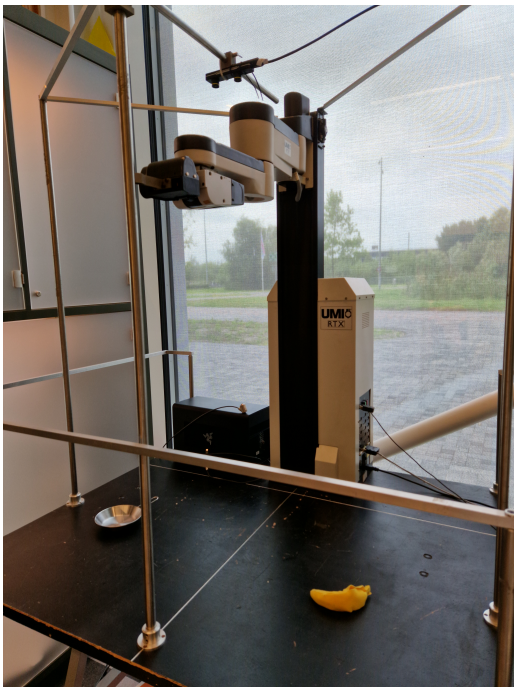


Internship report
A ROS 2 Interface for the UMI-RTX robotic arm

Théo MASSA
ENSTA Bretagne, France
Under the supervision of Arnoud Visser
Intelligent Robotics Lab, Universiteit van Amsterdam, The Netherlands
`https://github.com/gardegu/LAB42_RTX_control`

September 30, 2023

List of Figures

2.1 Model of the arm [3] . . . 7
2.2 Wrist system [7] . . . 8
2.3 Virtual model of the arm . . . 10
2.4 Communication between the arm and the computer . . . 11
2.5 3 ways of communication [3] . . . 11
2.6 Inverse and forward kinematics . . . 12
2.7 Evolution of the error according to iterations . . . 14
2.8 Evolution of the number of iterations according to ϵ in logarithmic scale . . . 14

3.1 The banana plush on the dark horizontal plane that supports the arm . . . 15
3.2 Example of detected banana and its contour with OpenCV . . . 16
3.3 The ZED Mini stereo device . . . 16
3.4 One of the views of the chessboard used to calibrate the ZED Mini device . . . 17
3.5 Detected corners on the precedent left view, associated to the declared pattern . . . 17
3.6 Work scene view and associated rectified image . . . 18
3.7 Left view of the scene and the associated depth map . . . 18
3.8 Left view with target detection on and depth map . . . 20
3.9 3D point cloud of the scene. Front and side view. On the bottom of the front view, one can see the 3D model of the stereo camera . . . 20

5.1 Node graph when running only the simulation . . . 24
5.2 Node graph when running real arm . . . 25
5.3 Current version of our custom GUI . . . 26

Acknowledgement

I would like to deeply thank LAB42 and its team, especially my supervisor, Arnoud, and his colleague, Joey, for their contribution to this work and for the help always provided when needed. This internship has been a great opportunity. It has been a privilege to be welcomed by the University of Amsterdam, a top-rated center of scientific knowledge, and to have access to state-of-the-art technology that has allowed us to go further in my research and to open my horizons of possibilities. I have had great working conditions here and really enjoyed my stay.

Résumé

Le but de ce projet est de travailler avec un vieux bras robotique appelé UMI-RTX (créé dans les années 1980) et de lui faire saisir des objets sur un plan à l'aide de sa pince. Des travaux ont déjà été réalisés sur ce robot par des étudiants, mais principalement avec de vieux outils. L'idée de ce projet est d'implémenter une nouvelle façon de le faire fonctionner et d'utiliser des outils plus récents. Plus précisément, mon objectif est de mettre en place un environnement ROS 2 et de construire une interface qui permettra d'effectuer l'analyse d'images, la planification de trajectoires et la saisie de cibles.

Une banane en peluche a été choisie comme cible. Grâce à la bibliothèque de vision par ordinateur OpenCV et au kit de développement logiciel de Stereolabs (le fabricant de notre caméra), nous avons réussi, dans un nœud ROS 2 dédié, à détecter la banane sur le terrain et à calculer sa position en 3D.

Une fois cela fait, cette information est envoyée à un nœud dédié à la cinématique inverse. Dans ce nœud, les états des articulations nécessaires pour atteindre la pose visée sont traités et envoyés à la fois à une simulation et au bras réel. Pour cela, nous avons deux nœuds, chacun dédié à sa propre partie, l'un pour la simulation, l'autre pour le bras réel.

Enfin, nous avons conçu une interface utilisateur graphique (GUI) personnalisée dans laquelle la simulation, l'image traitée et la carte de profondeur sont intégrées, et dans laquelle nous pouvons choisir entre le contrôle automatique du bras et un mode manuel, dans lequel nous pouvons choisir notre propre pose cible.

Grâce à ce processus, le bras est maintenant capable de suivre et d'attraper la banane. Nous pouvons afficher le résultat dans notre interface graphique et choisir de manipuler le bras ou de le laisser suivre le processus automatique.

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 this project is to implement a new way of making it work and to use more recent tools. More specifically, my goal is to set up a ROS 2 environment and build an interface that will allow to perform image analysis, trajectory planning, and target grabbing.

A plush banana has been chosen as a target. With the computer vision library OpenCV and the software development kit of Stereolabs (the manufacturer of our camera), we managed, in a dedicated ROS 2 node, to detect the banana on the field and compute its 3D position.

Once this is done, this information is sent 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, processed image and the depthmap 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 pose.

Thanks to this process, the arm is now able to follow and grab the banana. We can display the result in our GUI and choose to manipulate the arm or let it go through the automatic process.

Contents

1 Introduction **6**
- 1.1 Context . . . 6
- 1.2 Objectives . . . 6
- 1.3 Contribution . . . 6

2 Arm manipulation **7**
- 2.1 Description . . . 7
- 2.2 URDF description . . . 8
- 2.3 Communication with the arm . . . 10
- 2.4 Inverse kinematics . . . 12

3 Computer vision (*Guillaume Garde's work*) **15**
- 3.1 Detection of the target in a horizontal plane . . . 15
- 3.2 Generating depth with stereo vision using OpenCV . . . 16
  - 3.2.1 Parameters . . . 17
  - 3.2.2 Calibrating the stereo camera . . . 17
  - 3.2.3 Stereo rectification . . . 18
  - 3.2.4 Disparity map computation . . . 18
- 3.3 Generating depth with Stereolabs' SDK . . . 19
  - 3.3.1 Stereolabs . . . 19
  - 3.3.2 Using Stereolabs' software development kit . . . 19
  - 3.3.3 Integration into the project . . . 19

4 Docker image **21**
- 4.1 Docker . . . 21
- 4.2 Necessity of Docker . . . 21
- 4.3 Building the bespoke project's image . . . 22

5 ROS 2 Interface **23**
- 5.1 Configuration and ROS 2 presentation . . . 23
- 5.2 ROS 2 Architecture . . . 24
- 5.3 Simulation . . . 25
- 5.4 Custom GUI . . . 26

6 Conclusion **28**

Chapter 1

Introduction

1.1 Context

This project is led in an internship context, mainly for educational purposes, at Lab42 in the University of Amsterdam, the Netherlands.
It is important to notice that this internship was done in collaboration with Guillaume Garde, another ENSTA Bretagne student in robotics. Therefore, we worked together on this project. However I will focus more on what I did myself and only explain Guillaume's work without going too much in details as it is important too for the whole purpose of the internship. The way less detailed part concerning Guillaume's work will be indicated.
The subject of this paper is an old industrial arm, the UMI-RTX, which was created in the 1980s [7][8]. Despite his age, the arm is still compatible with recent software and hardware. Its educational appeal is obvious when we consider that a lot of work has already been done on it, especially by Van Der Borght [1] and Dooms [3] who worked on a ROS 1 interface in order to control it.

1.2 Objectives

The objectives of this project are plural. First, it is to improve our ROS 2 skills and learn new knowledge. We have studied, on various levels, a wide scope of robotic disciplines like computer vision, simulation, C++, network, and embedded Linux. Our goal, shared by ENSTA Bretagne, our school, is to consolidate them. As students, this kind of internship is strong in apprenticeships. Then, more importantly, the main objective of this project is to create a ROS 2 interface in order to be able to control the arm. The robot should be able to detect a recognizable object, a yellow banana plush in our case, move to this object, and grab it. To this extent, we have to rely on previous works [1][3] and create, from not much except the hardware drivers, a full ROS 2 interface.

1.3 Contribution

This project has been split in two. Guillaume's objective was to manage the computer vision part in order to succeed in our objectives. I personally worked on the arm manipulation, created a simulation, managed the ROS 2 architecture and created a working custom GUI that will make all those aspects work together. The both of us have worked together on building the Docker image for this project.

Chapter 2

Arm manipulation

2.1 Description

This project uses the UMI-RTX arm, which is quite basic in its composition [7]. 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, they are only controled 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 [7] that allow it to be controlled and know its state.

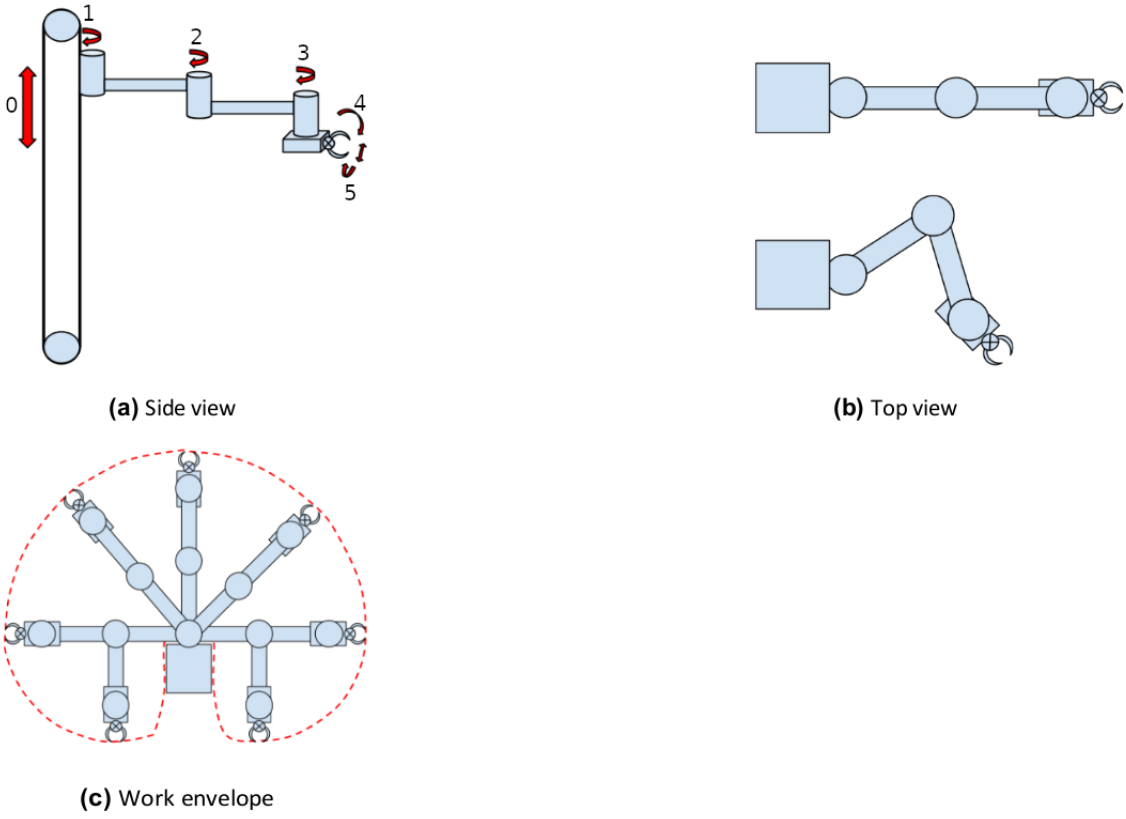


Figure 2.1: Model of the arm [3]

As shown in *Figure 2.1*, this arm can be compared to a human arm. 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 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 typical 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, causing two rotation axis at the same origin. A view of this system can be seen in the following figure:

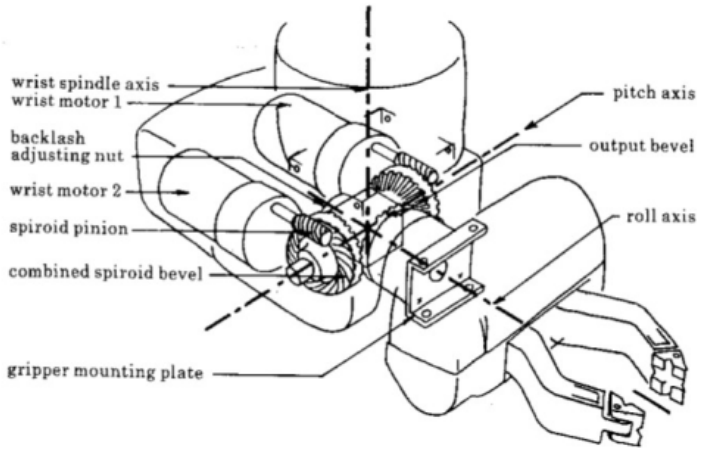


Figure 2.2: Wrist system [7]

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**.

2.2 URDF description

For practical purposes, it is really useful, even mandatory, to have a digital twin of the arm [5]. 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. Having a virtual clone of our system is always something vital, and it is becoming increasingly sought after, especially in robotics [5].

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 on Appendix C [1].

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 *section 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, it was said before that two motors worked together to handle those angles. Therefore, one have to be careful when converting this

[1] The URDF description comes from here :
https://github.com/LHSRobotics/hsrdp/blob/master/hsrdp_bridge/umi_rtx_100.urdf

description into reality. The conversion formulas are:

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

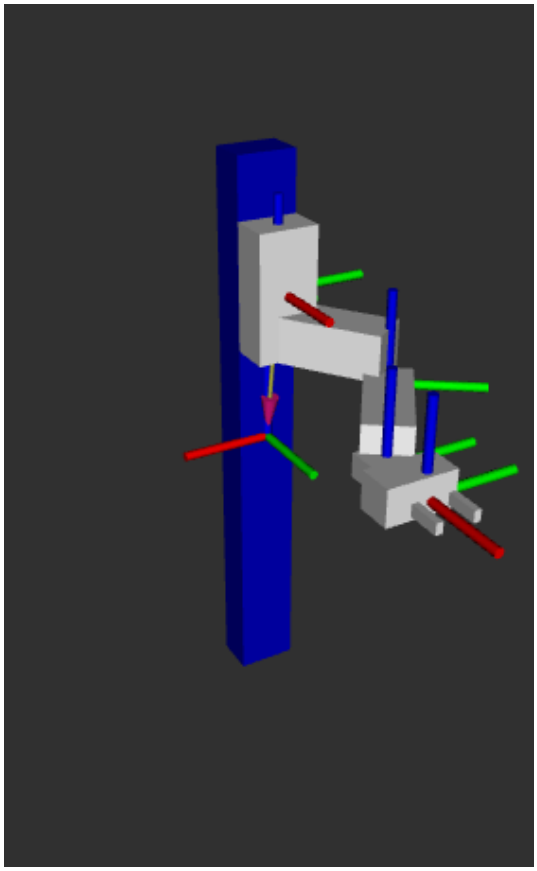


Figure 2.3: Virtual model of the arm

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 [8][7], 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, previously developed drivers were at our disposal. Thanks to our supervisor A. Visser, a fully developed hardware driver that communicates with the arm, furnished by him, is available.
However, this driver is one that allows to control the arm only via the terminal, when the purpose is to build an interface that does not just happen via the terminal. The reason of

this is that it is more intuitive to have a dedicated interface instead of controlling through the terminal. See *section 5.4* for more details. For this, one had to look into the source code of the drivers, understand how they manage to communicate with the arm, and reuse their functions in its own code.

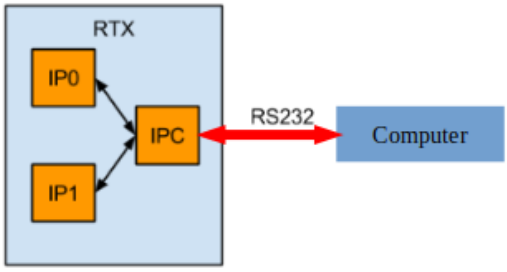


Figure 2.4: Communication between the arm and the computer

One particularity of the arm, is that the motors are controlled by two 8031 chips, IPs[2] called. Each IP ensures the proper operation of a selection of motors. Table 2.2 shows 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 [3]. 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

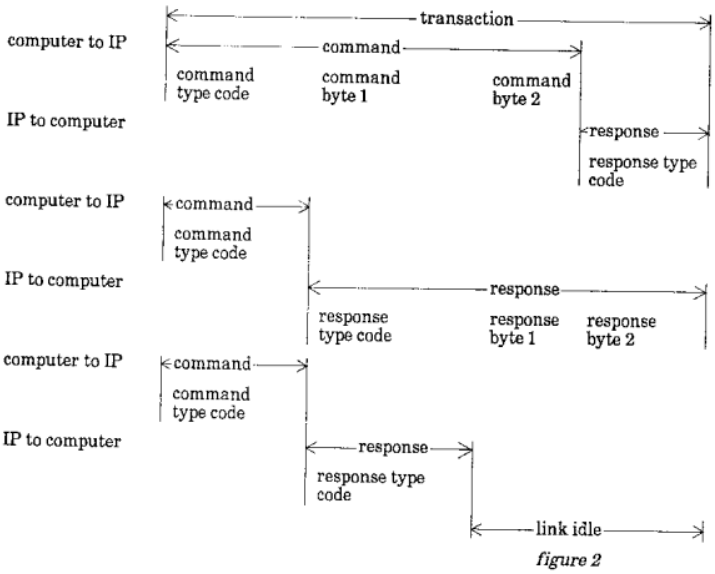


Figure 2.5: 3 ways of communication [3]

[2]Intelligent Peripheral

In order to use the arm, a precise procedure have to be followed. First, one have to start communication with the arm by launching the daemon and specifying which USB port is used by the arm. Then, in our code, the communications are initialized by sending a certain command to the arm, and every time the arm is used, an initialization procedure has to be processed for the arm to know what 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.

After the initialization process, everything is ready to command the arm. All that's left is to use the predefined commands to set motors' positions in memory and then go to those position. It is indeed a two-step process to command the motors.

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 the end-point pose given the state of every joint is processed, the opposite here is done here.

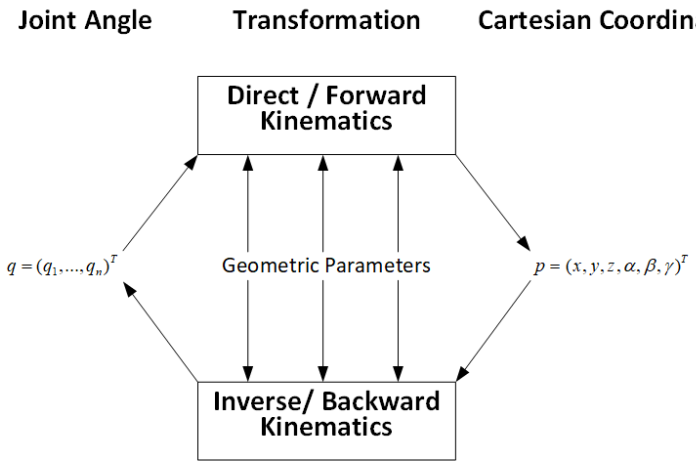


Figure 2.6: Inverse and forward kinematics

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 computation is a bit simplified.

To compute those inverse kinematics, a C++ library named Pinocchio [2] was used, 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, a method called CLIK, for Closed-Loop Inverse Kinematics [4], was chosen, 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.

The different rotation matrices are defined 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 $SE(3)$ space, defined by the desired position and R, the desired rotation.

Then one can 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 then 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 all those parameters are set, the iterative process can begin.

- First, the forward kinematics with the current configuration defined by the vector q is computed.

- Then, the transformation $T \in SE(3)$ between the current pose and the desired one is calculated and the logarithmic error computed, which is defined by :

$$err = log(T)$$

- if $||err|| \leq \epsilon$ with ϵ a defined coefficient that characterises the precision wanted, or if a certain amount of iteration occurred, the iterative process is stopped

- Else, the Jacobian J of the current configuration is computed.

- The vector v is defined 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 as the speed vector that will get the configuration closer to the desired one.

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

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

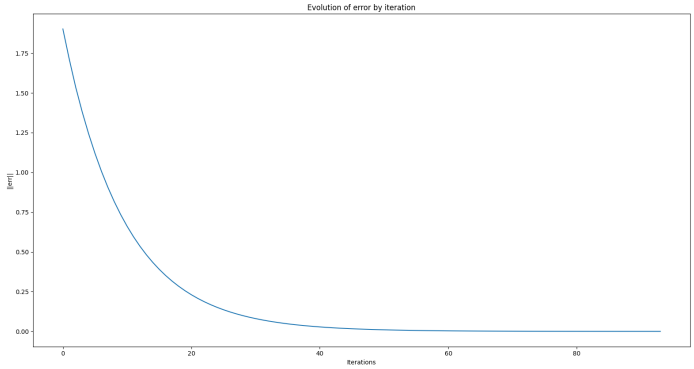


Figure 2.7: Evolution of the error according to iterations

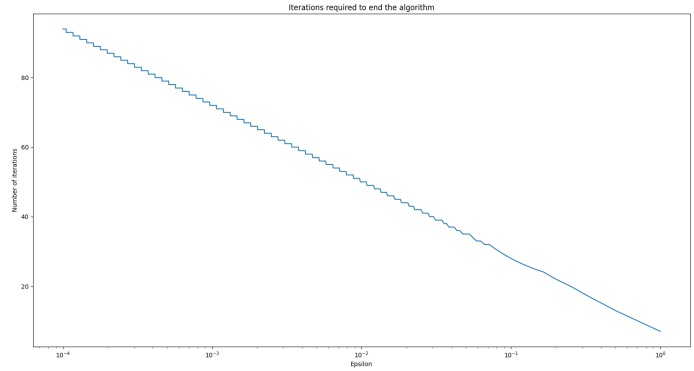


Figure 2.8: Evolution of the number of iterations according to ϵ in logarithmic scale

As one can see in the 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 ϵ, 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.
See *Chapter 4* to see how the inverse kinematic calculation is integrated into our ROS 2 architecture.

Chapter 3

Computer vision (*Guillaume Garde's work*)

To make a robotic arm grab a target, a strong starting move is to rely on one key element: computer vision [11]. 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 a target and get its 3D position in the camera's frame. However due to the fact I haven't personnaly worked deeply on this aspect, so I won't go into details, I will only look through the main aspects and results of Guillaume's work, in order to keep this essential aspect of the project and respect that this part is not mine.

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; its colour is convenient to detect; its softness makes it easy for a gripper to grab it; and it 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[12]. 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 [12].

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, to extract objects in between. Then one can perform contour detection on these objects.

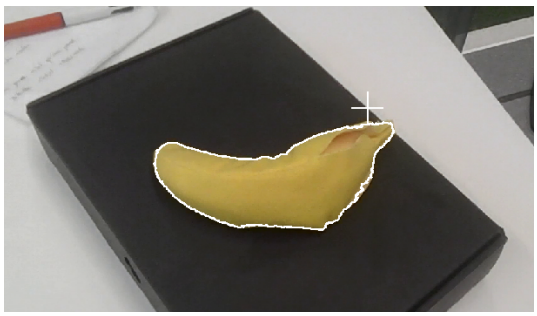


Figure 3.2: Example of detected banana and its contour with OpenCV

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 using OpenCV

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 [6]. This is called stereo vision. The stereo device used in this project is the ZED Mini camera device from *StereoLabs*[1].

Figure 3.3: The ZED Mini stereo device

The main idea behind stereo vision is to reproduce human vision [13]. One will use the difference in perception of the scene to extract depth information.

[1] https://www.stereolabs.com/zed-mini/

3.2.1 Parameters

One needs to know some parameters associated with the scene and the camera in order to access depth information. The first type of parameters are the *intrinsic* parameters [13], which are internal to the camera meanwhile the second type of parameters are the *extrinsic* parameters [13], which links the scene reference frame to that of the camera. The third type of parameter is matrices that won't be described here. Accessing depth information can only be done through the determination of these parameter.

3.2.2 Calibrating the stereo camera

The first step of the process is to calibrate the ZED Mini device in order to compute the extrinsic parameters, the fundamental matrix and the camera matrices [6] [13]. 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 in different poses to guarantee robustness.

Figure 3.4: One of the views of the chessboard used to calibrate the ZED Mini device

One has to declare the inner pattern that will be searched by the algorithm. It is very important to correctly count the inner corners for the algorithm to work. It is also necessary to provide the algorithm with the size of a square.

Once this is done, the 2D positions of the corners in the left and right images are saved and, for each pair of images, the 3D real coordinates of the corners with respect to the top left corner with no knowledge of the depth yet, are saved too. Finally, thanks to dedicated OpenCV function, it is possible to process the wanted parameters.

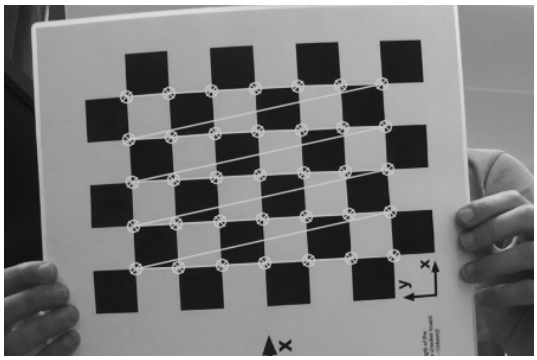


Figure 3.5: Detected corners on the precedent left view, associated to the declared pattern

3.2.3 Stereo rectification

The next part is to correct the distortion. To do so, one must use the parameters found during the previous step. Then one can remap the images to be rectified. This must be applied once for each view. On the following images is an image of the work scene taken by the left camera and its associated rectified image. One can see on the bottom image that the curved lines have been corrected into straight lines, and now everything is set to work on disparity.

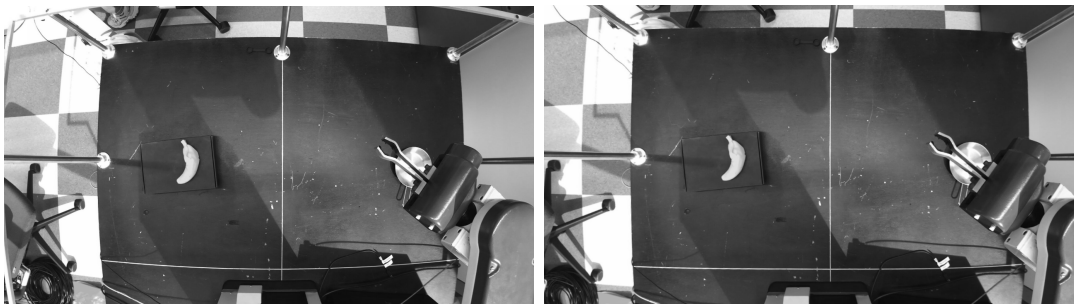


Figure 3.6: Work scene view and associated rectified image

3.2.4 Disparity map computation

Stereo vision reproduces human vision [13] [6]. Each camera of the stereo device will perceive the scene in its own way, just like the human eye. 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, and is called *disparity* [13] [6]. Computing disparity for every object in the scene allows for the creation of a disparity map. Moreover, considering that depth and disparity are inversely proportional, one can access a depth map with a disparity map.

OpenCV provides algorithms to compute disparity between two stereo-associated views. One thing important to remember, though, is that these algorithms are sensitive to texture. Hence it is necessary to make them as robust as possible before testing above the planar support of the UMI-RTX. The following pictures illustrate the process. On the first one, one can see an extraction from the rectified left view, on which were kept objects that are seen by both cameras and the disparity map obtained on the second one.

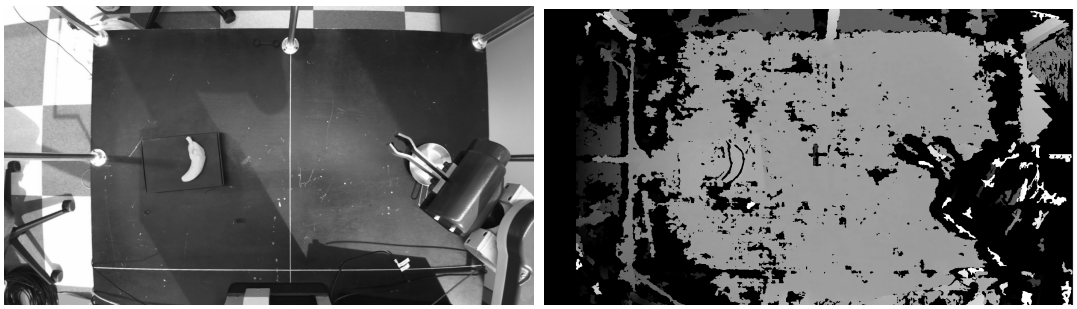


Figure 3.7: Left view of the scene and the associated depth map

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 disparity maps made during

the internship with this method, despite being far from perfect.

3.3 Generating depth with Stereolabs' SDK

The depth map obtained with OpenCV alone is too imprecise yet. The chosen process is coherent, but adjusting parameters to get valuable and useful 3D information is too complex. One has the opportunity to improve this system by using a far better solution. *Section 3.2* shows how to perform depth computing only by using OpenCV and letting aside Stereolabs' drivers. However, considering the fact that this project uses a Stereolabs' device, one can choose to try and work with their software development kit (SDK). It has built-in tools that enable the user to access trustworthy and quality information.

3.3.1 Stereolabs

Stereolabs is a French company based in Silicon Valley. It is a world leader in the use of stereo vision and artificial intelligence to provide 3D depth and motion sensing solutions. It sells stereo cameras, software, and embedded PCs for 3D perception and AI. This project works with one of their cameras, the ZED Mini, initially designed for mixed reality, and their software development kit.

3.3.2 Using Stereolabs' software development kit

The SDK[2] is a toolbox for creating software on a particular platform. It's packed with tools to build software, spot and solve issues, and often includes pre-made code that's perfect for that platform's operating system. It allows the user to use built-in tools and their associated graphic user interfaces, such as a simple stereo viewer or a depth viewer with 3D scene reconstruction.

A choice of a SDK was necessary considering that there is no general package provided for stereo cameras. Besides, it allows the user to access valuable information with great precision, which could not be accessed before. Among the provided data are the precise focal lengths (see Appendix A), which are needed to compute the real positions in the plan, horizontal and vertical, in metric units from the computed depth. The last release of the SDK uses an NVIDIA library called CUDA that causes the necessity of having a computer with great graphic processing capabilities. Having PCs that are not powerful enough to use its tools and do not have graphic processing units (GPUs) will be a problem. This is why using the ZED SDK is really a new step in this project compared to simple OpenCV-based stereo vision. From here, it is necessary to start working with a very powerful computer that has everything needed to continue the work.

3.3.3 Integration into the project

It is a good choice to use the SDK if one needs something precise and efficient to compute depth. The idea behind this choice is to keep the structure of *Section 3.2* and adapt it to

[2] https://www.ibm.com/blog/sdk-vs-api/

the SDK's tools. The advantage of doing so is that the code is much lighter and clearer now. There is, indeed, no use for a calibration or rectification part. Thanks to the SDK, one can use the stereo camera nearly directly at full capacity.

One can see below the depth map obtained for our system. After some convenient conversions, those images can now be included in the ROS 2 architecture described later.

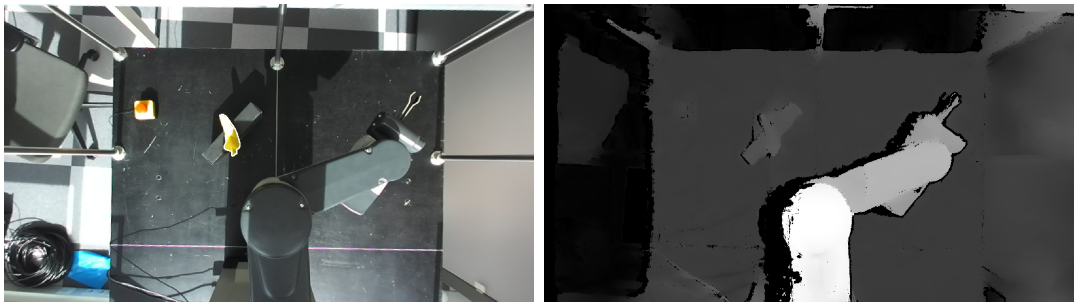


Figure 3.8: Left view with target detection on and depth map

Those figures are useful to make a fair comparison with the images from *Section 3.2.4*. On it, one can see the project's work space with the robotic arm and the banana. On the first one, it is clear that the banana has been detected and its contour drawn. On the second one, one can see the UMI-RTX, brighter than the rest of the image, above its planar support (much darker). There lies the banana target on a box.

Afterwards, the process is similar. The target detection is identical to *Section 3.1*, but the acquisition of its orientation has been added to it (yaw, pitch, and roll).

Then, once the contour of the banana and the best fitting line have been found, one can find an orientation for the target. At this point, one also has the coordinates of its centroid. This is where the 3D point cloud comes in.

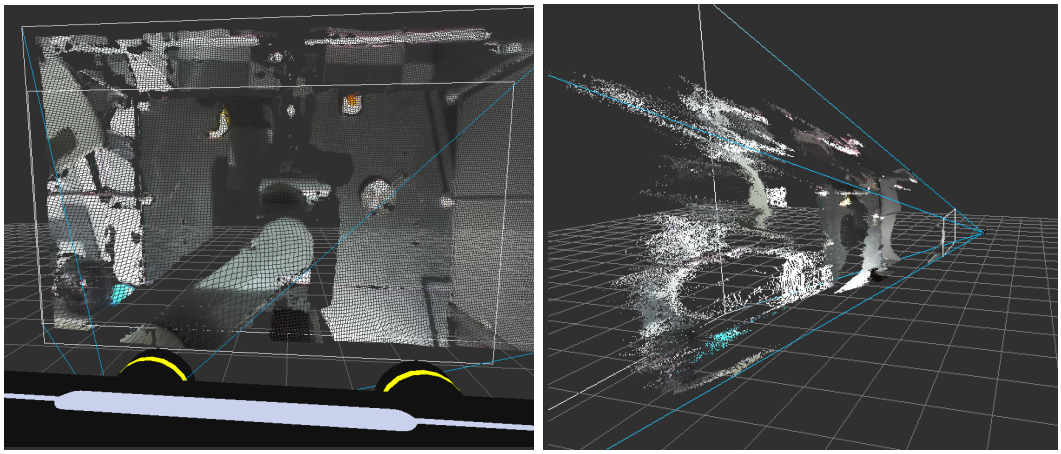


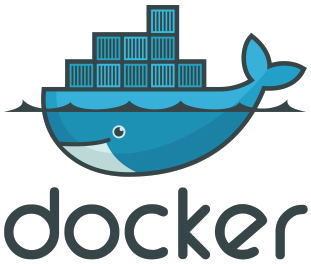
Figure 3.9: 3D point cloud of the scene. Front and side view. On the bottom of the front view, one can see the 3D model of the stereo camera

As one has the coordinates in pixels of the target's centroid, one can find the corresponding metric coordinates in the point cloud. Then one has the 3D coordinates in millimeters of the banana. But they are in the reference frame of the left camera. Hence, it is necessary to add offsets with respect to the origin of the world frame of the UMI-RTX. At this point, one just has to publish the coordinates of the target.

Chapter 4

Docker image

4.1 Docker



Docker [1] is a lightweight and open-source containerization platform that allows developers to package applications and their dependencies into isolated, standardized units called containers. These containers can then be easily transported between different systems, ensuring consistent behavior across diverse environments, from development to production [10].

4.2 Necessity of Docker

As described in *Section 3.3*, a powerful computer with a Nvidia GPU was needed to run the application. This computer, kindly furnished by our supervisor, came with a Linux installation. However, the version of Ubuntu installed wasn't the same as the one we worked with until now. This computer came with the Ubuntu 22.04 distribution, while we worked on Ubuntu 20.04. This difference is important because the version of ROS 2 differs according to the Ubuntu distribution, as well as the way the code has to be done. This means our work is not compatible with an Ubuntu 22.04 distribution, more precisely with ROS 2 Humble.

To solve this issue, it was decided to create a Docker image of an Ubuntu 20.04 installation that would take advantage of the hardware of the computer. This containerization allows anyone who has Docker to use our project. They don't need a ROS installation any more, just to install Docker. As a result, this makes our project really portable. The only thing is that it requires a NVIDIA graphic card for the reasons explained in *Section 3.3*.

[1] https://www.docker.com/

4.3 Building the bespoke project's image

When it comes to creating a new Docker image, one has to understand an important thing. There is a "bank" of basic distribution that can be downloaded and used as a starting point for its own image. For example, any distribution of Ubuntu or Windows can be used as the base, and any additional layers can be installed with a script and commands.

For our image, there were two options. We could build an image from a fresh Ubuntu 20.04 distribution and install every driver, package, and other stuff manually, or we could use a predefined distribution furnished by Stereolabs and build around it. Our first try was to do it fully manually with the first option.

However, this decision was probably the wrong one because there was an issue when the ZED SDK was added to our image. Files were missing, and drivers refused to work. For that reason, it was decided to rely on an image furnished directly by Stereolabs, which was an Ubuntu 20.04 distribution with the SDK already installed on it. Once it was done, all that was left to do was install ROS2, download our project through Git, and follow the procedure in order to install it correctly. Our installation script can be found in Appendix D.

One particularity of this image is that the command to launch it is special. It needs several privileged accesses to the computer hardware. It needs access to the GPU for image processing and the screen to launch the GUI. For this, one has to give it privileged access when launching with docker tags: *–gpus all -it –privileged -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix* and also opening the screen to usage with **xhost** + in the terminal.

Chapter 5

ROS 2 Interface

5.1 Configuration and ROS 2 presentation

For this project, Ubuntu 20.04 and ROS 2 Foxy were used.

ROS 2 [1] (Robot Operating System 2) is an open-source framework designed for building and controlling robotic systems [9]. 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 [9].

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 [9].

[1] https://docs.ros.org/en/foxy/index.html

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 [9].

5.2 ROS 2 Architecture

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

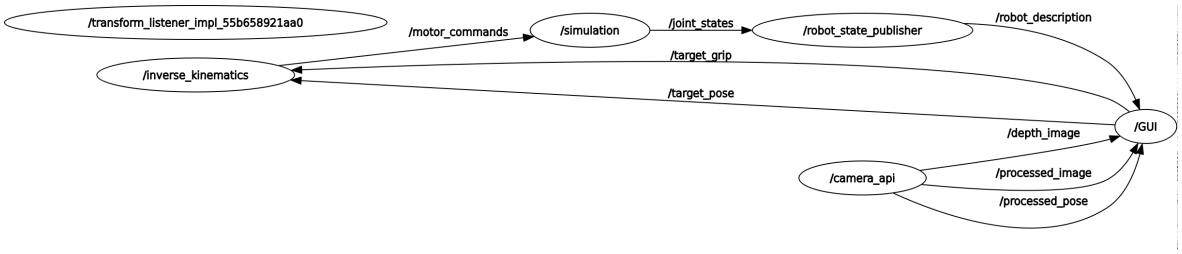


Figure 5.1: Node graph when running only the simulation

Here, the central node is */GUI*. This node is the one where the GUI is executed and where information is centralized. It sends the targeted pose to the */inverse_ kinematics* node that will process the inverse kinematics 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 *section 5.4*. This closed loop allows to control the arm as wanted.

In this system, one has to be careful how he deals with the yaw. This is because for the real arm, the yaw is equal to 0 when it is in reality equal to $arctan2(y, x)$ where x and y are the coordinates of the end-effector. To be clearer, in simulation, the yaw origin is the y-axis, whereas for the encoders, the yaw origin and neutral point follow the axis between the zed-axis and the wrist. Because of this, one has to be careful to avoid confusion with the yaw value.

Concerning the node handling the stereovision, which is */camera_ api*, the ZED SDK described earlier was used. Our working version of the interface fully uses the SDK, and one can see that it sends lots of information, like the desired pose or two images, the depth map, and an image where the object is surrounded. This data is sent to *GUIw* which is the intermediary between the data and the simulation and/or the arm.

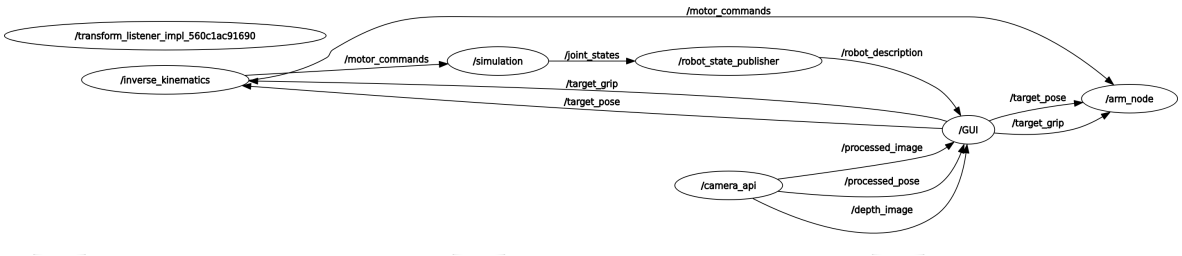


Figure 5.2: Node graph when running real arm

For the real arm, the node graph is a bit more complicated due to 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, useless calculations are avoided. Indeed, if commands are sent 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 5.1: 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_pose	*geometry_msgs/msg/Point*	Pose (position & orientation) to reach
/target_grip	*std_msgs/msg/Float32*	Grip to reach
/processed_pose	*geometry_msgs/msg/Point*	Pose processed by the computer vision
/processed_image	*sensor_msgs/msg/Image*	Image where the object is shown
/depth_image	*sensor_msgs/msg/Image*	Depth map

5.3 Simulation

Simulation takes place thanks to RViz2, a ROS 2 visualization 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) provides a convenient way to manage the frames relative positions by maintaining a tree-like structure of frames and managing the transformations between them. It helps to seamlessly convert coordinates from one frame to another and handle various operations involving transformations and coordinate systems. 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 rendering of this simulation and how it is processed are explained in the following part of our GUI description.

5.4 Custom GUI

For the interface, several options were available. It could just have been a terminal-driven interface, but this choice reduced the possibilities and was really not adapted to our goal of fully controlling the arm. There was also the possibility of using external software like Foxglove [2] that would be adapted to this interface.
However, building our own GUI was the preferred choice because it offered more control, and it was quite interesting to build our own Graphic Interface. It was preferred to centralize all the information in one interface that is simple to use and autonomous.

Thanks to Qt5 [3], a custom GUI (for Graphic User Interface) was designed, one that allows us to define the desired pose if we want to manually control the arm, see the simulation through to the integration of RViz2 into the GUI, and check the processed image or the depth map as well. 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 quickly explain how RViz works, it is based on Qt (which facilitates the integration into 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. What is called a VisualizationManager, a tool that allows us to add specific displays to our panel, has to be instantiated. Thanks to this tool, one is able to add a 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 only this is done, only the different frames defined by the TFs and not the model will be seen. This is because the *RobotModel* display needs to subscribe to a ROS 2 topic dedicated to providing the description of the robot. Then a display's property that allows us to subscribe to the topic */robot_description*, which is such a topic, is used.

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

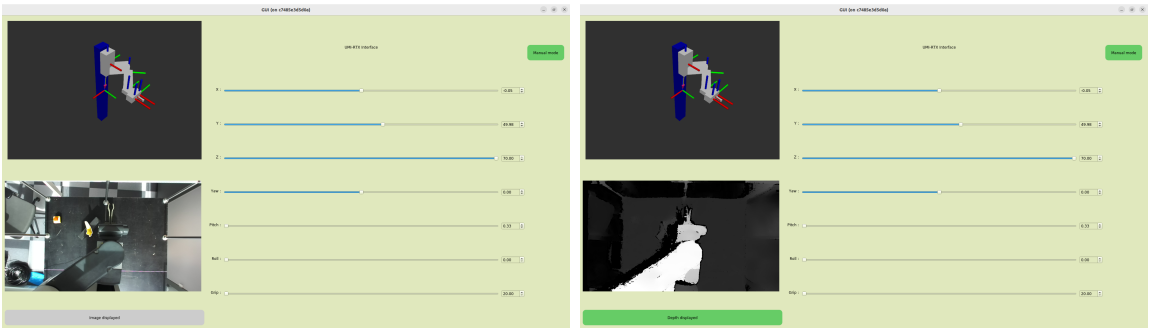


Figure 5.3: Current version of our custom GUI

[2] https://foxglove.dev/
[3] https://doc.qt.io/qt-5.15/

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 the arm has 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 are wanted for our system.
Thanks to "*Manual mode*", one can control the arm with the sliders displayed on our interface. If the button is clicked, it switches to "*Grab mode*", in which the arm follows a predefined procedure: it goes to where the object is, grabs it, displays the banana, and then puts it at another place. The algorithm works, but there are still minor issues. Because of the lack of precision, sometimes the grip misses the banana, but it is always very close. Despite that, the planned trajectory is well followed; one just has to be careful not to plan the arm to move too fast because the system is really limited by the motors' speed.

In order to prove this assumption, a series of tests have been conducted. The test was to launch the automatic mode from a random position of the arm and of the target. The only condition was that the arm had to be above the target. It resulted that 63% of the attempts to realise our algorithm were successful and in most of the missed attempts, the arm was close to the target and it missed by not much. This can be the consequence of a lack of precision in our algorithm. More, when the arm grabbed the target correctly, it was put correctly at its end pose (a predefined position) on 85% of the tries. Those success rates can be considered sufficient for the time put on it.

Chapter 6

Conclusion

Our progress during this internship has been interesting. Our purpose was to create a ROS 2 interface that could make the UMI-RTX grab a simple object using computer vision and inverse kinematics. We are proud of the work accomplished during these past few months. It was quite a challenge to gather state-of-the-art and classic technology.

That does not mean there is no work left to be done. Our interface can be improved. Nonetheless, it works efficiently. We managed to get a strong computer vision process, and the arm responds well to the commands we send through it. The simulation is also efficient and well integrated into the GUI, which allows us to gather all the useful commands and data. More concretely, we succeeded in using stereo vision to access the target's 3D position with respect to the UMI-RTX's world frame; in converting these coordinates into orders for the arm to follow a calculated trajectory; and in grabbing the target and lifting it to another location. In that aspect, we can say that we have found a way to accomplish our mission.

It is also important to mention the work we did with Docker. Creating a custom Docker image was quite challenging, but we succeeded in doing it, making the project way more accessible now. This custom image has several advantages. First, it allows everyone who has a Nvidia GPU and Docker installed to use the arm; Ubuntu and ROS are no longer mandatory to do so. Secondly, it also allows us to compile and sum up in a file what needs to be done in order to make the utilization successful. The only counterpart is that we didn't manage to do a fully self-built custom image. But in the end, it works really well, and we are quite satisfied with the work we did on it.

As more general improvements, we could work on the threshold settings that enable the code to detect the targeted banana. Depending on the light in the laboratory, there can be some noise that makes the banana go undetected. Concerning the arm, efficiency can still be improved. When using automated mode, the movement is quite jerky due to the time the arm needs to reach a position. However, when we try to have a full view of our work, it seems we have globally accomplished what we wanted to do, despite minor aspects that still need some work.

We are also very satisfied with all that we have learned. We have consolidated our ROS 2 skills, improved our Docker knowledge, and learned a lot about computer vision and creating a custom GUI. We are grateful for all of this.

Bibliography

[1] Sebastian Van Der Borght. Camera gebaseerde robotsturing. Master's thesis, KU Leuven, 2015-2016.

[2] 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.

[3] Xavier Dooms. Camera gebaseerde robotsturing d.m.v. ros implementatie met opencv. Master's thesis, KU Leuven, 2014-2015.

[4] 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.

[5] Aidan Fuller, Zhong Fan, Charles Day, and Chris Barlow. Digital twin: Enabling technologies, challenges and open research. *IEEE Access*, 8:108952–108971, 2020.

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

[7] Universal Machine Intelligence Ltd. *Inside RTX: Guide to the Design, Mechanics and Electronics.* April 1987.

[8] Universal Machine Intelligence Ltd. *Maintenance Manual for RTX*, August 1987.

[9] 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.

[10] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[11] P.K. Shukla, K.P. Singh, A.K. Tripathi, and A. Engelbrecht. *Computer Vision and Robotics: Proceedings of CVR 2022.* Algorithms for Intelligent Systems. Springer Nature Singapore, 2023.

[12] Hélène Thomas. Traitement numérique des images. Technical report, ENSTA Bretagne, 2 rue François Verny, 29200 France, 2023. Private communication.

[13] Hélène Thomas. Vision par ordinateur. Technical report, ENSTA Bretagne, 2 rue François Verny, 29200 France, 2023. Private communication.

How to cite this report

Do not forget to cite this report if it was of any help in your project. Here are the BibTeX corresponding citing lines:

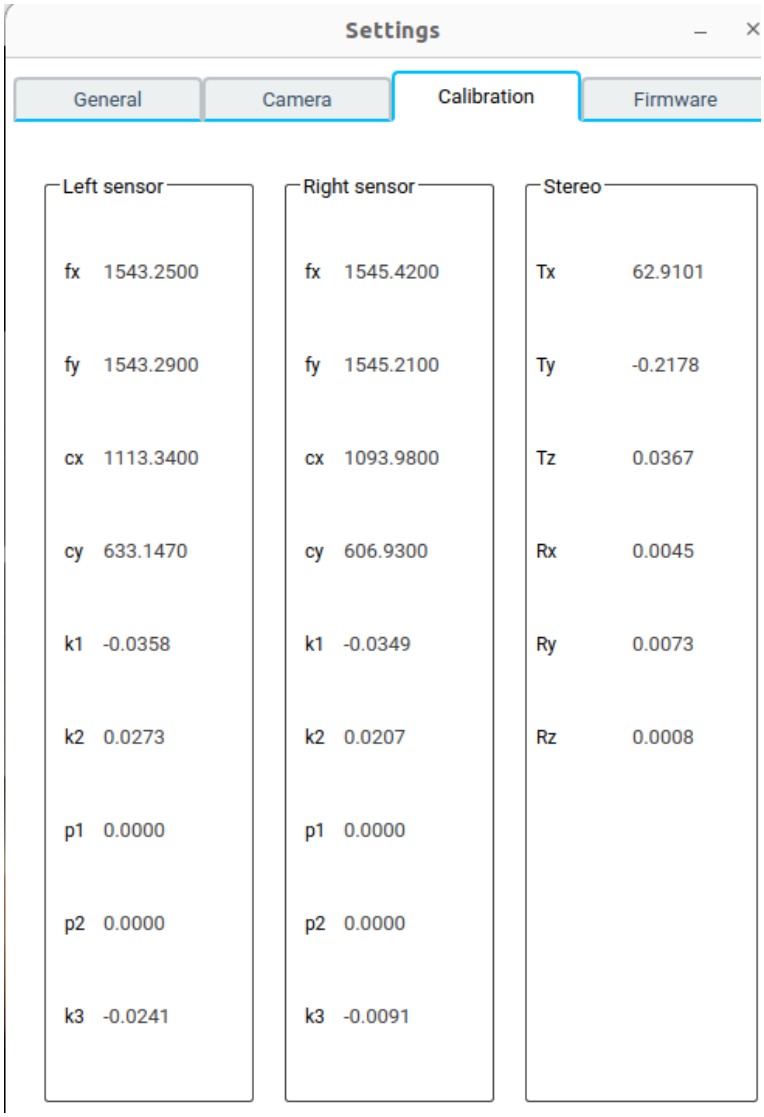
```
@mastersthesis{Massa2023,
    title={A ROS 2 Interface
    for the UMI-RTX robotic arm},
    author={MASSA, Theo},
    school={ENSTA Bretagne and the University of Amsterdam},
    year={2023}
}
```

Otherwise, you can use this APA format:

```
Massa, T. (A. 2023). A ROS 2 Interface for the UMI-RTX robotic arm. The
    University of Amsterdam, Amsterdam, the Netherlands.
```

Appendices

A. Some camera parameters given by the SDK



B. Converting images from Stereolabs format to OpenCV format

Listing 6.1: format conversion method from Stereolabs to OpenCV

```
cv::Mat Camera_API::slMat2cvMat(sl::Mat& input){
    return cv::Mat(input.getHeight(), input.getWidth(), getOCVtype(input.
        getDataType()), input.getPtr<sl::uchar1>(sl::MEM::CPU), input.
        getStepBytes(sl::MEM::CPU));
}

int Camera_API::getOCVtype(sl::MAT_TYPE type){
    int cv_type = -1;
    switch (type) {
        case MAT_TYPE::F32_C1: cv_type = CV_32FC1; break;
        case MAT_TYPE::F32_C2: cv_type = CV_32FC2; break;
        case MAT_TYPE::F32_C3: cv_type = CV_32FC3; break;
        case MAT_TYPE::F32_C4: cv_type = CV_32FC4; break;
        case MAT_TYPE::U8_C1: cv_type = CV_8UC1; break;
        case MAT_TYPE::U8_C2: cv_type = CV_8UC2; break;
        case MAT_TYPE::U8_C3: cv_type = CV_8UC3; break;
        case MAT_TYPE::U8_C4: cv_type = CV_8UC4; break;
        default: break;
    }
    return cv_type;
}
```

C. URDF description of the arm

Listing 6.2: 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>
```

D. Dockerfile

Listing 6.3: Dockerfile of our image

```
FROM stereolabs/zed:4.0-gl-devel-cuda11.4-ubuntu20.04

ARG DEBIAN_FRONTEND=noninteractive
SHELL ["/bin/bash", "-c"]

ENV USER=root
# Setlocale
RUN apt update && apt install locales
RUN locale-gen en_US en_US.UTF-8
RUN update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
RUN export LANG=en_US.UTF-8
RUN apt install software-properties-common -y
RUN add-apt-repository universe

RUN apt update && apt install curl -y
RUN curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o
    /usr/share/keyrings/ros-archive-keyring.gpg

RUN echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/
    ros-archive-keyring.gpg] http://packages.ros.org/ros2/ubuntu $(. /etc/os-
    release && echo $UBUNTU_CODENAME) main" | tee /etc/apt/sources.list.d/ros2.
    list > /dev/null

RUN apt update
RUN apt upgrade -y
RUN apt install ros-foxy-desktop python3-argcomplete -y

RUN echo "source /opt/ros/foxy/setup.bash" >> ~/.bashrc
RUN source /opt/ros/foxy/setup.bash
RUN source ~/.bashrc
RUN apt-get install git wget -y

WORKDIR /home/Stage
RUN git clone https://github.com/gardegu/LAB42_RTX_control.git
WORKDIR /home/Stage/LAB42_RTX_control
RUN ./install_dependencies.sh
RUN mkdir logs

RUN apt install python3-colcon-common-extensions -y
RUN echo 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/ros/foxy/lib:/opt/ros/
    foxy/opt/rviz_ogre_vendor:/opt/ros/foxy/opt/aml_cpp_vendor' >> ~/.bashrc
RUN echo 'export PATH=$PATH:/opt/ros/foxy/bin' >> ~/.bashrc
RUN echo 'export PYTHONPATH=$PYTHONPATH:/opt/ros/foxy/lib/python3.8/site-
    packages' >> ~/.bashrc

WORKDIR /home/Stage/LAB42_RTX_control
RUN apt install nano -y
```

ENSTA Bretagne

RAPPORT D'EVALUATION
ASSESSMENT REPORT

Merci de retourner ce rapport par courrier ou par voie électronique en fin du stage à :
At the end of the internship, please return this report via mail or email to:

ENSTA Bretagne – Bureau des stages - 2 rue François Verny - 29806 BREST cedex 9 – FRANCE
☎ 00.33 (0) 2.98.34.87.70 / stages@ensta-bretagne.fr

I - ORGANISME / *HOST ORGANISATION*

NOM / *Name* Universiteit van Amsterdam

Adresse / *Address* Science Park 900, 1098 XH, Amsterdam, The Netherlands

Tél / *Phone (including country and area code)* +31653697548

Nom du superviseur / *Name of internship supervisor* Arnoud Visser

Fonction / *Function* Senior Lecturer

Adresse e-mail / *E-mail address* a.visser@uva.nl

Nom du stagiaire accueilli / *Name of intern* Théo Massa

II - EVALUATION / *ASSESSMENT*

Veuillez attribuer une note, en encerclant la lettre appropriée, pour chacune des caractéristiques suivantes. Cette note devra se situer entre **A (très bien)** et **F (très faible)**
*Please attribute a mark from **A (excellent)** to **F (very weak).***

MISSION / *TASK*

- La mission de départ a-t-elle été remplie ? A B C D E F
  Was the initial contract carried out to your satisfaction?

- Manquait-il au stagiaire des connaissances ? ☐ oui/*yes* ☑ non/*no*
  Was the intern lacking skills?

  Si oui, lesquelles ? / *If so, which skills?* _____

ESPRIT D'EQUIPE / *TEAM SPIRIT*

- Le stagiaire s'est-il bien intégré dans l'organisme d'accueil (disponible, sérieux, s'est adapté au travail en groupe) / *Did the intern easily integrate the host organisation? (flexible, conscientious, adapted to team work)*

  A B C D E F

Souhaitez-vous nous faire part d'observations ou suggestions ? / *If you wish to comment or make a suggestion, please do so here* _____

COMPORTEMENT AU TRAVAIL / *BEHAVIOUR TOWARDS WORK*

Le comportement du stagiaire était-il conforme à vos attentes (Ponctuel, ordonné, respectueux, soucieux de participer et d'acquérir de nouvelles connaissances) ?
Did the intern live up to expectations? (Punctual, methodical, responsive to management instructions, attentive to quality, concerned with acquiring new skills)?

A B C D E F

Souhaitez-vous nous faire part d'observations ou suggestions ? / *If you wish to comment or make a suggestion, please do so here* _____

_____

INITIATIVE – AUTONOMIE / *INITIATIVE – AUTONOMY*

Le stagiaire s'est –il rapidement adapté à de nouvelles situations ? A B C D E F
(Proposition de solutions aux problèmes rencontrés, autonomie dans le travail, etc.)

Did the intern adapt well to new situations? A B C D E F
(eg. suggested solutions to problems encountered, demonstrated autonomy in his/her job, etc.)

Souhaitez-vous nous faire part d'observations ou suggestions ? / *If you wish to comment or make a suggestion, please do so here* _____

_____

CULTUREL – COMMUNICATION / *CULTURAL – COMMUNICATION*

Le stagiaire était-il ouvert, d'une manière générale, à la communication ? A B C D E F
Was the intern open to listening and expressing himself /herself?

Souhaitez-vous nous faire part d'observations ou suggestions ? / *If you wish to comment or make a suggestion, please do so here* _____

_____

OPINION GLOBALE / *OVERALL ASSESSMENT*

❖ La valeur technique du stagiaire était : A B C D E F
Please evaluate the technical skills of the intern:

III - PARTENARIAT FUTUR / *FUTURE PARTNERSHIP*

❖ Etes-vous prêt à accueillir un autre stagiaire l'an prochain ?

Would you be willing to host another intern next year? ☑ oui/*yes* ☐ non/*no*

Fait à _____, le _____
In Amsterdam, on August 18, 2023

Signature Entreprise _____ Signature stagiaire
Company stamp _____ *Intern's signature*

Instituut voor Informatica
UNIVERSITEIT VAN AMSTERDAM
Science Park 904
1098 XH Amsterdam
tel. 020-525 7463
www.science.uva.nl

Merci pour votre coopération
We thank you very much for your cooperation