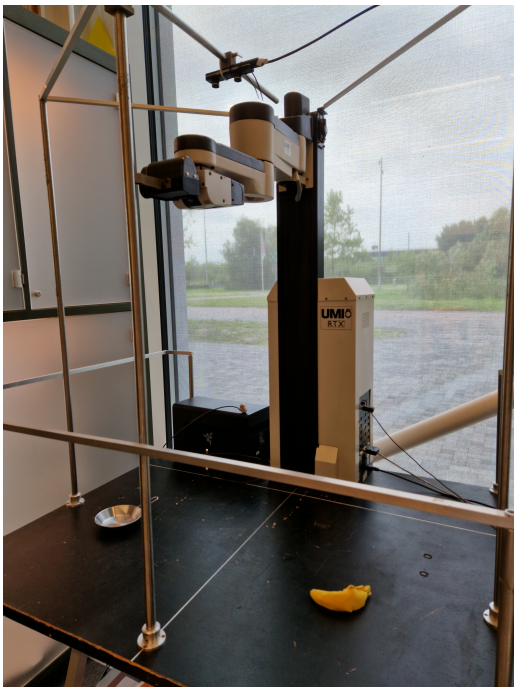


Internship report
Computer vision in a ROS 2 Interface for the UMI-RTX robotic arm

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

`https://github.com/gardegu/LAB42_RTX_control`

October 1, 2023

Abstract

This report presents the computer vision part of a project conducted in collaboration with my colleague Théo MASSA, *FISE 24, robotique autonome*, during our common intership at LAB42 in the University of Amsterdam, Amsterdam, Netherlands. The aim of this project was to work with an old robotic arm called the UMI-RTX (created in the 1980s) and make it grab objects on a plane with its gripper. Some work had already been done on this robot by students, but mainly with old tools. The idea of our project was to implement a new way of making it work and to use more recent tools. More specifically, our goal was to set up a ROS 2 environment and build an interface that would allow us to perform image analysis, trajectory planning, and target grabbing. This report focuses on the computer vision part. Another document, available at `https://www.intelligentroboticslab.nl/reports-and-theses/`, sums up all the work that has been done during this internship without separating Théo's work from mine.

We chose a plush banana as a target. With the computer vision library OpenCV and the software development kit of Stereolabs (the manufacturer of our camera), I managed, in a dedicated ROS 2 node, to detect the banana on the field and compute its 3D position. More precisely, I first wrote my own code to compute depth based on stereo vision and OpenCV built-in tools. Then, to get better results, I wrote a second code using Stereolabs' tools and got great, accurate results.

The rest of the project was done collaboratively with Théo and will not be detailed here. The information was sent to a node dedicated to inverse kinematics. In this node, the joints' states required to reach the aimed pose were processed and sent both to a simulation and the real arm. For this, we wrote two nodes, each dedicated to its own part, one for the simulation and the other for the real arm. Théo's custom Graphic User Interface (GUI) integrated the simulation, processed images, and depthmap. It also enabled us to choose between automatic control of the arm and a manual mode, where we could choose our own target pose.

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

To make our work accessible and portable, we used Docker tools to enable anyone to use our project.

Résumé

Ce rapport présente le travail réalisé dans le cadre d'un projet collaboratif sur lequel j'ai travaillé pendant mon stage au LAB42 de l'Université d'Amsterdam. J'y ai travaillé aux côtés de Théo MASSA, camarade de promotion, sur un sujet commun. Le but de ce projet était de faire saisir des objets à un vieux bras robotisé de type UMI-RTX en se basant sur ROS 2 et sur la vision par ordinateur. Il s'agissait pour nous d'utiliser des outils récents pour faire fonctionner ce robot des années 1980. Nous nous sommes répartis les tâches et ce rapport se propose de développer celles que j'ai réalisées. Il s'agira donc ici de vision par ordinateur, d'architecture ROS 2 et de conteneurisation.

Pour saisir les objets ciblés, j'ai travaillé sur des algorithmes de vision par ordinateur, en particulier sur la stéréo vision. Grâce à un premier code basé sur les outils intégrés de la librairie OpenCV, j'ai mis en place un algorithme de détection d'objet et de détermination de position 3D. Si la détection fonctionne à merveille, ce n'est pas le cas du calcul de la profondeur qui reste approximatif. Cependant, j'ai réussi à pallier ce problème en écrivant un second code de calcul de profondeur basé sur les outils du concepteur de la caméra stéréo utilisée dans ce projet. Les résultats sont alors extrêmement précis et suffisants pour notre projet.

Cette partie de vision par ordinateur a été intégrée dans le projet pour fonctionner avec les parties de Théo, notamment des calculs de cinématique inverse, le jumeau numérique du robot dans une simulation, et une interface graphique développée spécialement pour ce projet qui nous permet d'afficher les images utiles et de choisir entre un mode de pilotage manuel ou automatique du robot.

Grâce à notre travail, nous avons réussi à faire attraper un objet à notre robot. Et, pour rendre ce projet accessible et portable, nous avons mis en place avec Docker une conteneurisation de notre architecture ROS 2 utilisable par tous.

Key words

Computer vision, ROS 2, Docker, robotic arm.

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 me 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. My final thoughts of gratitude are for Théo, my friend and colleague, with whom I have shared this experience.

Disclaimer

Our supervisor required Théo and me to submit a joint project report in the middle and at the end of the internship. This report reflected our partnership on this work, which was the fruit of the pooling of the tasks we had carried out. As a result, we wrote this report together, which was published on the LAB42 website. The report that follows takes up many elements of this work very freely, since I am one of the authors. I also quoted Théo's work. Parts that have been written by him or by us will appear in blue in this report.

Contents

Chapter 1

Introduction

This project was led in an internship context, mainly for educational purposes, at Lab42 at the University of Amsterdam (UVA), the Netherlands [7]. The UVA is a top-100 university globally and was founded in 1632. Renowned for its rich history and academic excellence, UvA offers a diverse range of programs across various disciplines, including arts, sciences, social sciences, and humanities. With a commitment to fostering a vibrant learning environment, the university attracts students from all over the world. UvA's world-class faculty, cutting-edge research, and modern facilities make it a top choice for those seeking a globally recognized education in a dynamic and culturally diverse setting. Research is at the center of the UVA and around 500 doctoral degrees are conferred every year[1]. One of their top laboratories is LAB42. LAB42 is the UVA's new center for Digital Innovation and AI. Where students and researchers from UvA's Informatics Institute, the Institute for Logic, Language and Computation (ILLC) and companies all work together. The project presented here has been conducted in LAB42.

The aim of this project was to work on an old industrial arm, the UMI-RTX, which was created in the 1980s [10][11], and to make it autonomously grab an object that would have been previously detected. A lot of work had 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. Their work served as an introduction to this project and the supervisor of this project also gave hardware drivers as a mean of beginning the work.

This project was split into two parts. Théo MASSA worked on the first part[2]: ensuring the operation of the arm manipulation, making a simulation, and integrating it into the ROS 2 architecture. The other part, presented here, concerned computer vision, its integration into the ROS 2 interface, and the construction of the Docker image for this project.

The problem to be addressed through this work was how to perform target grabbing using an old robotic arm and state-of-the-art scientific tools. This project studied the possibility of mixing technologies of different ages to successfully run a complex task. More specifically in this report, the focus has been set on the possibility of using OpenCV and stereo vision to perform depth computation inside a ROS 2 architecture for the arm.

This report first presents an attempt to carry out this task using OpenCV, but also a

[1] `https://lab42.uva.nl/`

[2] His work is available in *Appendices G and H*.

new perspective on the issue in order to obtain results with a higher level of success and accuracy. Then it will show how the computer vision part of this project was integrated into a ROS 2 interface. To go further, this report also presents a way to make this work easily portable and accessible by using Docker and its tools.

Videos of the UMI-RTX in action are available here: https://www.youtube.com/playlist?list=PLr7kwtXen7-Se0UGnNa_Y2hR0W9sA3iEf

Figure 1.1: Views from LAB42 and the Intelligent Robotics Lab (Source: https://lab42.uva.nl/read).

Chapter 2

Computer Vision

To make a robotic arm grab a target, a strong starting move is to rely on one key element: computer vision [15]. 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 were used to detect a target and get its 3D position in the camera's frame. The language of programming that was used here is C++, and the OpenCV[1] methods that are cited here were written accordingly with the C++ syntax.

2.1 Detection of the target in a horizontal plane

The target, depicted by figure 2.1 was a yellow banana plush. It was put on a dark horizontal plane on which the UMI-RTX was fixed.

Figure 2.1: The banana plush on the dark horizontal plane that supports the arm

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

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

coherent with the fact that most objects are not rectangular but have curves; therefore, the approach was more general.

The image process was made with OpenCV, which includes built-in methods for computer vision. To extract the banana from the scene, a specific color space was chosen: the HSV color space (Hue, Saturation, Value) [16]. It is more common to hear about the RGB [16] color space (Red, Green, Blue), in which each color is represented by a set of three values between 0 and 255 corresponding to a proportion of the associated color. This is the color space usually used to associate a color to screen's pixels. However, the HSV space is an appropriate color space to perform color detection when there are changing brightness conditions because one color can still be detected in different light contexts. A representation of these spaces is given in figure 2.2. Each color is represented by a triplet of values between 0 and 255, corresponding to its values of hue, saturation, and value [16].

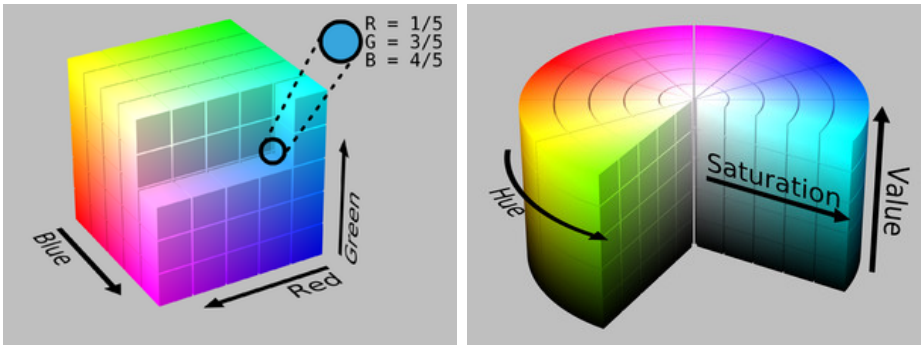


Figure 2.2: Representation of the RGB and HSV color spaces [16]

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 were selected, (20,100,100) and (60,255,255), to extract objects in between[2]. These values were chosen to binarize the image with *cv::inRange()* and extract yellow objects that have similar HSV values. The result was a binarized image with white objects on a black background (see figure 2.3).

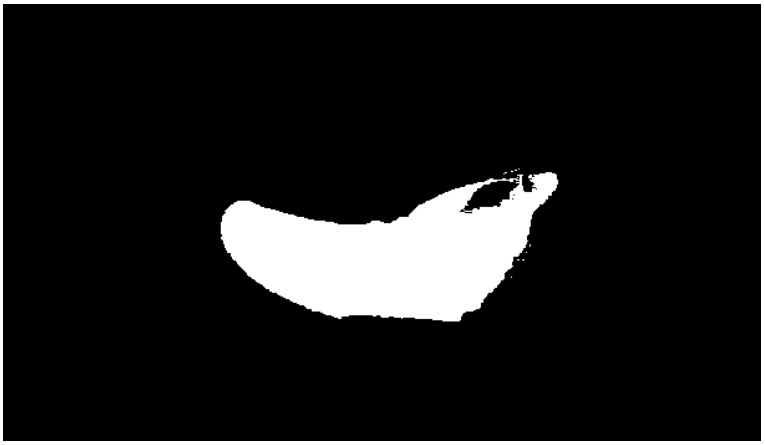


Figure 2.3: Image obtained after binarization to extract the banana

[2]The saturation value range has been chosen wide to be sure to detect the target due to highly variable light in Lab42 (natural and/or artificial light). These values can be adjusted.

Then the image was ready for contour detection. A hypothesis made for the project was that the only visible yellow object in the scene would be the banana target. This ensured that when performing contour detection, only the contour of the banana would be found. The method *cv::findContours()* gathers all the contours detected, in this case only the contour of the target (see figure 2.4). Then, *cv::moments()* gave access to the moments of the contour, and the coordinates of its centroid in the reference frame of the image became easily computable.

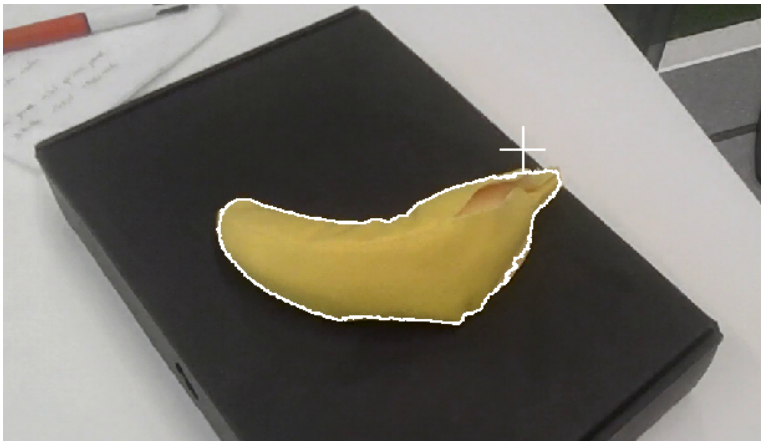


Figure 2.4: Example of detected banana and its contour with OpenCV

With the coordinates of the centroid, the banana could be located in the horizontal plane. The next step was to access its depth with respect to the camera and grab it with the UMI-RTX's gripper.

2.2 Generating depth with stereo vision using OpenCV

To allow the arm to grab the target, it needed to know where it was. 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 was the ZED Mini camera device from *StereoLabs*[3] depicted in figure 2.5.

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

Figure 2.5: The ZED Mini stereo device

On this type of device, both lenses are on the same support and have parallel optical axes and coplanar image planes (see figure 2.6). Some stereo installations use two distinct cameras that can be separated from each other according to need [8].

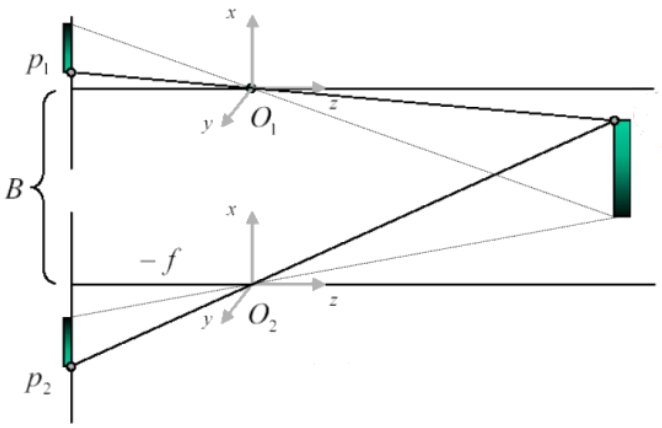


Figure 2.6: Vision of an object (on the right) with a stereo camera (on the left). O_1 and O_2 are the optical centers and B the distance between the optical axes [16]

The main idea behind stereo vision is to reproduce human vision [17]. From the difference in perception of the scene, depth information can be extracted. OpenCV provides methods and algorithms to get to that point step by step [9]. 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* [4] [8].

2.2.1 A bit of geometry

Projective space is an extension of Euclidean space where parallel lines meet at infinity [8]. To work in projective space, it is necessary to use *homogeneous coordinates* [17] [8].

These coordinates are used to characterize changes in space and allow to consider points at infinity and calculate 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 (see figure 2.7). It describes the relationship between two views of the same object [17].

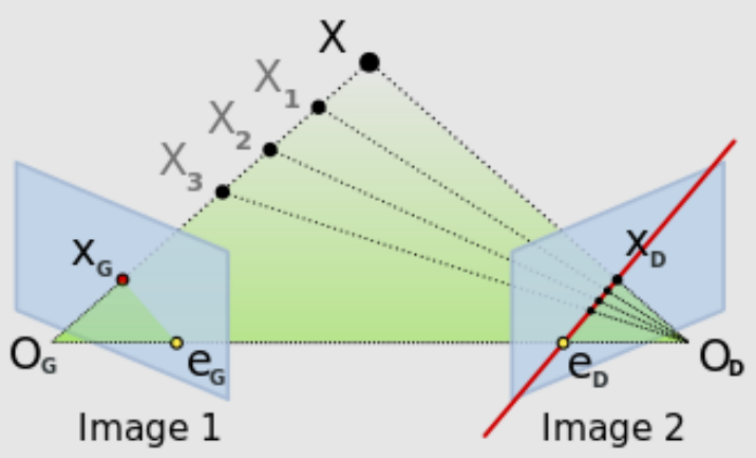


Figure 2.7: Representation of the epipolar plane [17]

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

2.2.2 Parameters

Knowing some parameters associated with the scene and the camera was necessary in order to access depth information. The first type of parameter is the *intrinsic* parameters [17], 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 [17], 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] [17] 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.

2.2.3 Calibrating the stereo camera

The first step of this process was to calibrate the ZED Mini device in order to compute the extrinsic parameters, the fundamental matrix, and the camera matrices [8][17]. To do so, stereo images with easy-to-detect points were used to apply correspondence algorithms to compute the results. In this project, a black and white chessboard (see figure 2.8) was photographed five times in different poses to guarantee robustness.

Figure 2.8: One of the views of the chessboard used to calibrate the ZED Mini device

It was necessary to declare the inner pattern that would be searched by the algorithm. In this case, it was the inner part of the chessboard that had 7 by 5 corners. It is very important to correctly count the inner corners for the algorithm to work. It was also necessary to provide the algorithm with the size of a square (3.1 cm here). Then, for each pair of images, *cv::findChessboardCorners()* was applied for the left and right views. This method would detect the declared pattern in the images and the associated corners (see figure 2.9). Then, it was possible to use *cv::cornerSubPix()* to refine the positions of the detected corners.

Once this had been done, the 2D positions of the corners in the left and right images were saved in associated vectors, and, for each pair of images, the 3D coordinates of the corners with respect to the top left corner, using real dimensions, with the third coordinate set to 0 for now, were saved in a dedicated vector. Which means that there was then a vector whose five components were the same and that there were two other vectors whose components were relative to the images. Finally, *cv::stereoCalibrate()* allowed to compute the following parameters: calibration Root-Mean-Square (RMS) error, left camera matrix, right camera matrix, left distortion coefficients, right distortion coefficients, rotation matrix, translation vector, essential matrix, and fundamental matrix. The algorithm that summarizes the process is in Appendix A.

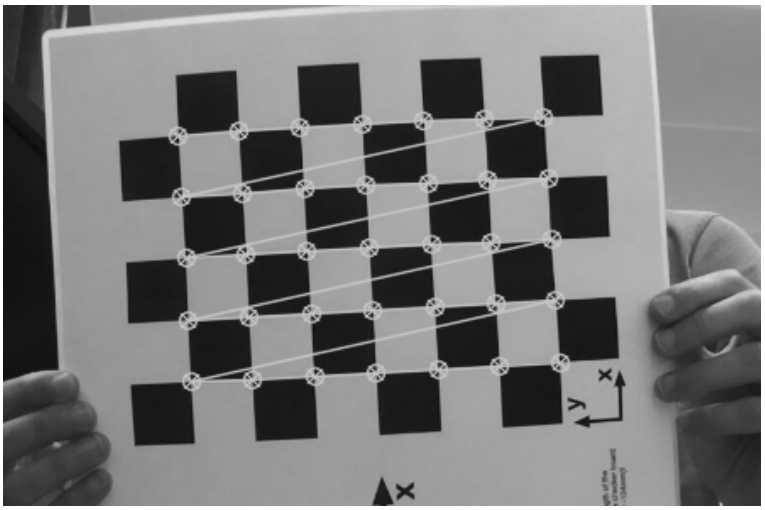


Figure 2.9: Detected corners on the precedent left view, associated to the declared pattern

2.2.4 Stereo rectification parameters computation

After calibrating, it was necessary to rectify certain aspects. The goal here was to make both camera image planes the same by finding the right rotation matrices. This makes all the epipolar lines parallel and simplifies the correspondence problem [8]. For this part, camera matrices and distortion coefficients that had been computed during the previous step were employed. *cv::stereoRectify()* was used to 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.

2.2.5 Stereo rectification

The next step involved the computation of the joint undistortion and rectification transformation and the representation of the result in the form of maps for remapping, achieved by utilizing *cv::initUndistortRectifyMap()*. To accomplish this, the parameters obtained during the previous step were used. Then, the images to be rectified (in this case, the work scene) were remapped using *cv::remap()*. This procedure had to be executed individually for each view. Figure 2.10 depicts the work scene captured by the left camera, along with its corresponding rectified image. As seen in the bottom image, the previously curved lines have been corrected into straight lines, thereby preparing the data for further disparity analysis.

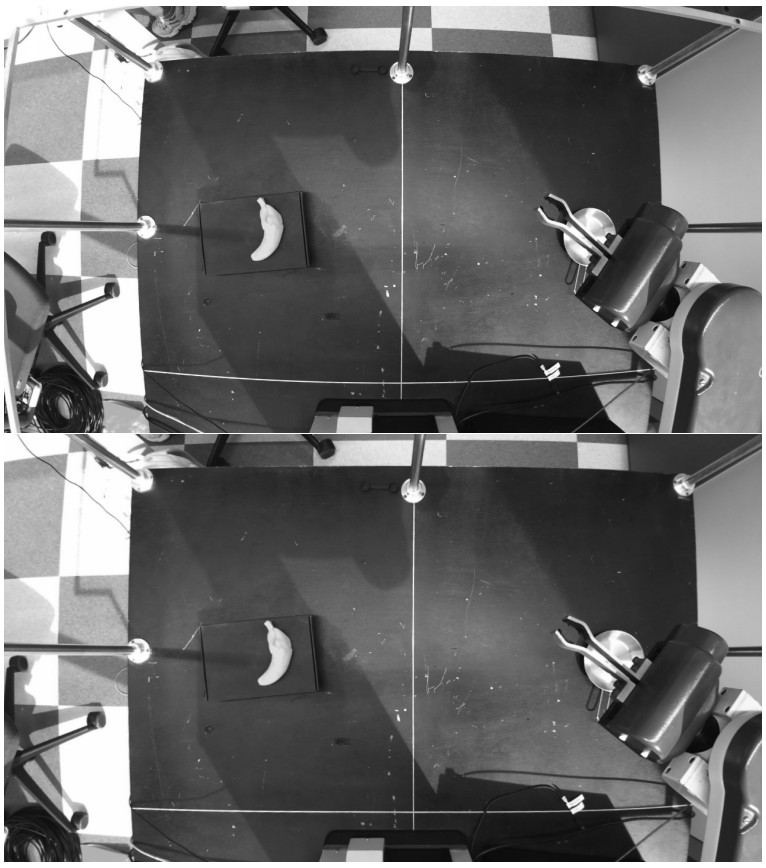


Figure 2.10: Work scene view and associated rectified image

2.2.6 Disparity map computation

Stereo vision reproduces human vision [17][8]. Each camera within the stereo device perceives the scene similarly to the human eye. Consequently, depending on the particular view under consideration, the scene appears to shift either to the right or to the left. This shift provides valuable depth information. There exists a horizontal disparity in the positioning of objects between the views, which can be quantified in terms of pixels and is referred to as "disparity" [17][8]. The underlying principle is straightforward: when examining an object within a stereo image of a scene, a greater disparity indicates the object's proximity. Calculating disparity for each object in the scene facilitates the generation of a disparity map. Furthermore, given that depth (Z) and disparity (d) are proportionally related through the equation $Z = \frac{fB}{d}$, it is possible to derive a depth map from a disparity map.

OpenCV provides two algorithms for computing disparity between two stereo-associated views: *cv::stereoBM()* and *cv::stereoSGBM()*, which is a modified version of the former. These algorithms perform horizontal block matching between the views. Certain parameters of the constructor needed to be configured before computing the disparity map, and each parameter exerts a precise influence on the results. Setting them correctly was challenging. A concise description of these parameters can be found in Appendix B. It's crucial to remember that these algorithms are sensitive to texture. The absence of texture in the images can lead to poor results. Therefore, it was necessary to optimize them for robustness before testing them on the planar support of the UMI-RTX. Throughout this project, numerous configurations involving different images and parameter values were experimented with, yielding diverse results. Figures 2.11 and 2.12 illustrate the process. The first image shows an excerpt from the rectified left view, highlighting objects visible to both cameras.

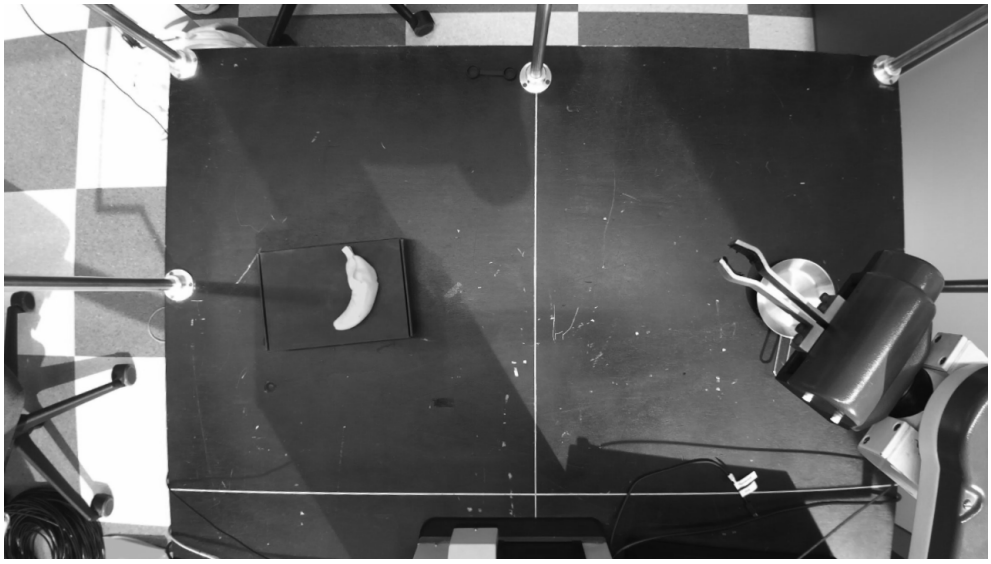


Figure 2.11: Left view of the scene

For this scene, here is the disparity map obtained:

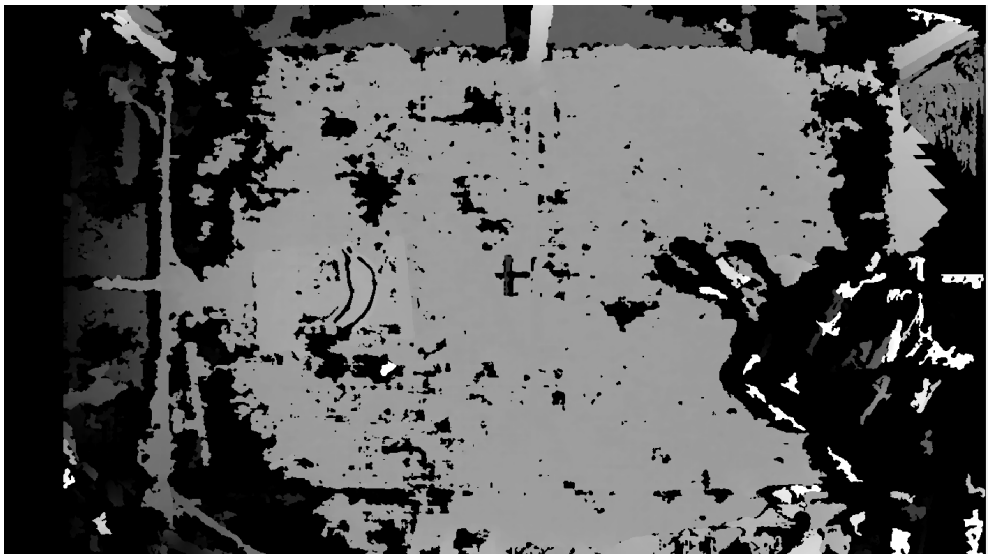


Figure 2.12: The associated disparity map

On the disparity map, various levels of gray were observed. The brighter, the closer. This map was still noisy, but it was one of the cleanest disparity maps made during the project. It is important to consider that finding convenient parameter values was difficult and that the option of scaling down the disparity or normalizing remained available. To ease parameter optimization, an adjustment interface in the form of a Graphic User Interface (GUI), with trackbars associated with individual parameters, was used[4].

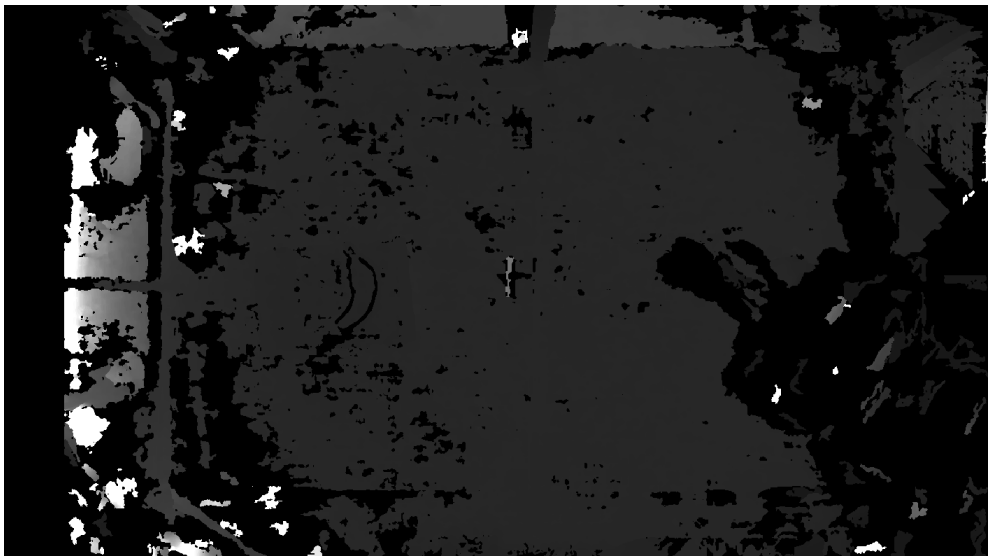


Figure 2.13: The associated depth map

However, changing one parameter often strongly disturbs the harmony of the map. The depth map associated with this image was not very good yet (see figure 2.13).

[4]See Appendix C.

2.3 Generating depth with Stereolabs' SDK

The depth map obtained with OpenCV alone was too imprecise. The chosen process was coherent, but adjusting parameters to get valuable and useful 3D information was too complex. There was an opportunity to enhance this system by adopting a more effective solution. As demonstrated in Section 3.2, depth computation can be accomplished solely using OpenCV, thereby bypassing Stereolabs' drivers. Nonetheless, given the utilization of a Stereolabs device in this project, the option of exploring and working with their software development kit (SDK) remained available. It has built-in tools that enable the user to access trustworthy and quality information.

2.3.1 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 used one of their cameras, the ZED Mini, initially designed for mixed reality, and their software development kit.

2.3.2 Using Stereolabs' software development kit

The SDK[5] 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 (see figures 2.14 and 2.15) and their associated graphic user interfaces, such as a simple stereo viewer or a depth viewer with 3D scene reconstruction.

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

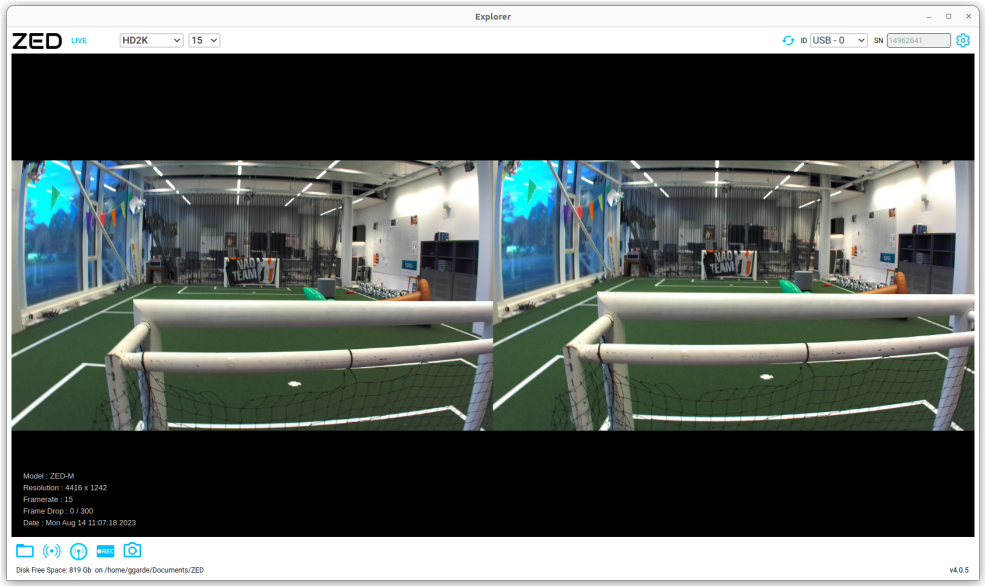


Figure 2.14: ZED_Explorer a simple tool to access the stereo vision of the ZED Mini - view of LAB42

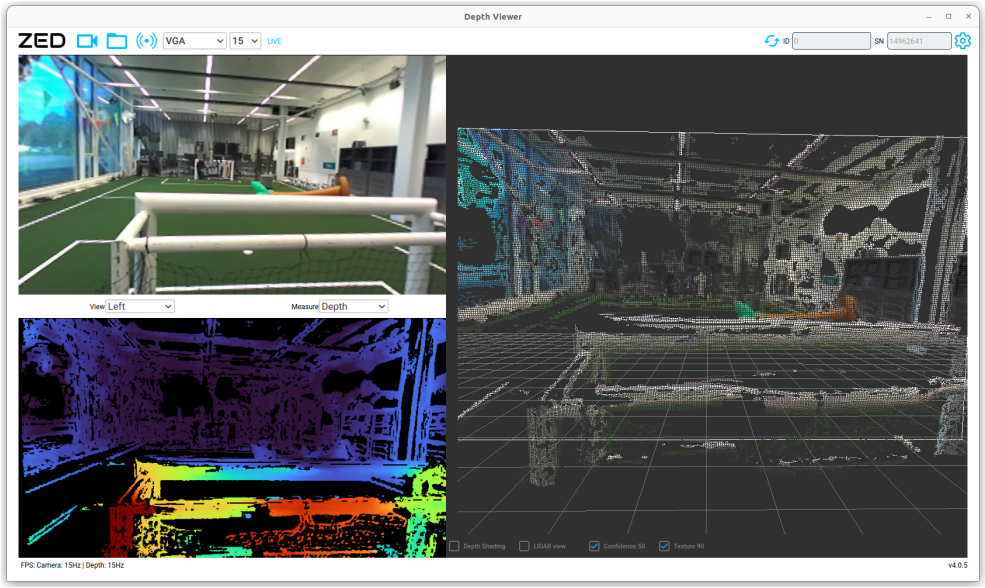


Figure 2.15: ZED_Depth_Viewer a simple tool to access the depth map of the scene and a 3D reconstruction - view of LAB42

This project used the latest release of the SDK, the ZED SDK 4.0. A choice of an SDK was necessary considering that there was no general package provided for stereo cameras. Stereolabs also provides a diagnosis service that analyzes the installation of the SDK, the status of the ZED device, and the availability of the graphic drivers. Besides, it allows the user to access valuable information with great precision, which could not be accessed before by simply using the stereo device and OpenCV. Among the provided data are the precise focal lengths (see Appendix C), which are needed to compute the (X, Y) positions, horizontal and vertical, in metric units from Z, the computed depth. In addition, these tools allow the user to set the desired frame rate per second (FPS), video quality, and even the depth mode. It means that the working mode can be chosen coherently with the

performances needed. For instance, in this project, a highly performing mode was used. In the context of this work, it suited perfectly. However, it was essential to consider that higher performance often translates to heavier computational demands. There is even a final mode that pushes performances further. It consists of adding artificial intelligence, which improves the depth map by pertinently correcting the values. But it was not used in this project. The last release of the SDK (version 4.0) uses an NVIDIA library called CUDA to run AI and computer vision tasks. Hence the necessity of having a computer with great graphic processing capabilities. Even though it was easy to download[6] the SDK and to access its documentation[7], having PCs that were not powerful enough to use its tools and do not have graphic processing units (GPUs) was a problem. This is why using the ZED SDK was really a new step in this project compared to simple OpenCV-based stereo vision. From here, it was necessary to start working with a very powerful computer that had everything needed to continue the work.

2.3.3 Integration into the project

It was a good choice to use the SDK when needing something precise and efficient to compute depth. Using OpenCV alone had slowed this project down. The idea behind this choice was to keep the structure of Section 3.2 and adapt it to the SDK's tools. The advantage of doing so was that the code was much lighter and clearer now. There was, indeed, no use for a calibration or rectification part. Thanks to the SDK, the stereo camera can nearly directly work at full capacity. There were only a few adjustments to make. The initial process for the vision code was:

(i) setting the stereo SGBM algorithm and its parameters

(ii) opening the camera and checking its availability

(iii) stereo calibrating

(iv) stereo rectifying

(v) entering the processing loop

- (a) reading the stereo image of the scene
- (b) splitting the image into left and right views
- (c) detecting the banana's position and orientation
- (d) computing disparity
- (e) computing depth
- (f) publishing the resulting images and data

The advantage of this code was that the evolution of the images was easily visible. It got through the whole depth computation process. Whereas with the SDK, the process was much lighter since there was no use for calibrating, rectifying the views, or computing disparity. To use the SDK in this code, it was necessary to specify in the *CMakeLists.txt* the appropriate dependencies:

[6] https://www.stereolabs.com/docs/installation/linux/

[7] https://www.stereolabs.com/docs/

```
find_package(ZED 3 REQUIRED)
find_package(CUDA ${ZED_CUDA_VERSION} REQUIRED)
```

In the header of the computer vision node, the inclusion of Stereolabs' tools was accomplished in the following manner:

```
#include <sl/Camera.hpp>
```

The steps then were:

(i) setting camera parameters (resolution, fps, depth mode, units, distance range)

(ii) opening the camera and checking its availability

(iii) entering the processing loop

- (a) getting the left view, the depth map and the 3D points cloud
- (b) detecting the banana's position and orientation
- (c) getting the associated 3D coordinates
- (d) publishing the resulting images and data

This work was conducted with specific parameters: a **HD720** resolution, **15 fps**, the **ultra-depth** mode, in **millimeters**, and a minimum computable distance of **100 mm**. Then, the methods *zed.retrieveImage()*, granted access to the left and right views and to the depth map[8]. It was as simple as that. These images were then published on dedicated topics. Subscribing to them or using RQt[9] was enough to display them (see figures 2.16 and 2.17).

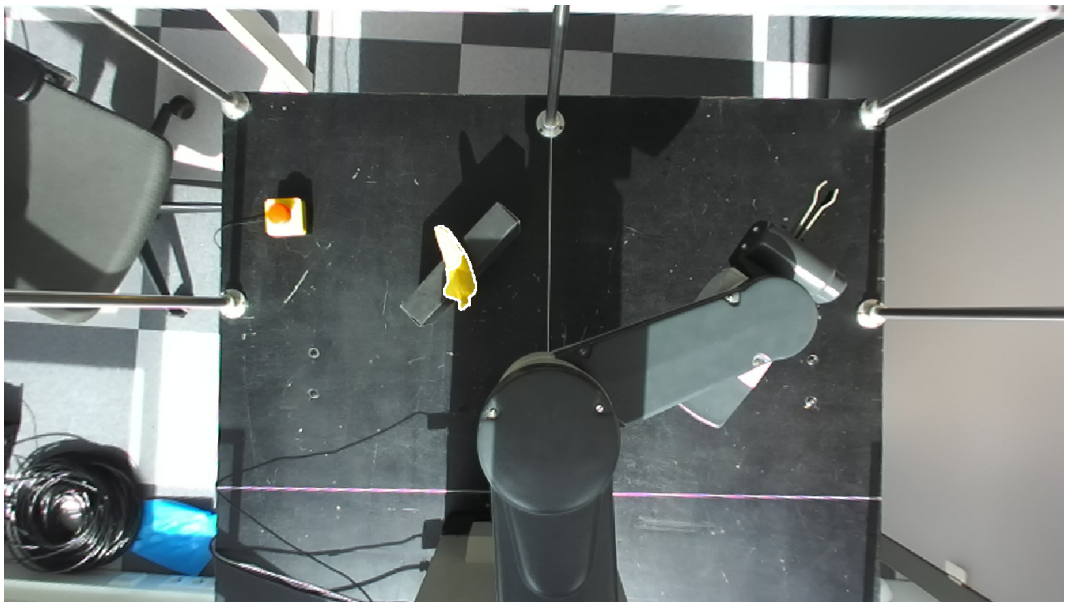


Figure 2.16: Left view with target detection on

[8] With *zed.retrieveMeasure()*, access is given, among others, to the 3D points cloud

[9] http://wiki.ros.org/rqt

Figure 2.17: Depth map of the scene

Figure 2.16 and *figure 2.17* are useful to make a fair comparison with the images from figures 2.11 and 2.12. On *Figure 2.16* and *figure 2.17*, the project's work space with the robotic arm and the banana is depicted. On the first one, it is clear that the banana has been detected and its contour drawn. On the second one, the UMI-RTX is brighter than the rest of the image, above its planar support (much darker). There lies the banana target on a box. This helped gain insight into how the situation was evolving, and it was then easy to figure out if something went wrong.

However, the images obtained could not yet be correctly seen or published on ROS 2 topics. They needed to be converted from the format designed by Stereolabs to an OpenCV format so that they could be published with ROS 2 (note that ROS 2 nodes were provided by Stereolabs. However, in that case, these functions could not have been integrated into a custom node of their project; one could only use these unmodifiable nodes). To do so, two handmade methods provided by Stereolabs were used: *slMat2cvMat()* and *getOCVtype()*. These are not part of the SDK but use items from it[10]. *slMat2cvMat()* creates a *cv::Mat* matrix from a *sl::Mat* by accessing its dimensions, the OpenCV type given by *getOCVtype()* from the Stereolabs type, and a pointer to its values in the memory. Once this is done, it is required to convert one last time from the BGRA format to BGR. It is used in computer vision for alpha compositing. The idea is to combine one image with a background to create a transparency effect [12]. Publishing OpenCV images with ROS 2 was easily feasible by using:

```
sensor_msgs::msg::Image::SharedPtr X_msg = cv_bridge::CvImage(std_msgs::msg
    ::Header(),"X_format",X_source_image).toImageMsg()
```

Afterwards, the process was similar. The target detection was identical to Section 3.1, but the acquisition of its orientation had been added to it (yaw, pitch, and roll). This was done through the *get_ angles()* method. The idea was to find a line that fitted the contour best by minimizing the distance to the edges. It could be done with *cv::fitLine()* [9]. Then, once the contour of the banana and the fittest line had been found, the last

[10] See Appendix E

step was to find an orientation for the target. At this point, the coordinates of its centroid were also available. This is where the 3D point cloud came in (see figure 2.18).

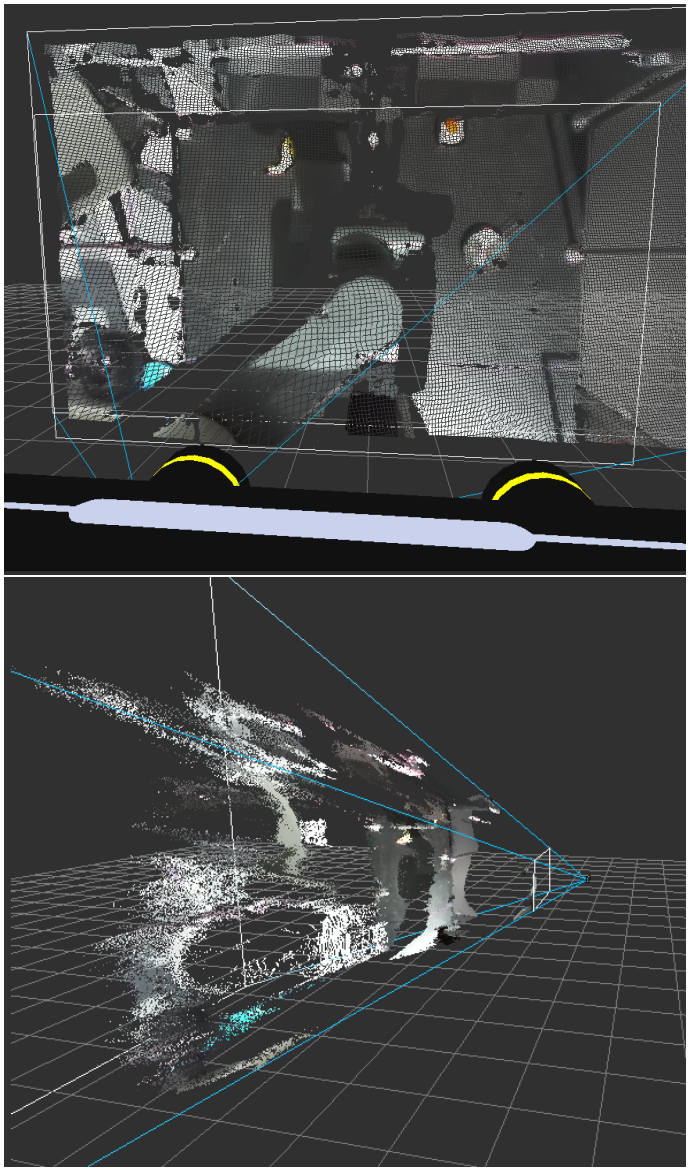


Figure 2.18: 3D point cloud of the scene. Front and side view. On the bottom of the front view, one can see the 3D model of the stereo camera

Given the pixel coordinates of the target's centroid, the corresponding metric (X, Y, Z) coordinates in the point cloud can be determined using the *getValue()* method. This is how the 3D coordinates in millimeters of the banana were obtained. But they were in the reference frame of the left camera. Hence, it was necessary to add offsets with respect to the origin of the world frame of the UMI-RTX. The origin was marked by the intersection of the lines (see *Figure 3.13*). At this point, it was just needed to publish the processed and depth images and the coordinates of the target. Nonetheless, this code had to be run with a specific version of ROS 2 (Foxy). Therefore, it was possible to use a Docker image that provided everything required.

Chapter 3

ROS 2 Interface

3.1 Presentation of ROS 2

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

ROS 2 (Robot Operating System 2) [1] is an open-source platform created for the development and administration of robotic systems [13]. It builds upon the legacy of ROS 1 while introducing numerous improvements and additional features aimed at simplifying the development and deployment of robotic applications. Thanks to its modular and decentralized design, ROS 2 offers a flexible and expandable framework for building complex robotic systems.

By employing a publisher-subscriber messaging system, ROS 2 facilitates efficient communication between different elements within a robotic system, streamlining the flow of data and commands. It boasts compatibility with various programming languages and offers a wide range of tools and libraries that streamline the development process. Additionally, ROS 2 places a strong emphasis on real-time and embedded systems, making it a highly suitable choice for a broad spectrum of robotic applications, ranging from compact embedded devices to extensive distributed systems [13].

ROS 2 offers several significant benefits to developers and roboticists. It enhances performance and reliability through its optimized middleware, facilitating communication between nodes. This improved performance is especially beneficial for tasks that

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

demand real-time or low-latency operations. Additionally, ROS 2 places a strong emphasis on security and safety, integrating features such as authentication and granular access control, making it a more suitable option for applications and environments characterized by their sensitivity [13].

The core of ROS 2's communication framework revolves around two fundamental concepts: nodes and topics. Nodes represent discrete software components responsible for carrying out specific tasks within a robotic system. They serve as the foundational building blocks of a ROS application and interact with each other through message passing. On the other hand, topics act as the communication channels through which nodes exchange messages within the ROS ecosystem. A topic functions as a dedicated pathway where nodes can either transmit messages or subscribe to receive them. This adheres to the publish-subscribe communication pattern, where nodes that produce data publish messages to a topic, and nodes interested in that data subscribe to the relevant topic to receive these messages. [13].

3.2 Architecture of the project

Here are the interactions between every ROS node when only the simulation was running:

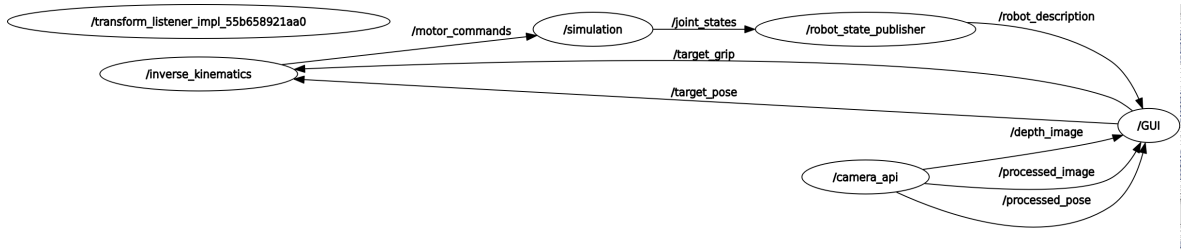


Figure 3.1: Node graph when running only the simulation

Here, the central node, denoted as */GUI*, serves as the execution environment for the graphical user interface (GUI) and functions as the central hub for data consolidation. It transmits the designated pose to the */inverse_kinematics* node, which is responsible for processing inverse kinematics computations to obtain and subsequently dispatch the necessary joint states essential for achieving the desired arm positioning. This transmission is facilitated through the utilization of the */motor_commands* topic.

The */simulation* node is configured to subscribe to the aforementioned topic and employs the */robot_state_publisher* mechanism to convey the robot's descriptive information to the integrated simulation panel within our GUI. A comprehensive description of this panel can be found in Section 5.4 of this thesis. This established closed-loop configuration facilitates precise control of the robotic arm in accordance with predetermined specifications.

In this system, meticulous attention must be paid to the handling of yaw angles. This precaution arises from the fact that, in the case of the physical arm, the yaw angle is ostensibly zero, whereas in reality, it corresponds to the value $arctan2(y, x)$, where x and y denote the coordinates of the end-effector. To clarify further, within the simulation environment, the yaw reference point aligns with the y-axis, whereas, for the encoders,

the yaw reference and neutral point align with the axis formed between the z-axis and the wrist. Consequently, it is imperative to exercise caution to prevent any potential confusion regarding the interpretation of yaw values.

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.

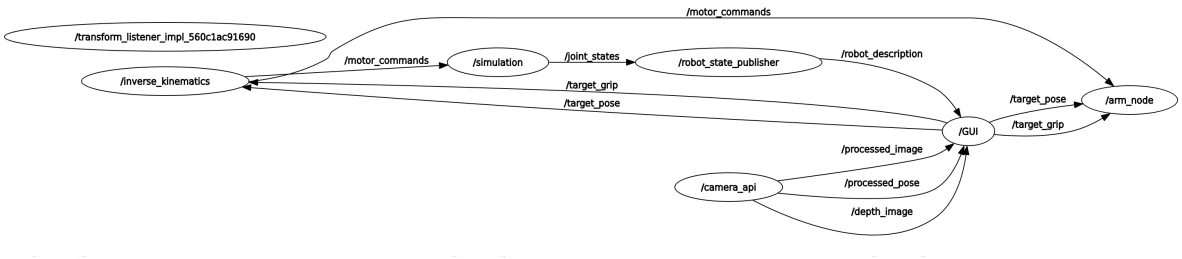


Figure 3.2: Node graph when running real arm

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.

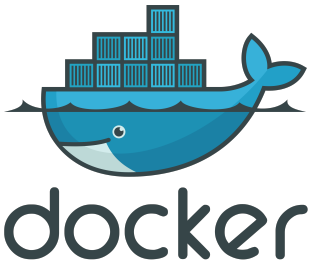
Table 3.1: Messages description

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

Chapter 4

Docker Image

4.1 Docker



Docker[1] is a platform for containerization that enables developers to package applications and their dependencies into lightweight, portable containers. These containers can run consistently across different environments, from development laptops to production servers, ensuring software consistency and ease of deployment. Docker simplifies application deployment, scaling, and management by isolating applications from the underlying infrastructure and providing a standardized way to package, distribute, and execute software, making it a popular choice for modern software development and DevOps practices [14].

4.2 Necessity of Docker

It was mentioned in Section 3.3 that a powerful computer with a Nvidia GPU was needed to run the application. The computer at disposal for this project came with a Linux installation. However, it had the Ubuntu 22.04 distribution, while the ROS 2 interface was written for Ubuntu 20.04. This slight difference was not to be neglected because ROS 2 ran under distinct version according to the Ubuntu distribution, and the code would not be identical. This means that the first version of the code, using OpenCV, was 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 could allow anyone to use this work project with Docker. No ROS installation was needed

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

any more. The only requirement was having Docker. As a result, this project became really portable. However, it required a NVIDIA graphic card as mentioned in Section 3.3.

4.3 Building the bespoke project's image

When it came to creating a new Docker image, one important thing had to be taken into account. Docker hosts a set of simple distributions that can be downloaded and used as a starting point for any new image. For instance, any distribution of Ubuntu or Windows can be used as base, and any additional layers can be installed with a script and commands.

Two options were coherent with this project. Building an image from a fresh Ubuntu 20.04 distribution and installing every driver, package, and other stuff manually. Or, using a predefined distribution furnished by Stereolabs and building around it. The first try was to do it fully manually with the first option.

However, 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 had been done, all that was left to do was installing ROS2, downloading the project's Git repository, and following the procedure in order to install it correctly. The *Dockerfile* can be found in Appendix F.

The particularity of this image was that the command to launch it was special. It needed several privileged accesses to the computer hardware: access to the GPU for image processing and the screen to launch the GUI. Privileged access was given at the launch with the docker tags: *–gpus all -it –privileged -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix*. Besides setting access to the screen for usage was done with **xhost +** in the terminal.

Chapter 5

Conclusion

The purpose of this work was to enable the UMI-RTX to accurately obtain its target's coordinates so that it could then proceed with the grabbing protocol. This computer vision process had to be integrated into a ROS 2 interface piloting the arm. After 4 months, the results were quite satisfying.

Of course, the process could be improved. But it became reliable and robust. It was able to use stereo vision to compute a depth map of the working scene, access the target's 3D position with respect to the UMI-RTX's world frame, and communicate this information to the other components of the ROS 2 architecture so that they could be translated into instructions.

Still, one of the major issues was the first version of the computer vision part. This code only used OpenCV, not the SDK. The problem was that the depth map was not reliable. It is good that the code was entirely built for this work and that it did not use the drivers. But the drawback was that setting the parameters correctly made it difficult to understand their influence. A list of improvement points can be made for further research: better understanding the format and type of disparity data; correcting the normalization if need be; pre- and/or post-filtering the images; computing depth using the disparity-to-depth matrix. It would have been smarter to write the SDK-based node first to have some strong results to rely on and compare to. A lot of time was lost trying to figure out what was missing. With more general improvements, the threshold settings that enabled the code to detect the targeted banana could be improved. Depending on the light in the laboratory, there could be some noise that made the banana go undetected.

It is also important to mention the work done with Docker. Creating a custom Docker image was quite challenging, but doing it enabled the project to be more accessible. This custom image had several advantages. It allowed everyone having a Nvidia GPU and Docker installed to use the arm; Ubuntu and ROS were no longer mandatory to do so. The only counterpart was that the image was not fully self-built and customized. It used an image furnished by Stereolabs, where the SDK had already been installed. Nonetheless, it worked well and gave satisfying results.

In a more personal perspective, this internship has been a great opportunity to consolidate my ROS 2 skills, improve my Docker knowledge, and learn a lot about computer vision. I am very satisfied with this experience.

List of Figures

List of Tables

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

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

[7] Guillaume GARDE and Theo MASSA. A ros 2 interface for the umi-rtx robotic arm. Master's thesis, ENSTA Bretagne and the University of Amsterdam, 2023.

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

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

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

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

[12] R. Lukac and K.N. Plataniotis. *Color Image Processing: Methods and Applications.* Image Processing Series. CRC Press, 2018.

[13] 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.

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

[15] 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.

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

[17] 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{Garde2023,
    title={Computer vision in a ROS 2 Interface
    for the UMI-RTX robotic arm},
    author={GARDE, Guillaume},
    school={ENSTA Bretagne and the University of Amsterdam},
    year={2023}
}
```

Otherwise, you can use this APA format:

```
Garde, G. (A. 2023). Computer vision in a ROS 2 Interface
for the UMI-RTX robotic arm. The University of Amsterdam, Amsterdam, the
    Netherlands.
```

Appendices

A. 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.

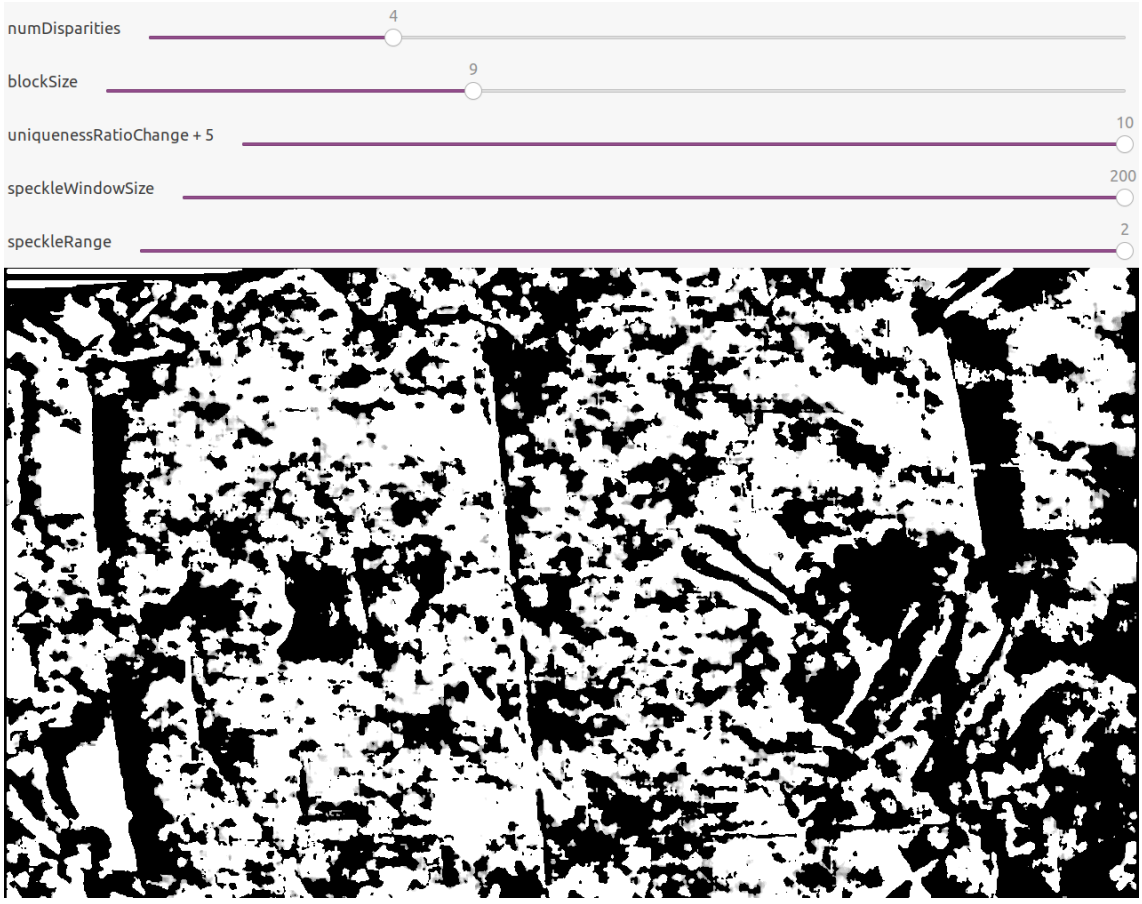
B. A quick description of the stereo block matching algorithms' parameters

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*.

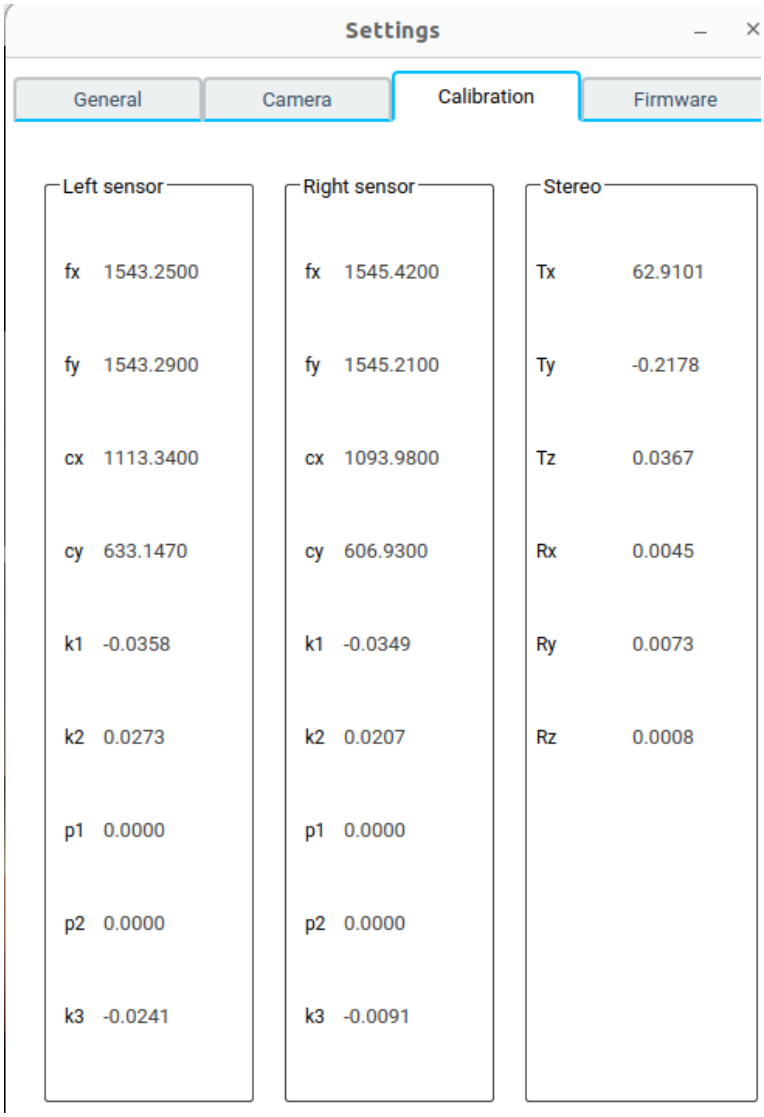
[1] Automatically computed with 0 as minimum value.

C. Customized simple GUI to set disparity parameters with trackbars



This disparity map is bad. This illustrates the fact that even with trackbars to help set the right values, it remains delicate to get reliable data this way.

D. Some camera parameters given by the SDK



E. Converting images from Stereolabs format to OpenCV format

Listing 5.1: 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;
}
```

F. Dockerfile

Listing 5.2: 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
```

G. Arm manipulation (by Théo Massa) [7]

Description

This project uses the UMI-RTX arm, which is quite basic in its composition [10]. 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 [10] that allow it to be controlled and know its state.

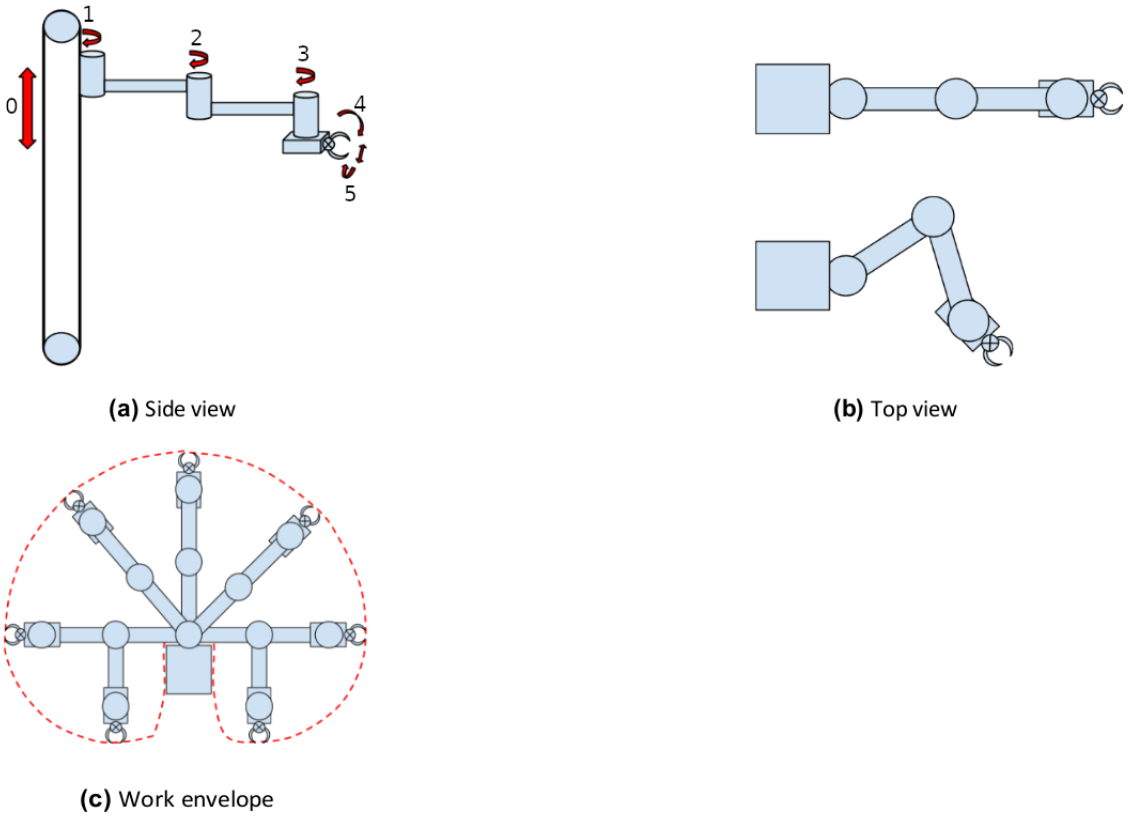


Figure 5.1: Model of the arm [3]

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:

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:

Table 5.1: Description of the joints ID

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

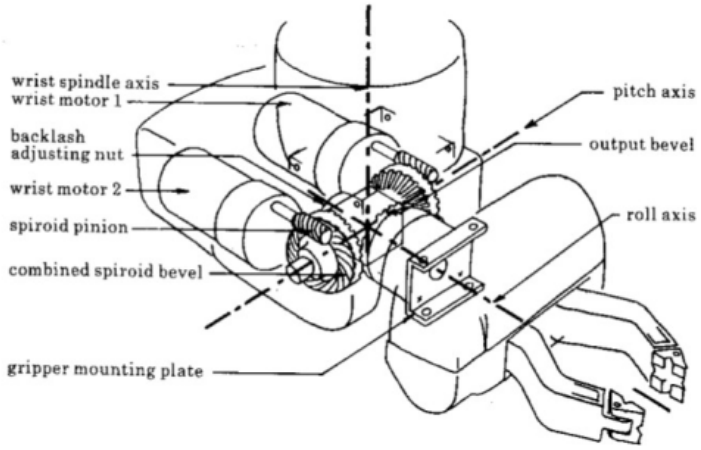


Figure 5.2: Wrist system [10]

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**.

URDF description

For practical purposes, it is really useful, even mandatory, to have a digital twin of the arm [6]. 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 [6].

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 D [2].

Listing 5.3: URDF Description of the arm

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

  <link name="base_link">
    <visual>
      <geometry>
        <box size="1.252 0.132 0.091"/>
```

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

```
    </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.

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 description into reality. The conversion formulas are:

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

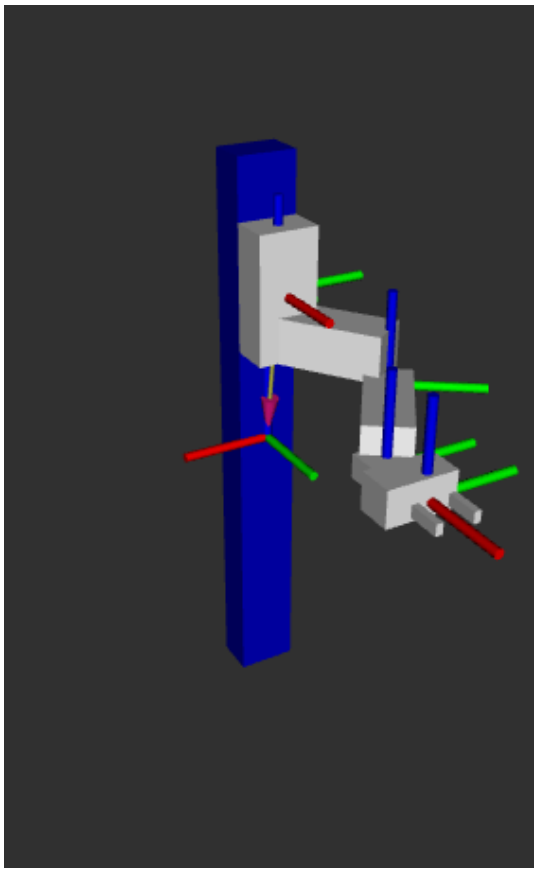


Figure 5.3: Virtual model of the arm

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

Communication with the arm

As the arm is old, the communication is not direct. The documentation is limited [11][10], 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. 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.

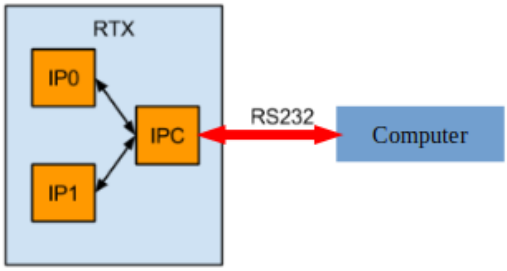


Figure 5.4: Communication between the arm and the computer

One particularity of the arm, is that the motors are controlled by two 8031 chips, IPs[3] 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 5.2: Overview of motors with corresponding IP

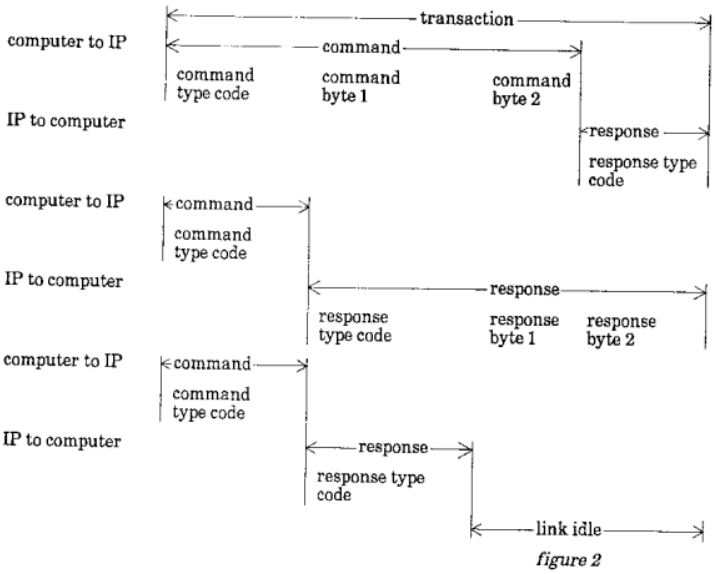


Figure 5.5: 3 ways of communication [3]

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

[3]Intelligent Peripheral

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.

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.

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.

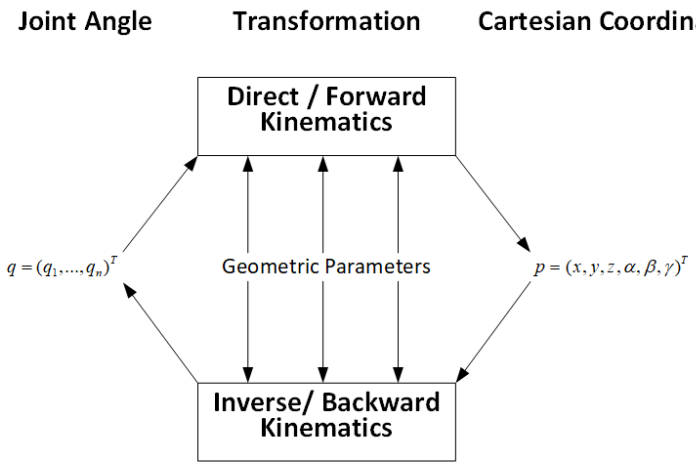


Figure 5.6: Inverse and forward kinematics

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 [5], 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*.

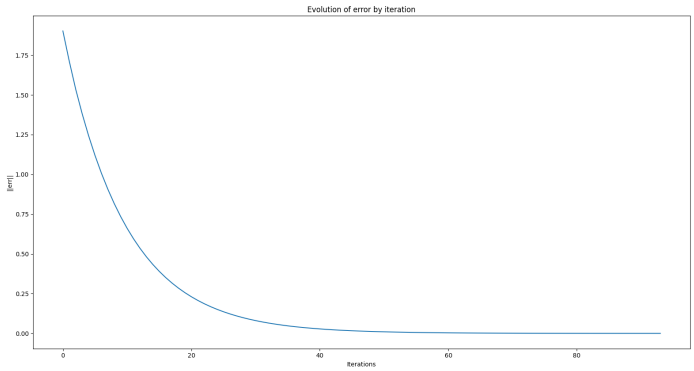


Figure 5.7: Evolution of the error according to iterations

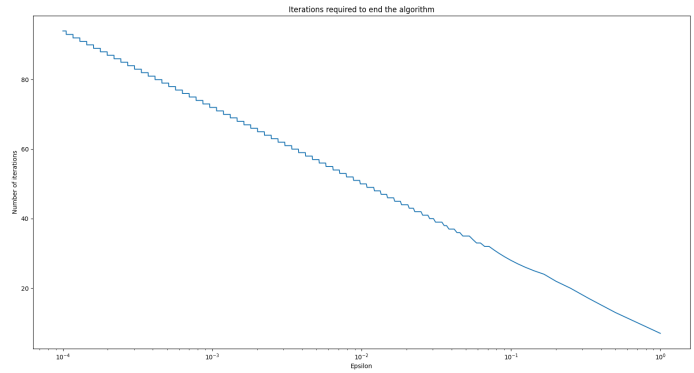


Figure 5.8: Evolution of the number of iterations according to ϵ in logarithmic scale

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.

H. ROS 2 Interface (by Théo Massa) [7]

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.

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 [4] 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 [5], 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.

[4] https://foxglove.dev/
[5] https://doc.qt.io/qt-5.15/

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 5.9: 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 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. 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.

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* Guillaume Garde

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