The depth map associated with this image is not very good yet.

Figure 3.12: The associated depth map.

Chapter 4

ROS 2 Interface

4.1 Configuration and ROS 2 presentation

For this project, we work on Ubuntu 20.04 and are using ROS 2 Foxy.

Figure 4.1: ROS 2 Foxy Fitzroy

ROS 2 (Robot Operating System 2) is an open-source framework designed for building and controlling robotic systems [7]. It is the successor to ROS 1 and offers several improvements and new features to enhance the development and deployment of robotics applications. With its modular and distributed architecture, ROS 2 provides a flexible and scalable platform for creating advanced robot systems.

Utilising a pub-sub messaging model, ROS 2 enables efficient communication between different components of a robot system, facilitating the exchange of data and commands. It supports multiple programming languages and provides a variety of tools and libraries that simplify the development process. ROS 2 also focuses on real-time and embedded systems, making it suitable for a wide range of robotic applications, from small embedded

devices to large-scale distributed systems [7].

ROS 2 brings several key benefits to developers and roboticists. It offers improved performance and reliability thanks to its optimised middleware, which enables faster and more efficient communication between nodes. This enhanced performance is particularly beneficial for applications requiring real-time or low-latency operations. ROS 2 also emphasises security and safety with features such as authentication and fine-grained access control, making it more suitable for sensitive applications and environments [7].

ROS 2 communications are based on two principal concepts: nodes and topics. Nodes are individual software modules that perform a specific task within a robotic system. Nodes are the fundamental building blocks of a ROS application, and they communicate with each other by passing messages. On the other side of the coin, topics are the communication channels used by nodes to exchange messages in ROS. A topic is a named bus where nodes can publish messages or subscribe to receive messages. It follows the publish-subscribe communication pattern, where nodes that generate data publish messages to a topic, and nodes interested in that data subscribe to the topic to receive the messages [7].

4.2 ROS 2 Architecture

When running only the simulation, here are the interactions between every ROS node:

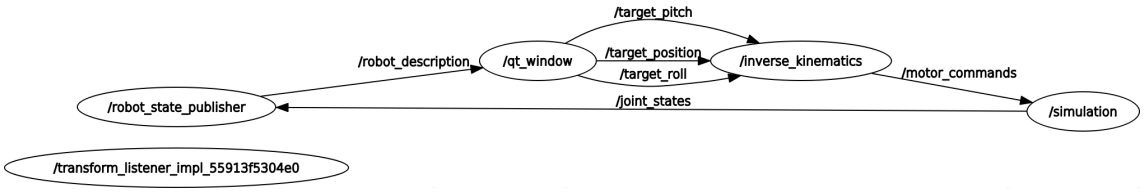

Figure 4.2: Node graph when running only the simulation

Here, the main node is */qt_window*. This node is the one where the GUI is executed. It sends the targeted position, roll and pitch to the */inverse_kinematics* node that will process the inverse kinematcs in order to get and send the required joints' state that will make the arm reach its target, through the */motor_commands* topic. The */simulation* node subscribes to this topic and, through */robot_state_publisher*, sends the robot description to the integrated simulation panel in our GUI that will be described in *4.4*.

We can see here that there are no topics dedicated to sending a targeted yaw. This is because for now, yaw stays equal to $arctan2(y, x)$ where x and y are the targeted position. In fact, this configuration is the easiest to manage because, for the real arm, the yaw is equal to 0 when it is in reality equal to this value. To be clearer, in simulation, the yaw origin is the y-axis, whereas for the encoders, the yaw origin and neutral point are following the axis between the zed-axis and the wrist.

It is also important to note that, as of yet, there is no camera node. This is because this node is not finished yet, so we didn't include it. However, the data coming from this node—a point message and potentially the angles too—will be sent to the */qt_window*, which is an intermediary between the camera and simulation/arm.

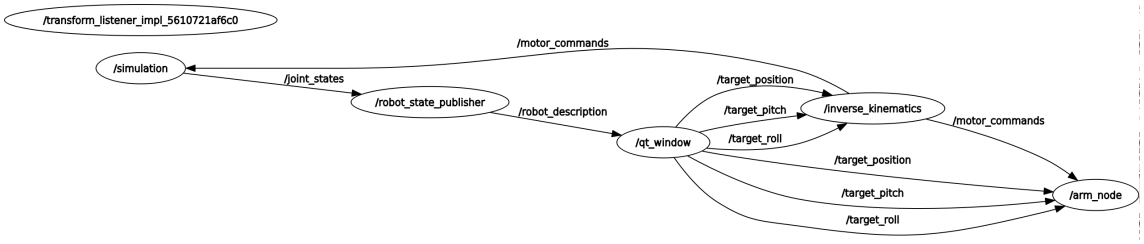

Figure 4.3: Node graph when running real arm

For the real arm, the node graph is a bit more complicated with the use of the arm node. This node also subscribes to the targeted pose to be aware of when the target changes. By doing so, we can avoid useless calculations. Indeed, if we send commands to the arm only when the target changes, it will cost less resources than trying to send commands at every loop, and it will also be more reactive to any changes.

Every topic distributes its own type of message among the standard messages that exist in ROS 2. Below are the message types associated with every topic of our ROS architecture.

Table 4.1: Messages description

Topics	**Messages**	**Purpose**
/joint_states	*sensor_msgs/msg/JointState*	State of the simulation
/motor_commands	*sensor_msgs/msg/JointState*	Configuration required to reach the target
/robot_description	*std_msgs/msg/String*	Description of the robot to visualize it in the simulation
/target_position	*geometry_msgs/msg/Point*	Position to reach
/target_pitch	*std_msgs/msg/Float32*	Pitch to reach
/target_roll	*std_msgs/msg/Float32*	Roll to reach

4.3 Simulation

Simulation takes place thanks to RViz2, a ROS 2 visualisation tool, in which every frame is well defined thanks to the inverse kinematics and the TF included in ROS 2. A TF (for Transformed Frame) is a tool that allows you to define the position of a frame relative to another one. It simplifies a lot the dispositions of the frames in the simulation. Those TF are obtained thanks to the URDF file that initiates them, and then they are actualized thanks to the inverse kinematics algorithm that gives the positions of each joint. The render of this simulation and how we process it are explained in the following part of our GUI description.

4.4 Custom GUI

Thanks to Qt5, we designed a custom GUI (for Graphic User Interface) that allows us to define the desired pose if we want to manually control the arm, see the simulation thanks

to the integration of RViz2 into the GUI, and check the processed image as well (for now this is just the integrated camera of our computer). This interface is an all-in-one interface, allowing us to control the arm as we want.

There were some aspects of this interface that required more work due to their higher difficulty to implement. Integrating RViz2 into this custom interface necessitated a lot of introspection into RViz's API. Unfortunately, the documentation was sparse, so it was challenging to understand which functions, classes, and concepts to use.

To explain quickly how RViz works, it is based on Qt (which facilitates a bit of the integration in our own GUI) and relies on what's called a render_panel. It is sort of the "frame" in which everything happens. It will be this object that will be integrated into our interface. However, this render_panel by itself is not sufficient to have everything printed. We have to instantiate what is called a VisualizationManager, a tool that allows us to add specific displays to our panel. Thanks to this tool, we are able to add RViz default displays that will allow us to see our arm and the TF, more precisely *rviz_default_plugins/RobotModel* and *rviz_default_plugins/TF*. However, if we do only that, we will just see the different frames defined by the TFs and not the model. This is because the *RobotModel* display needs to subscribe to a ROS 2 topic dedicated to providing the description of the robot. Then we use display's property that allows us to subscribe to the topic */robot_description*, which is such a topic.

Finally, we just have to add the tool necessary to be able to move the camera into this panel, and we have a fully custom and customizable RViz2 integration in our interface.

Figure 4.4: Current version of our custom GUI

The particularity of this custom GUI is that it is linked to a ROS node. This is necessary because, as explained above, this interface allows you to interact with the targeted pose. For this, it is necessary to have a connection with the node that handles the publication of this targeted pose, even more so when we consider that we have to choose if we want the arm to be controlled manually or automatically. To do so, a switch button was included in the interface that modifies a public Boolean of the node. This Boolean defines what type of commands we want for our system.

For now, the automated control makes the arm follow Lissajou's curve by keeping it in his range, as we can see in the figure below. This trajectory is just an example; we have to keep in mind that ultimately, it will be the position calculated by the image processing.

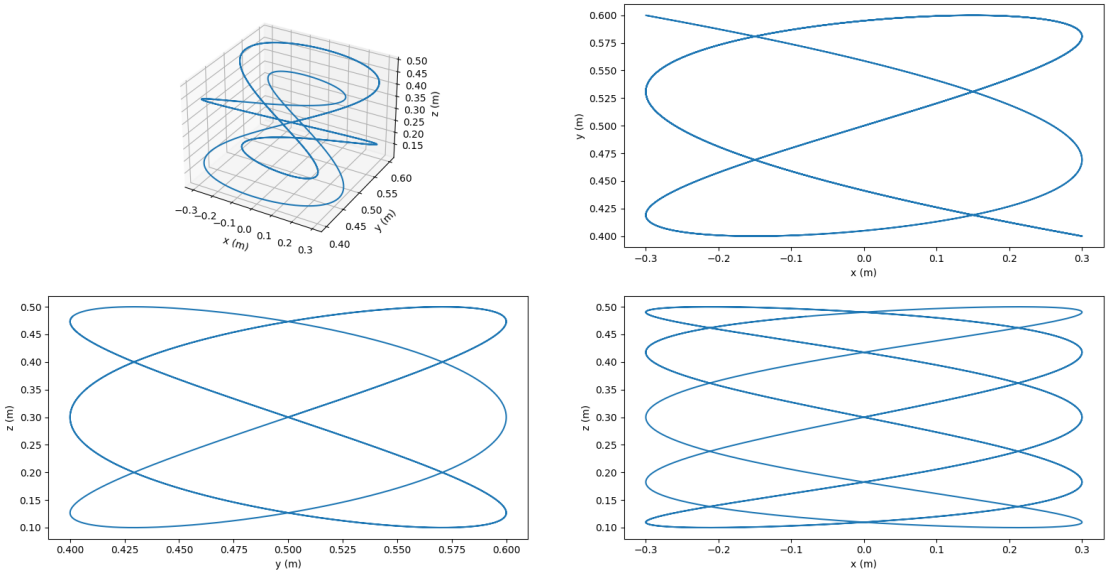

Figure 4.5: Lissajou's curve followed by the arm

Chapter 5

Conclusion

Our progress is interesting. The interface is working efficiently, and the arm responds pretty well to the commands we send through it. More importantly, the simulation is also efficient and well integrated into the GUI.

However, we still have to finish the image processing node; that is a great piece of work and one of the most complex parts of this project. It is great that the detection of the target (the banana plush) gives good results. The method is robust, and the dedicated node is already written and ready to be used. On the other hand, reliably accessing depth is not yet possible. It is good that we managed to write our own code and not use the drivers, which are more complex. But the drawback is that setting the parameters correctly is difficult [9] to understand their influence, and we can't produce good disparity maps, thus depth maps, yet. As examples of improvement points, we could cite: better understanding the format and type of disparity data; correcting the normalisation if need be; pre and/or post-filtering [**?**] the images; computing depth using the disparity-to-depth matrix, computing depth using the ZED M built-in modules, which require working with a strong GPU. Also, the code that has been written to compute depth needs to be written as ROS 2 nodes. The calibration and disparity computation parts are currently in the same code, which means that the camera is calibrated every time the code runs. This is a waste of time, and the ROS 2 interface will have one node for calibration and one node for disparity computation. We also have a code that splits the stereo images in two, which will be added to the disparity computation node. This node will group all the visualisation parts from target detection to depth computation.

Concerning the arm, efficiency can still be improved. When using automated mode, the movement is very jerky due to the time the arm needs to reach a position; it doesn't follow the full trajectory. We have to see whether movement can be smoothed when the target is in motion.

Bibliography

[1] Gary Bradksi Adrian Kaehler. *"Learning OpenCV 3"*. O'reilly media edition, 2016.

[2] Sebastian Van Der Borght. "camera gebaseerde robotsturing". Master's thesis, KU Leuven, 2015-2016.

[3] J. Carpentier, G. Saurel, G. Buondonno, J. Mirabel, F. Lamiraux, O. Stasse, and N. Mansard. "the Pinocchio C++ library – A fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives". In *International Symposium on System Integration (SII)*, 2019.

[4] Xavier Dooms. "camera gebaseerde robotstu-ring d.m.v. ros implementatie met opencv". Master's thesis, KU Leuven, 2014-2015.

[5] Dániel András Drexler. Solution of the closed-loop inverse kinematics algorithm using the crank-nicolson method. In *2016 IEEE 14th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, pages 351–356, 2016.

[6] Stephen Mann Leo Dorst, Daniel Fontijne. *"Geometric Algebra for Computer Science, an object-oriented approach to geometry"*. Morgan Kaufmann, 2007. Chapter 12.

[7] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.

[8] Andrew Zisserman Richard Hartley. *"Multiple View Geometry in computer vision"*. Cambridge, 2006. Chapters 9 to 12.

[9] Kaustubh Sadekar. "stereo camera depth estimation with opencv (python/c++)". 2021.

[10] Hélène Thomas. *Traitement numérique des Images*. Technical report, ENSTA Bretagne, 2 rue François Verny, 29200 France, 2023. Send an email to guillaume.garde@ensta-bretagne.org to access the document.

[11] Hélène Thomas. *Vision par ordinateur*. Technical report, ENSTA Bretagne, 2 rue François Verny, 29200 France, 2023. Send an email to guillaume.garde@ensta-bretagne.org to access the document.

Annex

Stereo calibration algorithm

Stereo calibrating

Input: 5 pairs of stereo images showing different poses of the calibration chessboard; rectified left and right views of the scene.

(i) **Declaring vectors to save the corners' coordinates**: *objectPoints* (3D coordinates of the corners with respect to the top left one for each pair of images), *cornersLeft* (2D coordinates of the corners in the left views of each pair of images), and *cornersRight* (same but for the right views).

(ii) **Declaring pattern size and squarre size**: here *cv::Size patternSize(7,5)* and *float squareSize = 3.1.*

(iii) **For each pair of images**:

  (a) Declare vectors to save 2D coordinates for the left and right view.

  (b) Use *cv::findChessboardCorners()*, which returns *true* if the pattern was found, for the left and right view.

  (c) If corners are detected, use *cv::cornersSubPix()* to refine detection and save the detected points in the associated vectors. Then push these vectors in *cornersLeft* and *cornersRight*. Declare a vector of 3D coordinates (using true dimensions) associated with the pattern with respect to the top left corner and under the form (x,y,0). Push it inside *ObjectPoint*.

(iv) **Declare output parameters and calibrate**: use *cv::stereoCalibrate()*.

Output: a *cv::Mat* disparity map.

A quick description of the stereo block matching algorithms' parameters

Parameter	Description
minDisparity	minimum possible disparity value
numDisparities	maximum disparity minus minimum disparity
blockSize	matched block size
P1	first smoothness parameter for close neighbor pixels
P2	second smoothness parameter for further neighbor pixels
disp12MaxDiff	maximum allowed difference (in integer pixel units) in the left-right disparity check
uniquenessRatio	margin in percentage by which the best (minimum) computed cost function value should "win" the second best value to consider the found match correct
speckleWindowSize	maximum size of smooth disparity regions to consider their noise speckles and invalidate
speckleRange	maximum disparity variation within each connected component

These are the main parameters for the *cv::stereoSGBM()* constructor. The *cv::stereoBM()* constructor only uses *numDisparity*[1] and *blockSize*.

[1] Automatically computed with 0 as minimum value.

URDF description of the arm

Listing 5.1: URDF Description of the arm

```
<?xml version="1.0"?>
<robot name="umi-rtx">

  <link name="base_link">
    <visual>
      <geometry>
        <box size="1.252 0.132 0.091"/>
      </geometry>
      <origin rpy="0 -1.57 1.57" xyz="0 -0.0455 0"/>
      <material name="blue">
        <color rgba="0 0 .8 1"/>
      </material>
    </visual>
  </link>

  <joint name="shoulder_updown" type="prismatic">
    <parent link="base_link"/>
    <child link="shoulder_link"/>
    <origin xyz="0 0.0445 -0.3" rpy="0 0 1.57"/>
    <!-- xyz="0.0445 0 0.134" -->
    <axis xyz="0 0 1"/>
    <limit lower="0.033" upper="0.948" effort="1" velocity="1"/>
  </joint>

  <link name="shoulder_link">
    <visual>
      <geometry>
        <box size="0.278 0.132 0.091"/>
      </geometry>
      <origin rpy="0 -1.57 0" xyz="0 0 0"/>
      <material name="white">
        <color rgba="1 1 1 1"/>
      </material>
    </visual>
  </link>

  <joint name="shoulder_joint" type="revolute">
    <parent link="shoulder_link"/>
    <child link="shoulder_to_elbow"/>
    <origin xyz="-0.01 0 -0.09"/>
    <axis xyz="0 0 1"/>
    <limit lower="-1.57" upper="1.57" effort="1" velocity="1"/>
  </joint>

  <link name="shoulder_to_elbow">
    <visual>
      <geometry>
        <box size="0.252 0.10 0.10"/>
```

```
    </geometry>
    <origin rpy="0 0 0" xyz="0.126 0 0"/>
    <material name="white">
      <color rgba="1 0 0 1"/>
    </material>
  </visual>
</link>

<joint name="elbow" type="revolute">
  <parent link="shoulder_to_elbow"/>
  <child link="elbow_to_wrist"/>
  <origin xyz="0.252 0 -0.07"/>
  <axis xyz="0 0 1"/>
  <limit lower="-3.14" upper="2.64" effort="1" velocity="1"/>
</joint>

<link name="elbow_to_wrist">
  <visual>
    <geometry>
      <box size="0.252 0.10 0.07"/>
    </geometry>
    <origin rpy="0 0 0" xyz="0.126 0 0"/>
    <material name="white">
      <color rgba="1 0 0 1"/>
    </material>
  </visual>
</link>

<joint name="wrist" type="revolute">
  <parent link="elbow_to_wrist"/>
  <child link="wrist_to_wrist_gripper_connection"/>
  <origin xyz="0.252 0 -0.05"/>
  <axis xyz="0 0 1"/>
  <limit lower="-1.92" upper="1.92" effort="1" velocity="1"/>
</joint>

<link name="wrist_gripper_connection_to_gripper">
  <visual>
    <geometry>
      <box size="0.10 0.10 0.05"/>
    </geometry>
    <origin rpy="0 0 0" xyz="0.05 0 0"/>
    <material name="white">
      <color rgba="1 0 0 1"/>
    </material>
  </visual>
</link>

<link name="wrist_to_wrist_gripper_connection">
  <visual>
    <geometry>
```

```
      <box size="0.075 0.10 0.05"/>
    </geometry>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <material name="white">
      <color rgba="1 0 0 1"/>
    </material>
  </visual>
</link>

<joint name="wrist_gripper_connection_roll" type="revolute">
  <parent link="wrist_to_wrist_gripper_connection"/>
  <child link="virtual_link"/>
  <origin xyz="0 0 0"/>
  <axis xyz="-1 0 0"/>
  <limit lower="-3.16" upper="2.3" effort="1" velocity="1"/>
</joint>

<link name="virtual_link"/>

<joint name="wrist_gripper_connection_pitch" type="revolute">
  <parent link="virtual_link"/>
  <child link="wrist_gripper_connection_to_gripper"/>
  <origin xyz="0 0 0"/>
  <axis xyz="0 -1 0"/>
  <limit lower="-1.71" upper="0.07" effort="1" velocity="1"/>
</joint>

<joint name="wrist_gripper_connection" type="fixed">
  <parent link="wrist_gripper_connection_to_gripper"/>
  <child link="gripper_base"/>
  <origin xyz="0 0 0"/>
</joint>

<link name="gripper_base">
  <visual>
    <geometry>
      <box size="0.077 0.15 0.078"/>
    </geometry>
    <origin rpy="0 0 0" xyz="0.10 -0.02 0.012"/>
    <material name="white">
      <color rgba="0.5 0.5 0.5 1"/>
    </material>
  </visual>
</link>

  <link name="left_finger">
  <visual>
    <geometry>
      <box size="0.07 0.015 0.025"/>
    </geometry>
    <origin rpy="0 0 0" xyz="0.06 -0.04 0"/>
```

```
      <material name="white">
        <color rgba="0.5 0.5 0.5 1"/>
      </material>
    </visual>
  </link>

    <link name="right_finger">
    <visual>
      <geometry>
        <box size="0.07 0.015 0.025"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0.06 0.04 0"/>
      <material name="white">
        <color rgba="0.5 0.5 0.5 1"/>
      </material>
    </visual>
  </link>

  <joint name="gripper_left" type="prismatic">
    <parent link="gripper_base"/>
    <child link="left_finger"/>
    <origin rpy="0 0 0" xyz="0.12 0 0"/>
    <axis xyz="0 1 0"/>
    <limit lower="0" upper="0.05" effort="1" velocity="1"/>
  </joint>

  <joint name="gripper_right" type="prismatic">
    <parent link="gripper_base"/>
    <child link="right_finger"/>
    <origin rpy="0 0 0" xyz="0.12 0 0"/>
    <mimic joint="gripper_left" multiplier="-1" />
    <axis xyz="0 1 0 "/>
    <limit lower="-0.05" upper="0" effort="1" velocity="1"/>
  </joint>
</robot>
```