École Nationale Supérieure des Techniques Avancées de Bretagne

ENGINEER-ASSISTANT INTERNSHIP REPORT

done at THALES ALENIA SPACE

From May to August 2023

Computer vision application and Robotic Arm Visual Pose Estimation

---

Clara Gondot

2nd year in Mobile Robotics, ENSTA Bretagne, Brest

Supervisors in Thales Alenia Space

Valter Basso

Intellectual Property Manager

Andrea Merlo

Head of Robotics and Mechatronics Group

Academic tutor

Luc Jaulin

Thales Alenia Space

Strada Antica di Collegno, 253

10146 Torino (TO)

October 18, 2023

Abstract

This report summarizes the experiences and skills I have learnt during my Engineer-Assistant internship of 2023. The main themes are computer vision, the control of a robotic arm and the planning of a high-level communication between the different parts, in the frame of On Orbit Servicing.

Résumé

Ce rapport synthétise l'ensemble des expériences et compétences apprises pendant mon stage dit Assisstant-Ingénieur de 2023. Les thèmes abordés sont la vision par ordinateur, le contrôle de bras robotique et la mise en place d'une communication haut-niveau entre les différents composants, dans le contexte du Service En Orbite.

Acknowledgment

I want to thank all my co-workers who all welcomed me warmly and made possible this cheerful and fulfilling experience. I especially adress my kind regards to A. Bongiovanni and M. Lapolla for helping me in my work and my tutors V. Basso and A. Merlo for offering me a glimpse of the company world.

Contents

Introduction

From May to August 2023, I worked as an intern in Thales Alenia Space, in their site in Torino. Thales Alenia Space places among the leading european companies in the aerospace industry and is the result of a joint venture between the French group Thales and the Italian group Leonardo. I was welcomed in one of the two Robotics Laboratory of the Torino site. Their projects assignments revolved around *On Orbit Servicing* missions. On Orbit Servicing (OOS) encompasses all sort of interactions between a client satellite and a servicing satellite, like repairs, recovering waste or refueling. This internship as assistant-engineer was an opportunity to learn about what my own future work could be like, by observing my colleagues and their work.

The team of 6 engineers that I joined participated in the conception of a Robotic Platform hosted on the servicing satellite. Among the different missions and sub-projects onboard of this platform, I familiarized myself with the *rendez-vous* and *docking* processes of the OOS context. For the rendez-vous step, I studied the detection and the pose computation of the client satellite, more specifically the computer vision tasks that can be used in the OOS context; and for the docking between the two satellites, I focused on the use of a robotic arm to attain the client satellite.

This report will begin with the details of the context and organization of the internship. There are then two main parts, the first one focusing on the theory and application of Computer Vision; and the second one about the control of the laboratory robotic arm.

The report will be divised in two half, the first part reporting on the activities related to computer vision, a project devised to apply the knowledge I had gotten and explore different options. The second part is on the use of a robotic arm present in the laboratory.

1 Context and Planning

1.1 Workplace environment

The assistant engineer internship marked both my first experience within a large company and my initial foray into an international enterprise site located in a foreign country, representing a significant milestone in my professional journey. This opportunity proved to be invaluable and exceptionally enriching. While the rest of this report will delve into the technical and technological aspects of what I learned, it was also a chance to observe the functioning of a company operating on a site housing several hundred employees. (In the subsequent sections of this report, I will delve into the technical and technological insights I gained during this experience. However, it is essential to highlight that it also presented a unique opportunity to observe the functioning of a company operating on a site housing several hundred employees.) This experience deepened my understanding of the organizational architecture of such a company, as well as the interactions between different departments and neighboring companies, which are critical for the execution of large-scale projects, such as the launch of the Euclid satellite.

At the beginning of my internship, I deliberately sought to step out of my comfort zone to make this experience as different as possible from what I was familiar with. Thus, among

the two robotics laboratories on the Turin site, I chose to join the one focused on 'On Orbit Servicing' instead of the one dealing with rovers and mobile robotics, which aligned more closely with my academic background from the previous year. By joining this laboratory, I had the opportunity to operate a robotic arm and a stereo camera.

The laboratory team I was welcomed into consisted of six young engineers, each specializing in various fields such as space engineering, system engineering, inverse kinematics and Cartesian control, artificial intelligence, computer vision, informatics tools, and more. They provided me with extensive support to explore potential projects and utilize the laboratory's tools and equipment. I gained invaluable knowledge through their guidance, assistance, and wealth of experience. The topics and objectives addressed during the internship were therefore developed based on the available equipment as well as the experience of my colleagues and their prior work.

1.2 Defining and planning the internship's objectives

Firstly, it is necessary to provide a precise introduction to the context of an On-Orbit Servicing (OOS) mission. Here, we are specifically interested in the initial stage of approaching a client satellite, see Figure 1. The objective is to establish contact between our satellite and the client satellite. Initially, we must determine the target's position and go through the different approach phases, from far to close range rendez-vous. Subsequently, as we attain the close surroundings of the target satellite, a robotic arm is deployed. Then, we have to identify the anchoring point on the client satellite using cameras and an illumination system on the servicing satellite.

This anchoring point is the subject of two scenarios, where the target may or may not be prepared for this type of intervention. In the case where the target is prepared, we can assume a more precise knowledge of the satellite's geometry, as well as the presence of markers that facilitate the final step. This final step involves using the robotic arm on the operational satellite to approach the client satellite and to establish a connection, it is referred to as the "capture" phase.

To correctly approach the anchoring point and close the gripper around it, the robotic arm uses the visual input gotten by the cameras system. Its role is to detect precisely and to compute the pose of the anchoring point on the target satellite relatively to the pose of the end-effector of the robotic arm. With this information, it is then possible to determine the control input for the robotic arm.

The development of the algorithm responsible for detecting and computing the pose of the anchoring point from a visual input was the work of my colleagues. In order to fully grasp the scope of their work, I needed to study the subjects I was less familiar with. The first step involved deepening my understanding of Image Processing and Computer Vision, which formed the core components of the algorithm's program. After a theoretical study, I constructed a software application designed to utilize a camera for object detection and pose computation in relation to the camera frame. The objective was to apply the computer vision algorithms I had studied and gain a deeper comprehension of the challenges of image segmentation.

Subsequently, we outlined clear objectives for the final month of the internship, during which I worked on controlling a robotic arm brought into the laboratory through a partner-

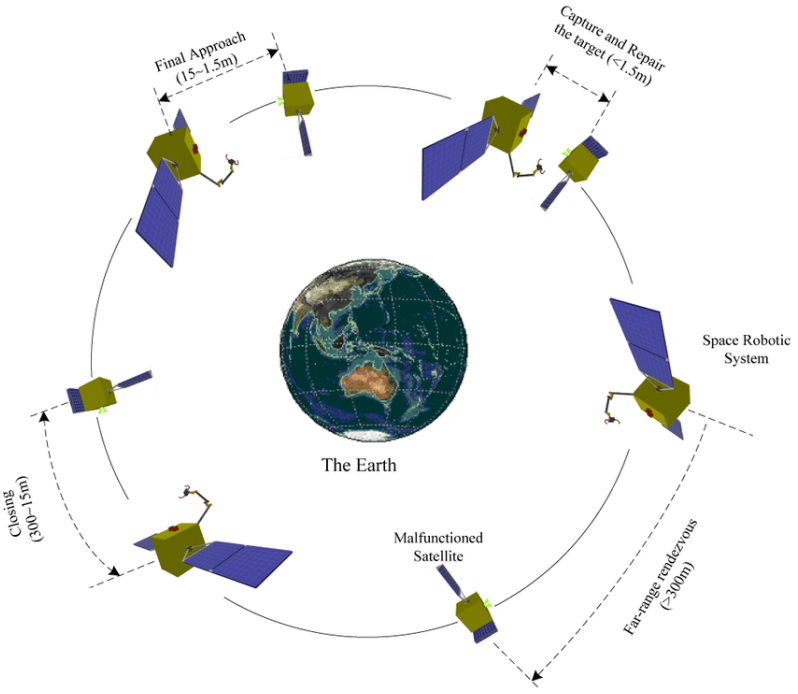


Fig. 1: A typical on-orbit servicing mission main steps

ship with the Italian Institute of Technology. The ultimate goal had to factor in the time constraints for me to become familiar with the arm's software and control, and was inspired by the thesis work of one of my colleagues. The assigned task was to develop an application enabling the arm to track a specific marker on a plane parallel to the camera's plane.

To follow this organization, this report will be divided into two sections. For each section, I have created a GitLab repository, from which select extracts will be included in the report and its annex, and that are also accessible online.

2 First Activities

2.1 Needed theoretical studies and CV context

As said previously, the first part of the internship was focused on computer vision, in order to estimate the pose of a target from one or several images. It was necessary to understand and study the state-of-the-art computer vision techniques, beginning with classes on the subject and simple application exercises.

Typically, when using images as a program input, the image goes through a pipeline with given processing steps, which ends with the extraction of the wanted information. Initially, it is best to do some Image Processing to detect some geometrical features, edges or corners, and filter and enhance part of the image, or select a region of interest. The next step, in the case of the work I did, is referred to as Image Matching, estimating the similarity between a pair of image. This process is part of the bigger problem of Object Detection in an image, which is still an important research subject nowadays, with a lot of challenges that computers

still struggle with: the scaling of objects, the changes in illumination, the object deformation or occlusion...

Apart from Neural Networks that started appearing approximately 10 years ago, the solution used to resolve this problem is to quantify common mathematical denominators between what we know about the target and a scene we've never seen before. There are three base operations that are needed in this process:

- Detection of feature points;
- Description of these key points;
- Matching between similar feature points across the images.

As an example, let's have a look on the pipeline used in an a case of satellite detection in the Figure 2.

Fig. 2: Object detection and Pose estimation pipeline

The first step is Image Processing with a blurring Gaussian Filter. Then the two following steps correspond to the Image Matching, and result with a region od interest supposedly containing the wanted object. Finally we estimate the pose of the detected object and use a Kalman Filter on the result.

2.2 Algorithms behind Object Detection

There are a lot of different methods conceived to resolve this problem, among them the *Scale Invariant Feature Transform* (SIFT) and the *Oriented FAST and Rotated BRIEF* (ORB) are some of the most well known algorithms. They are used to complete the two first steps of Object Detection: the detection and description of keypoints in an image.

2.2.1 SIFT algorithm

The SIFT algorithm was created two decades ago and its performance have been improved and critiqued thoroughly since then. It is still one of the main examples of object detection algorithms.

To summarize the important notions described before, the *keypoints* in SIFT are chosen by evaluating and refining sites with significant intensity changes in different scales of the image. It means that the keypoints can be used even with a change of scale of the scene, hence the name of the algorithm.

Once again simplified, a *descriptor* is a collection of vectors describing the surroundings of its keypoint, especially information about the orientation of the keypoint. It is computed

by computing the orientation of the pixels gradient and weighted by their magnitude, then doing the same for larger zones; keeping only the dominant gradient orientations. These descriptors are what is used to achieve robustness against illumination and rotation changes.

I used my student ID card as an example to compute and visualize keypoints and their descriptors computed by a SIFT algorithm. This computation can be easily done using the OpenCV python libraries, as is shown in the code extract in the Figure 3. The SIFT algorithm can be easily programmed, but its commercial use does require to pay a fee as it is patented.

```
1  def implement_SIFT(image):
2      """ A function using OpenCV libraries to program a SIFT algorithm,
           returning an image with a representation of the computed keypoints and
           descriptors.
3      """
4      grey = cv2.cvtColor(image, cv.COLOR_BGR2GRAY)
5      # creating a SIFT object with the wanted parameters
6      sift = cv2.SIFT_create(edgeThreshold=5, contrastThreshold=0.05)
7      # using the SIFT algorithm on the wanted image
8      keypoints, descriptors = sift.detectAndCompute(grey, None)
9      # displaying keypoints and descriptors found
10     result = cv2.drawKeypoints(grey, keypoints, image,
11         flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
12     return result
```

Fig. 3: SIFT Code application using OpenCV implementation

OpenCV created a specific class for the SIFT algorithm. On the 6$^{\text{th}}$ line in Figure 3, where the SIFT object is created, two parameters are entered. They influence the refining of keypoints by the algorithm. The *edgeThreshold* is used to filter the keypoints close to the image's edges and the *contrastThreshold* is for filtering out keypoints in low-contrast regions, see [4] for more details.

In Figure 4, we can see the graphic visualization of said keypoints and descriptors, with the circle and radius object corresponding to the scale and orientation of the keypoints.

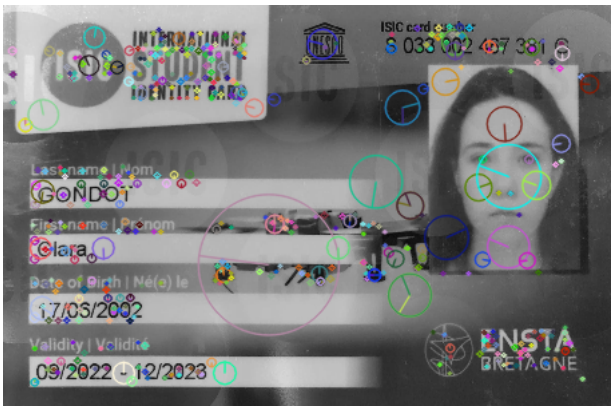


Fig. 4: Visual representation of keypoints and their descriptors computed by the SIFT algorithm

2.2.2 ORB algorithm

The ORB algorithm is more recent, it was created in 2011 as an alternative to SIFT and is free of use, unlike SIFT. It stems from other algorithms as its name suggests.

It uses the *Features from Accelerated Segment Test* (FAST) algorithm to compute keypoints: it classifies pixels by the intensity of their surroundings, keeping those with a major part of their neighborhood having a very high or very low relative intensity; then keeps only the local extremes with *non-maximum suppression*. It is several times faster than other existing corner detectors, but it is not robust to high levels of noise and depends on a threshold [5]. This method is not not resilient to rotation and scale changes, so ORB uses FAST on a multiscale image pyramid (which consists of different resolutions of the same image) to solve the scaling problem and adds an orientation to the keypoints by computing the Intensity moment of the zone and keeping the angle of the result.

For the descriptors, ORB uses the *Binary Robust Independent Elementary Features* (BRIEF) algorithm, tweaked to answer the rotation invariance issue. The BRIEF algorithms works with a fixed window pattern applied to around the keypoint and compares random pairs of pixels in this pattern. For each pairing, the algorithm assigns either a 0 or a 1 if the intensity of the first pixel is greater or lower compared to the second pixel. This process results in a binary descriptor, a format which allows high computation performances when using BRIEF descriptors. Since BRIEF is known for not working well with rotations, ORB does computes the BRIEF descriptor on a set of discrete orientations of the image. By comparing the keypoint's orientation to these discrete rotations, ORB creates rotation-invariant descriptors [7].

See below in the Figures 5 and 6 for a Python application of the ORB algorithm using again the OpenCV libraries and the resulting illustration of the keypoints and descriptors on my student ID card.

```
1  def implement_ORB(image):
2      """ A function using OpenCV libraries to program an ORB algorithm,
           returning an image with a representation of the computed keypoints and
           descriptors.
3      """
4      grey = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
5      # creating a SIFT object with the wanted parameters
6      orb = cv.ORB_create(nfeatures=1000, edgeThreshold=5, patchSize=5,
           fastThreshold=20)
7      # using the SIFT algorithm on the wanted image
8      kp, des = orb.detectAndCompute(grey, None)
9      # displaying keypoints and descriptors found
10     result = cv.drawKeypoints(grey, kp, image,
11         flags=cv.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
12     return result
```

Fig. 5: ORB Code application using OpenCV implementation

Thanks to the OpenCV implementation which relies in Python classes to create objects dedicated to the algorithms, the code implementation is very similar from one algorithm to the other. We also have a variety of parameters we can influence. Here we set *nfeatures*

to determine the size of the BRIEF binary descriptors, *fastThreshold* for the intensity comparison in the FAST algorithm, *edgeThreshold* to exclude the border of the picture when computing the keypoints and *patchSize* which needs to be close to *edgeThreshold*, see [3] for more details.

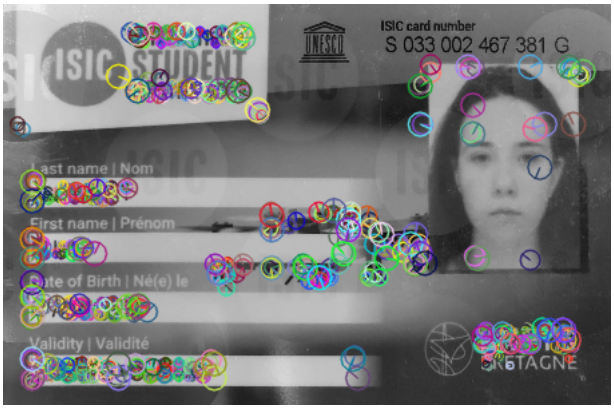


Fig. 6: Visual representation of keypoints and their descriptors computed by the ORB algorithm

2.2.3 Matcher algorithms

Once the keypoints and their descriptors are computed, you need a final algorithm to match them between two given images. During this internship, I had the opportunity to use two different algorithms: the *Brute Force Matcher* (BF matcher) and the *Fast Library for Approximate Nearest Neighbors Matcher* (FLANN matcher).

The BF matcher is a straightforward algorithm: you compare each feature from the first set to every feature in the second set, and computes the distance (for example Euclidean distance of the different criteria in the descriptors) between the two and finally chooses the closest match. It is simple however expensive in computation. See Figure 7 for its implementation.

```
1 def brute_force_matching(algo, kp, des, image, scene):
2     # BFMatcher with default parameters
3     bf = cv.BFMatcher()
4     kp2, des2 = algo.detectAndCompute(scene, None)
5     matches = bf.match(des, des2, k=2)
6     # keeping only first good matches
7     draw_params = dict(matchColor=(0, 255, 0), singlePointColor=(255, 0, 0),
          flags=cv.DrawMatchesFlags_DEFAULT)
8     result = cv.drawMatches(image, kp, scene, kp2, matches[:10], None, **
          draw_params)
9     return result
```

Fig. 7: Brute Force Matching application

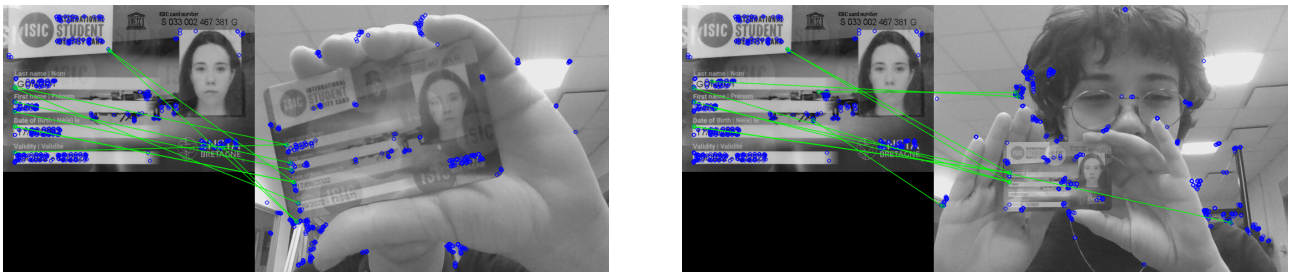


Fig. 8: Brute Force matcher and ORB descriptors results on two scenes with the same image reference.

The FLANN matcher is a more efficient way to compare the keypoints. It builds a multi-dimensional tree structure to accelerate the comparisons, but doesn't necessarily gets the nearest neighbor for the keypoints. See Figure 9 for its implementation.

```
1  def flann_matching(algo, kp, des, image, scene):
2      # FLANN parameters
3      FLANN_INDEX_KDTREE = 1
4      index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
5      search_params = dict()
6      flann = cv.FlannBasedMatcher(index_params, search_params)
7      kp2, des2 = algo.detectAndCompute(scene, None)
8      matches = flann.knnMatch(des, des2, k=2)
9      # keeping only good matches
10     matchesMask = [[0, 0] for i in range(len(matches))]
11     # ratio test as per Lowe's paper
12     for i, (m, n) in enumerate(matches):
13         if m.distance < 0.7 * n.distance:
14             matchesMask[i] = [1, 0]
15     draw_params = dict(matchColor=(0, 255, 0), singlePointColor=(255, 0, 0),
           matchesMask=matchesMask, flags=cv.DrawMatchesFlags_DEFAULT)
16     result = cv.drawMatchesKnn(image, kp, scene, kp2, matches, None, **
           draw_params)
17     return result
```

Fig. 9: FLANN Matching and Lowe's ratio test application

To get better results, we add another operation which filters out bad matches. We apply the Lowe's ratio to the distance of the 2 best matches gotten from the FLANN algorithm: if $\frac{d_1}{d_2} < 0.7$ then we keep the first match as a good one. This allows better results for the FLANN matcher, when we just keep the first 10 matches with the shortest distance for the BF matcher. The BF matcher is paired with the ORB algorithm and the FLANN matcher is paired with SIFT to get the results in the Figures 8 and 10.

2.3 Idea of application and architecture of the wanted project

The firsts half of the internship was focused on Computer Vision and the realization of an algorithm pipeline similar to the one in Figure 2. The objective was to detect the pose of

Fig. 10: FLANN matcher paired with SIFT descriptors results on the same two scenes.

my student ID card in a video flux using a single image reference, in a similar manner as the matching examples from before. I programmed this application as a ROS2 package, which takes as input an image of the target and an image of the scene in which we want to detect it and outputs the pose of the target. The package can be divided in three main functions:

- After choosing an algorithm between SIFT and ORB, detecting and describing keypoints from a given reference image. The keypoints and descriptors are then saved in a XML file and can be loaded at a later time.
- Receiving and processing the data from the camera: firstly detecting and describing keypoints in the image received using the same algorithm than before; then using a FLANN matcher to pair said keypoints with the one from the reference image.
- Computing the homography between the pairs of keypoints and deducing the pose of the object.

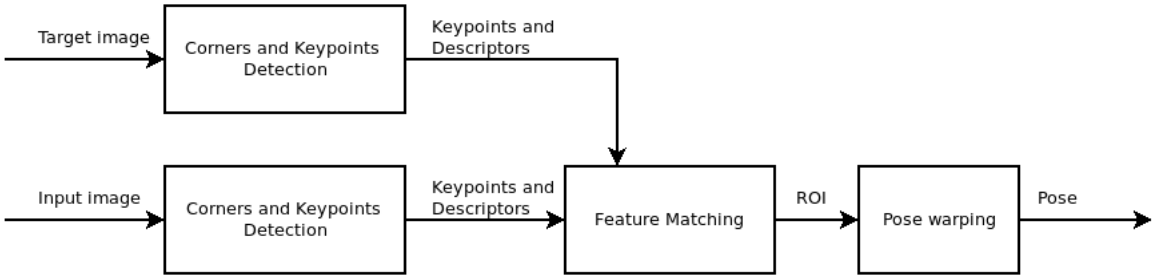


Fig. 11: Object detection and Pose estimation simplified pipeline

In this project, the input images are given by a Stereolabs ZED Mini camera. The Stereolabs cameras come with a very complete software and several third-parties integrations. This enabled me to use ROS2 to retrieve the data from the ZED Mini various sensors, precisely the left camera image and the distance computed with the stereo cameras from specific pixels. More details on the ZED camera are given in a later subsection.

The complete ROS2 package with instructions on how to install it can be found here: https://gitlab.ensta-bretagne.fr/gondotcl/zed-ros2-2023. From the three steps described earlier and the general pipeline, we get the architecture of the nodes and topics of the ROS2 package in Figure 12.

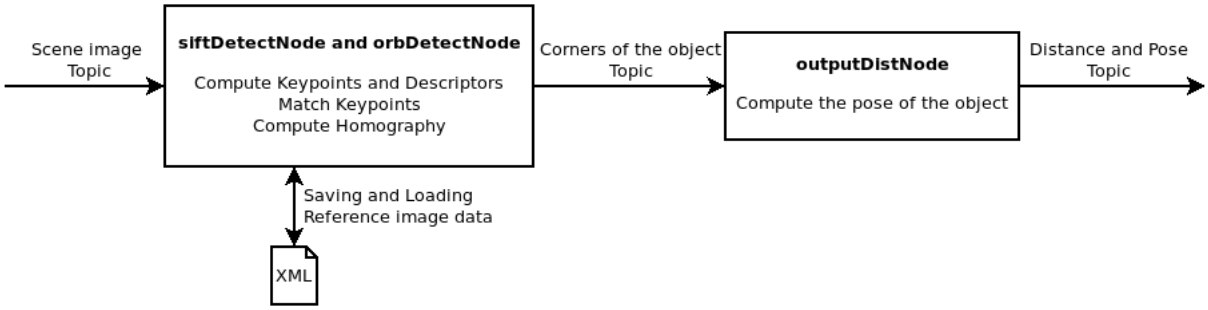


Fig. 12: Package *tag-detection* architecture

The nodes taking care of the computation of keypoints, their descriptors and the matching between the target image and the scene image work with similar programs than the one detailed in the last subsections, but the whole package is programmed in C++. The pose of the target has to be computed from the matches between keypoints that result from these algorithms: with multiple keypoints transformations from one image to the other, we can deduce the homography being applied to the target object reference pose and its pose in the scene image. The OpenCV library has a practical function doing this operation, as implemented in Figure 13.

```
1  std::vector<cv::Point2f> find_homography(
2          std::vector<cv::KeyPoint>& keypoints_object,
3          std::vector<cv::KeyPoint>& keypoints_scene,
4          std::vector<cv::DMatch>& good_matches,
5          std::vector<cv::Point2f>& object_corners)
6  {
7      //-- Localize the object and form two vectors of matching keypoints
8      std::vector<cv::Point2f>& object;
9      std::vector<cv::Point2f>& scene;
10     for (auto &good_match: good_matches) {
11         //-- Get the keypoints from the good matches
12         object.push_back(keypoints_object[good_match.queryIdx].pt);
13         scene.push_back(keypoints_scene[good_match.trainIdx].pt);
14         }
15     //-- Finding the homography matrix between the two lists of keypoints
16     cv::Mat H = findHomography(object, scene, cv::RANSAC, 3);
17     //-- Using the matrix to compute where the corners are in the image
18     //   The object corners correspond to the input image corners and are
           initialized once
19     std::vector<cv::Point2f> sorted_corners(4);
20     perspectiveTransform(object_corners, scene_corners, H);
21     return scene_corners;
```

Fig. 13: Computing the homography between two sets of keypoints

2.4 Computing the distance

Once the homography is computed, we have to compute the distance between the camera and the target. To do this we make use of the ZED Mini stereo cameras [1]. This operation is done by the *outputDistNode*. It takes three topics as inputs:

- The distance data sent by the ZED Mini as a greyscale image with the pixels intensity value corresponding to the distance computed by the camera. The intensity is null if the distance couldn't be computed.
- A mask with the zone in the image occupied by the target. The information is communicated as a binary image and the zone is a quadrilateral, which could be changed but works perfectly with my student ID card.
- A list with the four corners of the quadrilateral with the detected target.

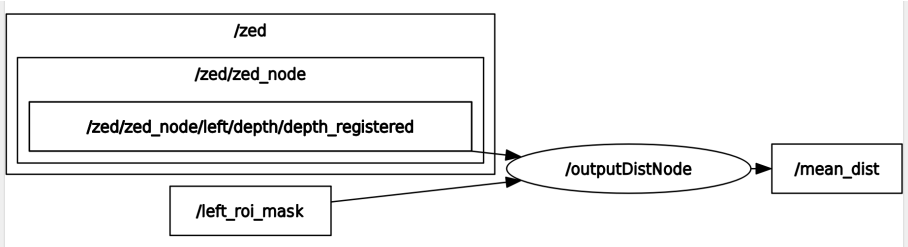


Fig. 14: The node graph around *outputDistNode*

With this information, we can either compute the center of the quadrilateral and publish the associated distance, or compute the mean distance for quadrilateral surface. The second method is useful to compensate any sensor errors, but the result can be corrupted if the zone defined is not accurate.

From the quadrilateral, it is also possible to get a frame for the target, with the x-axis along the longer border and the y-axis along the shorter border of the zone.

2.5 Results and commentary

To test out the performances of the package, I used my student ID card as before. We can criticize the performances of the object detection algorithms used.

The results I got while using the ORB algorithm to compute the keypoints were less conclusive than when using the SIFT algorithm. Firstly, these algorithms are mainly used to quantify the changes between two scenes and can be evaluated only for their resilience to rotation and illumination variations, as in the dataset shown in the Figure 17. Using a cropped image of the target was giving the matching algorithms less useful keypoints and descriptors. It was difficult to get better results, even after tweaking the parameters of the detection and matching algorithms. Another option to better the performances for the keypoints detection and descriptors computation would be to fine tune the pre-processing of the input images. Because this method of object detection was proved difficult to implement, in the next part of the internship a specific marker was placed on the object to detect, making this step easier to handle.

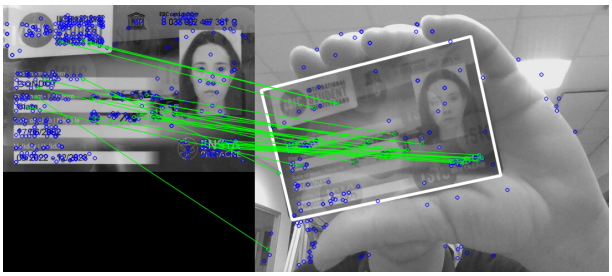
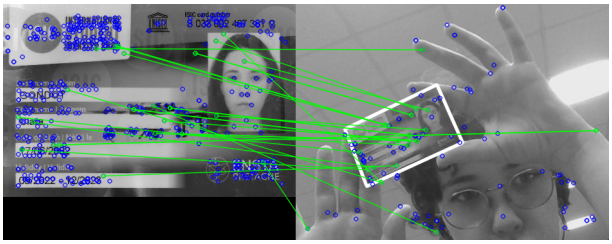


Fig. 15: Computing the homography using the matches computed previously with SIFT (left) and ORB (right).



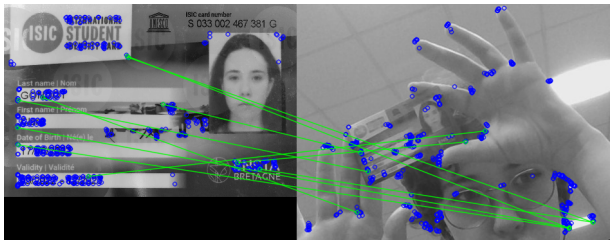


Fig. 16: Same as previously but with another scene.

3 Controlling a Robotic Arm using Visual Pose Estimation

3.1 Details of the project and equipment

During the second part of the internship, I focused on using visual servoing to control a robotic arm. The objective was to simulate the final approach phase of an On-Orbit Servicing mission, during which the arm is deployed and a vision algorithm is used to control it towards the client satellite. Firstly, I had to set up a camera and an object detection algorithm with capable of outputting the pose of the target. Then, the major part of the work consisted in familiarizing myself with and controlling the robotic arm.

3.1.1 Stereolabs ZED Mini

The ZED Mini was used before for the video input for the feature detection in the precedent part. It was also useful for the Pose Estimation. The ZED Mini comes with a software that does a lot of computation and pre-processing of the different sensors data, before publishing them in a collection of ROS2 topics. Here is a list of the features that were used in the project:

- The ZED Mini has an inertial unit and publishes its pose on a *TF2 tree*. TF2 is a ROS2 tool used to handle transformations between coordinates frames.
- The stereo camera of the ZED Mini allows to compute a distance map associated with the images. You can easily check if the distance associated with a pixel of one of the cameras image was computed correctly and retrieve it.

Fig. 17: Extracts from the HPSequences dataset displaying two viewpoint (top) and two illumination (bottom) based sequences. [6]

- Like in the precedent project, I used the topic on which the video stream of the left camera if published.

The camera was fixed to the robotic arm using a 3D printed piece keeping it on top of the end-effector, see Figure ?? below.

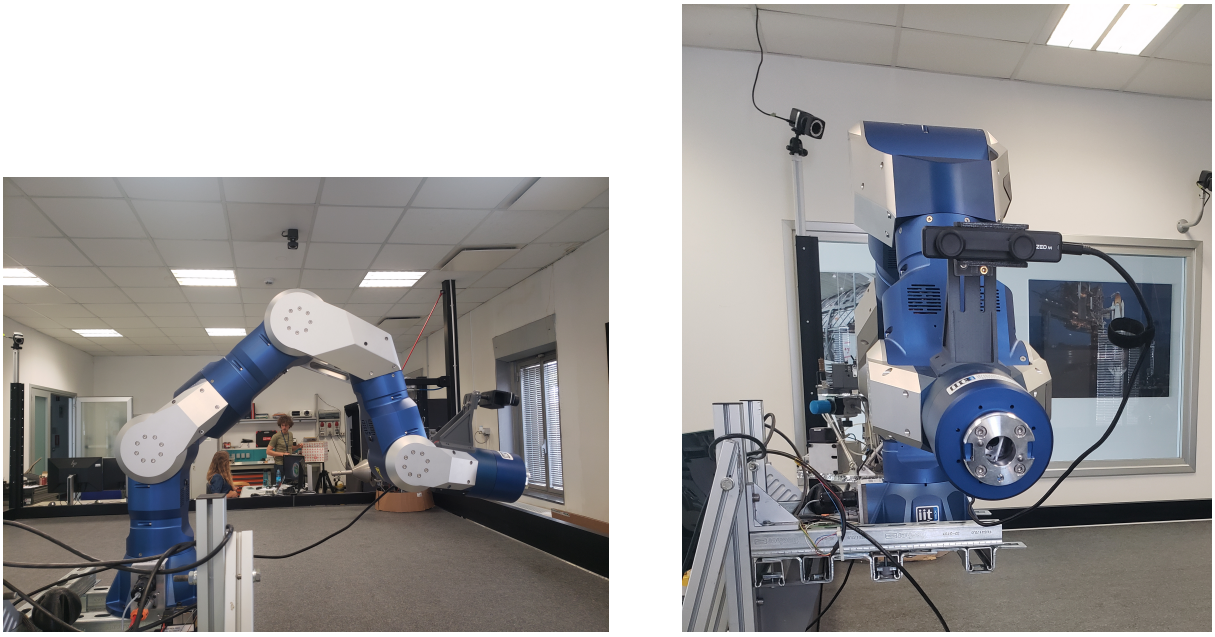


Fig. 18: Photographs of the camera placement on the arm.

3.1.2 Marker Detection

To make detecting the target and computing its pose easier, we used an ArUco marker. This marker library is commonly used for camera pose estimation and an OpenCV library

detecting these markers already exists [2]. This type of marker is easily detectable, because it is composed of highly contrasted edges. To compute the pose, the algorithm needs its pattern and size as an input.

For this project, I used a Git repository available here: https://github.com/lapo5/ROS2-Aruco-TargetTracking. Its forked version with my modifications is available here: https://github.com/GondotCl/ROS2-Aruco-TargetTracking/. It contains a ROS2 package to which I input the topic with the ZED Mini left camera image and creates multiple topics corresponding to the marker presence, corners and pose.

3.1.3 IIT Robotic Arm

The Arm present in the laboratory was brought here by the Humanoid and Human Centered Mechatronics Research Line of the Italian Institute of Technologies. It has 7 degrees of freedom and joints and measures approximately 1.2 meters. It is placed above a marble block on which dummy targets can circulate.

Understanding the arm's operation was quite difficult, as there was 2 more softwares integrated in its computer, that needed to cooperate with ROS, the software I used for my control implementation (see Figure 22).

First, the Robotic Arm came with a custom software *XBotCore*, made in the IIT. It is both:

- Used for Real-Time control of the robot, taking charge of the final layer of communication with the harware and ensuring the command.
- A middleware, configurable by adding programs named *Plugins*, that can interact and communicate with other softwares such as ROS.

This duality made the role and possibilities of *XBot* hard to totally understand. In the end, I chose to program my project with ROS as it was more configurable and complete as a middleware, as some functionalities were difficult to implement in a XBot Plugin.

The second software on the robotic arm is *CartesIO*, used for solving the mathematical equations of Inverse Kinematics that appear when using Cartesian Control to command the pose of a part of a robot, in our case the end-effector of the robotic arm. *CartesIO* includes:

- A programmatic API, allowing to reuse its tools in Python and C++ programs.
- A ROS based server, with launch files provided that loads the robot description and a Cartesian controller.
- Different control modes, using the joints impedance, torque or pose.

With this software, we can input the wanted pose of the end-effector and it computes all the corresponding positions and orientations of each joints, as well as sending them to XBot. As illustration of the software capabilities, we can launch a ROS node with a RVIZ graphic user interface in which we can control the end-effector position and orientation. In the Figures 19 and 20, we can see how the position of the entire arm is computed and controlled by the software to satisfy the end-effector pose change.

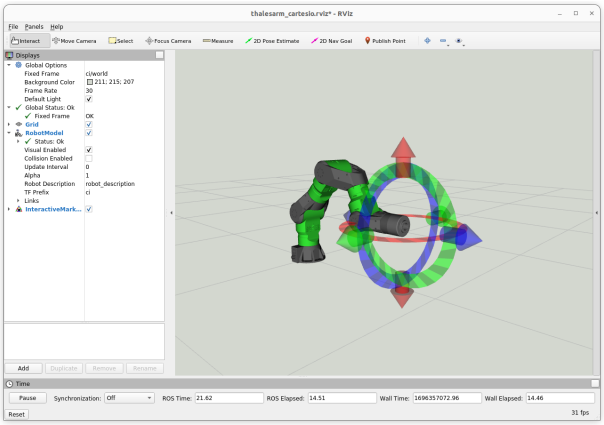


Fig. 19: CartesIO's RVIZ interactive visualization, arm in its homing pose

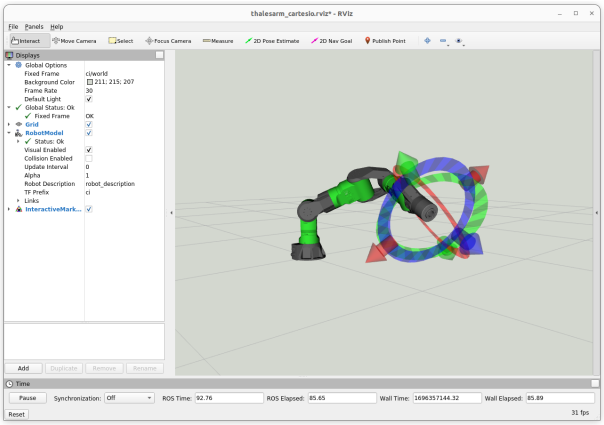


Fig. 20: CartesIO's RVIZ interactive visualization, after interaction with the end-effector pose

The robot's computer could also launch a ROS1 master node, which was used to communicate with another computer handling the computer vision. A ROS bridge package between Noetic and Foxy was necessary to communicate to the arm's computer and translate specific XBot messages especially.

3.1.4 Project's architecture

The ZED Mini was connected to the same computer that was used for the Computer Vision studies via a HDMI cable. For the remainder, there are two scenarios, if we are working in a simulation or on the real arm. For the simulation, the computer communicates with Gazebo, a *ROS2* application, in which a scene was programmed to communicate with the arm software and simulate its movements. When using the real arm, the first computer has to be connected with the arm computer via Ethernet. In both cases, the commands are computed and sent by the first computer on a ROS2 topic. On the arm's computer, a ROS1 Master Node is launched on start and is used to communicate with my own ROS2 programs via the ROS bridge.

3.2 Software implementation

3.2.1 Package architecture

As said earlier, ROS2 was used to send commands to the robotic arm. I created a four packages available here: https://gitlab.ensta-bretagne.fr/gondotcl/marker_tracking_pkg.

- The first package, *custom_interfaces*, contains 3 messages used by the other packages: *PointsList*, *DataStatus* and *Direction*.
- The second package, *marker_replacement*, is used to test the other packages and publishes topics mimicking the *ROS2-Aruco-TargetTracking* package.

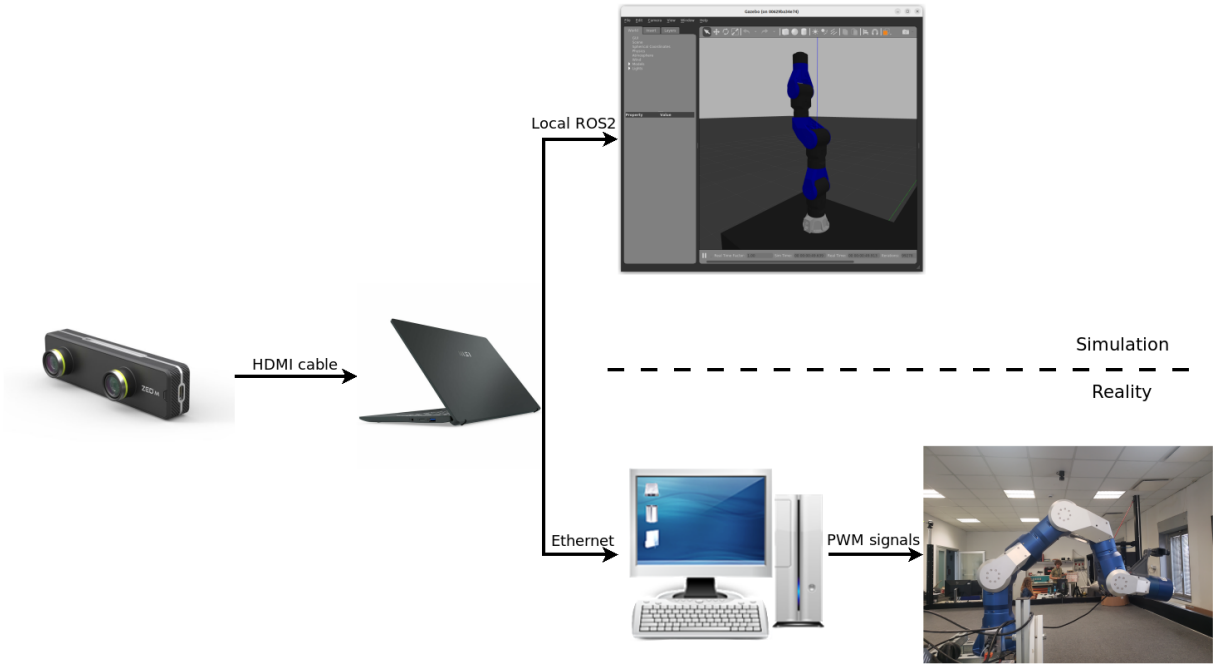


Fig. 21: The high-level electronic architecture.

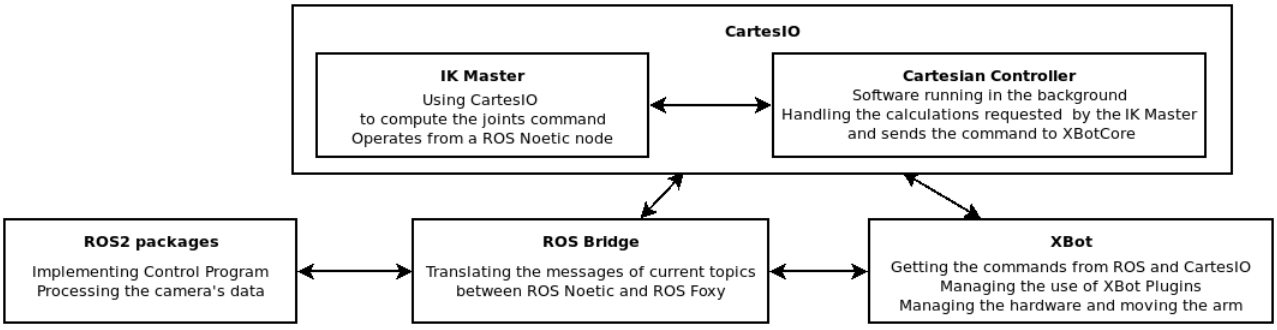


Fig. 22: The interactions between the different softwares involved to control the arm

- The third package, *pose_ manager*, is where the images and other topics are processed to compute and send the command. There are two alternative nodes: *marker_ positioning* and *arm_ transformation*, each using a different way to compute the wanted pose for the end-effector. They are detailed in the next part.
- The last package, *arm_ control*, is communicating with CartesIO, creating a specific object named *CartesianInterface* to do so. It is a bridge for *geometry_ msgs* to this interface. Because of compatibility issues, it was necessary to separate this Cartesian-Interface and any TF2 usage. TF2 is used in the *pose_ manager* package and then a simple geometry_msg is send for each pose we want to transmit to the interface with CartesIO. The package contains two nodes, *set_ pose* is a minimal one for testing purposes and *cartesian_ interface* which is connected to the rest of the project. The *cartesian_ interface* node is:
  - connected to the direction input on the *arm_ direction_ order* topic
  - connected to the marker pose on the *base_ link_ to_ marker_ transformation* topic
  - connected to the marker presence topic
  - publishing the end-effector pose on the topic *end_ effector_ pose* for TF2 to use in another package

In the following sections, I detail the two approaches explored to control the robotic arm.

3.2.2 Following the image center

The first option was to keep the marker in the center of the left camera image. By initializing the arm in its homing position and having the target on a vertical plane, we can admit that the frame of the marker and the camera are parallel. Since the camera and end-effector frames are fixed with respect to each other, the wanted transformation for the end-effector frame is then simplified to a translation in the plane. The command is computed in the *marker_ positioning* node, that takes the image size and the marker corners position as entries and outputs a combination of directions, between “up”, “down”, “left” and “right”.

To decide which direction to input the arm, we use the lower right and upper left corners position on the image and distinguish between different cases to see if the marker position is satisfying. Ideally, the corners have to be on each side of the picture like the (1) case in the Figure 24. But as a security we accept than interval of 8 pixels around the horizontal and vertical lines splitting equally the image, making the case (2) also acceptable.

From these comparisons, it is possible to know which direction the end-effector has to translate in to get the marker closer to the center. This information is published on a Direction topic, with the Direction message being an ensemble of 4 booleans corresponding to the 4 possible directions.

Fig. 23: The disposition of the arm and marker at the start of the test with associated frames

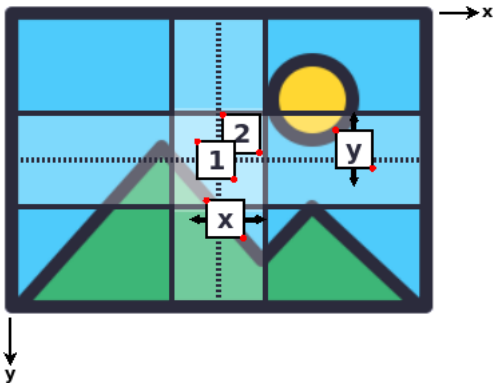


Fig. 24: Illustration of the case distinction

To command the arm, we have to send to *CartesIO* the new wanted pose. With T_{init} and T_{target} the translations between the end-effector and the base link of the robotic arm before the command and that we want to attain, we have:

$$T_{target} = T_{init} + q_{init} * \delta * dT$$

where $\delta = 0.1$ meters, q_{init} is the quaternion for the rotation between the end-effector and the base link and

$$dT = \begin{pmatrix} \begin{cases} up: & 1 \\ down: & -1 \end{cases} \\ \begin{cases} right: & 1 \\ left: & -1 \end{cases} \\ 0 \end{pmatrix}$$

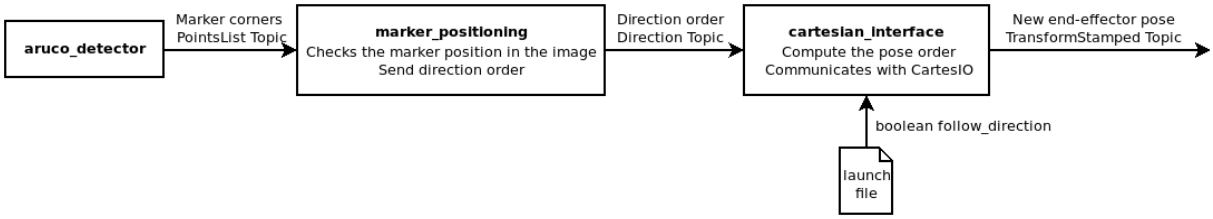


Fig. 25: Architecture around the marker_positioning node

To summarize, you can refer to Figure 25 with the different relations between nodes and topics below.

3.2.3 Aligning the frames

Since the ROS package detecting ArUco markers returns the frame associated with the marker, we can consider a more advanced control approach. I implemented a new ROS node for a command that aligns the gripper's frame on the straight line between the robot base and the marker center, allowing it to "point" toward its target. This situation is illustrated in the Figure 26.

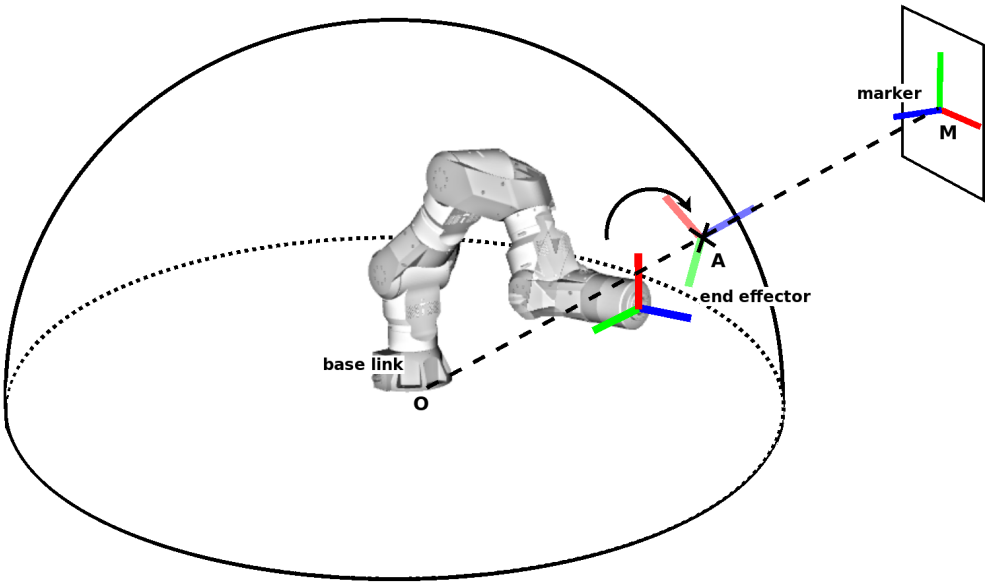


Fig. 26: Illustrating the target pose when aligning the end-effector frame to the marker frame

To compute the new pose for the end-effector, with $\overrightarrow{OM}$ the vector between the base link origin and the marker center and $r = 0.9$ meters the range of the robotic arm, we have the following translation T_{target} between the base link and the end-effector:

$$T_{target} = k * \frac{\overrightarrow{OM}}{||\overrightarrow{OM}||}$$

where

$$k = \begin{cases} ||\overrightarrow{OM}|| - 0.1 & \text{if } ||\overrightarrow{OM}|| < r \\ r & \text{else} \end{cases}$$

Then we need to compute the rotation between the end-effector frame and the base link frame and express it as a quaternion. In our application, we want the z-axis of the end effector pointing in the direction of the $\overrightarrow{OM}$ vector. Let's define $\overrightarrow{z_O}$ the z-axis of the base link frame, which is also considered the reference frame, and $\overrightarrow{z_A}$ the z-axis of the end-effector frame where A if the center of this frame. If $\overrightarrow{u} = \overrightarrow{z_O} \wedge \overrightarrow{z_A}$, then the quaternion corresponding to the rotation between the two frames is:

$$q = \begin{pmatrix} u_x \\ u_y \\ u_z \\ \overrightarrow{z_O} * \overrightarrow{z_A} + \sqrt{||\overrightarrow{z_O}||^2 * ||\overrightarrow{z_A}||^2} \end{pmatrix}$$

We want $(A, \overrightarrow{z_A})$ // (OM) and $A \in (OM)$, which means that $\overrightarrow{z_A} = \frac{\overrightarrow{OM}}{||\overrightarrow{OM}||}$. Since $\overrightarrow{z_O} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$, we can easily compute $\overrightarrow{u} = \begin{pmatrix} -z_{A,y} \\ z_{A,x} \\ 0 \end{pmatrix}$. Finally, since $||\overrightarrow{z_O}||^2 = ||\overrightarrow{z_A}||^2 = 1$, we have the wanted quaternion q_{target} between the two frames:

$$q_{target} = \begin{pmatrix} -z_{A,y} \\ z_{A,x} \\ 0 \\ z_{A,z} + 1 \end{pmatrix}$$

In the program, we get the $\overrightarrow{OM}$ via the ROS tool *TF2*. To use TF2 efficiently, we need to set up a proper hierarchy between the involved frames. The frames hierarchy can be visualized in the Figures 27 and involves the following interfaces:

- A static transformation is published by the *pose_ manager* package between the world and the base link.
- The *arm_ transformation* node broadcasts to TF2 the transformation base link to end-effector from the data published by the *cartesian_ interface* node and publishes the transformation base link to marker to a topic using a TF2 listener.
- The ZED Mini ROS package publishes a static transformation between the "TCP" and the left camera optical frame.
- The ArUco tracking package publishes the transformation between the left camera optical frame and the detected marker.

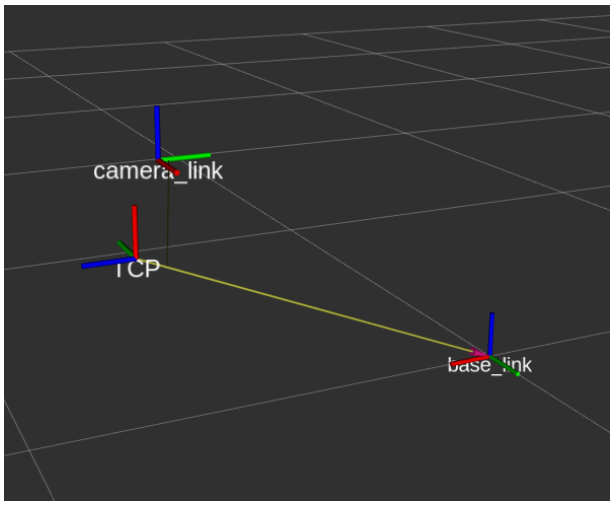


Fig. 27: TF2 architecture of the base, end-effector (TCP) and camera visualized in RVIZ

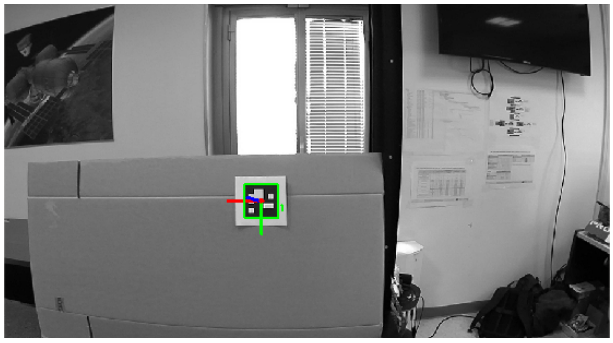


Fig. 28: TF2 frame of the marker visualized by the camera

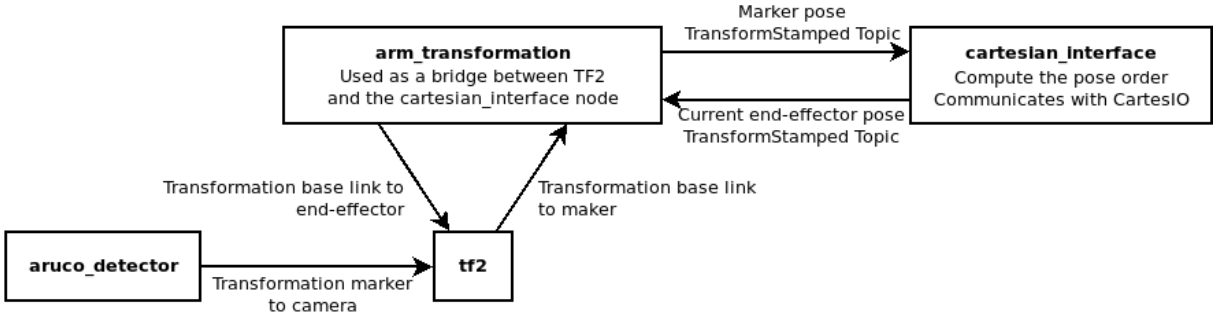


Fig. 29: Architecture around the pose control using TF2

The *cartesian_interface* node is the one computing the wanted pose for the end-effector once all the necessary poses are published. The $\overrightarrow{OM}$ vector is evaluated from the base link to marker transformation. Then it computes and sends the pose command to the CartesIO using the equations detailed before. The summary of the ROS architecture around this command is illustrated in the Figure 29 below.

3.3 Results and commentary

I was able to test out the first method to control the arm. Firstly in simulation, I tested if the *marker_positioning* node computed the right orders, by inputing the camera image and seeing what was published on the Direction topic, see Figure 30 where the direction order is to move left and up.

The second step was to send a direction order to the arm and see if it follows it, a screen capture of this test can be found here: https://www.youtube.com/watch?v=UgLrClH9nI0. Finally, I was able to to do a quick test of the algorithm on the real arm. The results can be

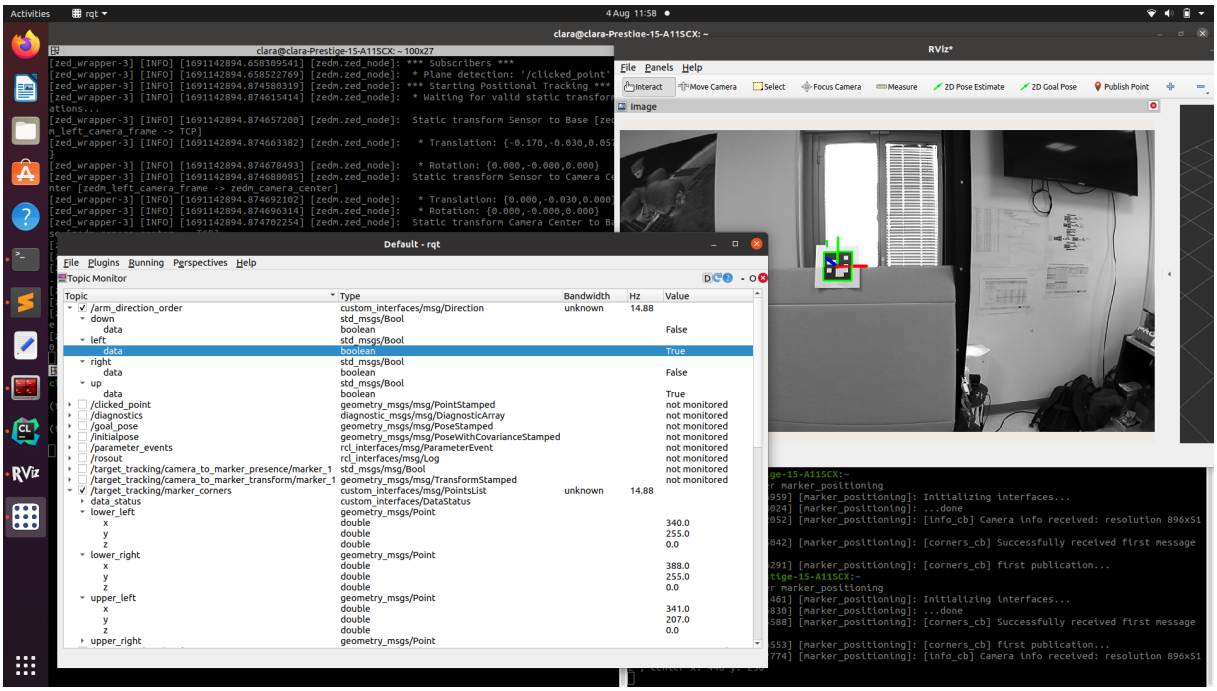


Fig. 30: Monitoring the orders sent after receiving the marker still image

seen here: https://youtu.be/CxWkDPXzIuA.

In the given situation where the marker is on a plane parallel to the camera plane, the algorithm does work. With more time and extensive testing, the program could be improved to work in any starting situation, by adding a sequence where the arm searches for the marker.

The second method was more difficult to implement and I only have a recording of the simulation of the order on RVIZ, where I move a marker frame via the *key_teleop* ROS package: https://youtu.be/CxWkDPXzIuA.

Conclusion

During this internship experience I had the opportunity to work on various aspects related to computer vision and its applications in the context of On Orbit Servicing.

Throughout these four months, I delved into the intricacies of computer vision, gaining knowledge in various concepts relevant to camera pose estimation and applied in the context of OOS. I explored the fundamentals of object detection, including algorithms like SIFT and ORB, and developed a solid understanding of matcher algorithms. Additionally, I was introduced to the promising fields of convolutional neural networks and artificial intelligence, which are poised to play a pivotal role in the future of visual servoing.

Having the chance to use the laboratory equipments during practical applications, I managed large code projects, which involved integrating and layering different tools. This experience allowed me to grasp the significance of maintaining code tidiness and staying motivated while working independently on a project. Both these skills and the knowledge I acquired will undoubtedly prove invaluable in my future work and academic pursuits.

Lastly, my time at the company provided me with valuable insights into the workings of a large organization. I observed how work rhythms vary depending on one's proximity to technology and administration. I also recognized the significance of effective communication and collaboration among team members, witnessing the diverse roles required for the success of a project while working alongside my colleagues.

Overall, this internship has not only enhanced my technical knowledge but also broadened my horizons in terms of the broader aspects of project development and management. I am now better prepared to pursue my career goals with a solid foundation in computer vision and a better appreciation for the dynamics of the corporate world.

References

[1] Depth Sensing Overview | Stereolabs.

[2] OpenCV: ArUco marker detection (aruco module). https://docs.opencv.org/4.x/d9/d6d/tutorial_table_of_ content_aruco.html.

[3] OpenCV: cv::ORB Class Reference. https://docs.opencv.org/3.4/db/d95/classcv_1_1ORB.html.

[4] OpenCV: cv::SIFT Class Reference. https://docs.opencv.org/4.x/d7/d60/classcv_1_1SIFT.html.

[5] OpenCV: FAST Algorithm for Corner Detection. https://docs.opencv.org/3.4/df/d0c/tutorial_py_ fast.html.

[6] Kristijan Bartol, David Bojanić, Tomislav Pribanić, Tomislav Petković, Yago Diez Donoso, and Joaquim Salvi Mas. On the Comparison of Classic and Deep Keypoint Detector and Descriptor Methods. In *2019 11th International Symposium on Image and Signal Processing and Analysis (ISPA)*, pages 64–69, September 2019. arXiv:2007.10000 [cs].

[7] Deepanshu Tyagi. Introduction to ORB (Oriented FAST and Rotated BRIEF), April 2020. https://medium.com/data-breach/introduction-to-orb-oriented-fast-and-rotated-brief-4220e8ec40cf.