Ensta
2, rue François Verny
29806 Brest cedex
France
Tel +33 (0)2 98 34 88 00
www.ensta-bretagne.fr

Rapport de stage de 2e année

FISE 2026

25 octobre 2025

V4R - Development of a gaze-tracking module

Yasmine RAOUX - BARKOUDAH

yasmine.raoux@ensta.fr

Acknowledgements

First, I would like to sincerely thank my internship supervisor, Markus Vincze for giving me the opportunity to work at the Vision for Robotics lab and for his kind guidance throughout the internship.

I also really appreciate Peter and Matthias for supervising me during this internship and for always being available to help me and to answer my questions when I needed it.

And finally, a big thank you to all the other members of the lab for being so kind and for making me feel welcome during my stay.

Abstract

This internship took place at the ACIN (Automation and Control Institute) of the Technological University of Vienna (TU Wien), within the Vision For Robotics (V4R) research group [1]. The main goal was to design and evaluate a perception module capable of estimating a human's gaze and pointing direction from RGB-D data to complete Human-Robot Interaction (HRI).

The work was divided into two phases. The first phase focused on implementing a gaze detection algorithm using the MediaPipe framework and projecting the estimated direction into the robot's 3D coordinate frame. Multiple experiments were performed under different lighting and positioning conditions. However, due to the low resolution of the robot's camera and poor lighting, the face detection frequently failed, which made difficult the proper evaluation of the gaze estimation module.

The second phase involved combining the gaze module with an existing pointing detection algorithm into a unified, ROS-based pipeline. The pointing module, which relies on the user's right arm landmarks, was adapted and integrated for data fusion. Their respective accuracy was tested by an analysis of the angular error between the detections.

Despite the limits set by the hardware and environment, the project successfully delivered a modular and reusable system architecture. This setup provides a starting point for future research in multimodal perception at the institute.

Table of Contents

Introduction

The growing field of collaborative and assistive robotics requires robots to interact more naturally with humans. A key part of successful Human-Robot Interaction (HRI) is the robot's ability to understand non-verbal instructions, like gaze and pointing gestures. These signals are important for a robot to determine what an operator is paying attention to and what their intent is during a shared task. While general gaze detection tools are available, making them work reliably in 3D, it's still a challenge in real time and in a complex environment.

This internship was completed with the Vision For Robotics (V4R) research team [1]. The V4R group already had tools for gesture and object detection, but they needed a reliable system for visual attention (gaze). Therefore, the main goal of my work was to design, implement, and test a gaze detection algorithm using MediaPipe on RGB-D images. This module was then combined with the existing pointing detection module. The final aim was to create a robust system that can estimate the user's combined attention vector in 3D, making the interaction smoother.

During the project, I explored different ways to process visual data, integrated the system into the robot's ROS framework, and ran tests on datasets and the physical robot. As expected, integrating these systems was challenging, mainly due to the robot's camera limits and the lighting conditions. However, I was able to develop a modular and reusable architecture that can be built upon in future research.

The rest of this report is organized as follows : Section 2 covers the institute's background and project goals. Section 3 details the tools and methods used, especially the MediaPipe framework and the HSR robot. Section 4 describes the implementation steps for gaze detection and 3D projection. Finally, Sections 5 and 6 present the results of the evaluation, the combined gaze/pointing analysis, and ideas for future work in multimodal HRI.

1 Context

1.1 V4R and its research focus

I did my internship at TU Wien (Technische Universität Wien), a technological university located in Vienna, Austria. I worked with the Vision for Robotics (V4R) research group, which is part of the ACIN institute (Automation and Control Institute). The internship began in early May and ended early September.

The V4R research group focuses on developing computer vision methods that allow robots to understand their environment, perceive objects, and act autonomously in everyday scenarios. The robots can later be used in manufacturing or household settings, for example. They develop robotic perception to achieve real-world interactions, like object modeling and manipulation, or human-robot interactions using semantic scene understanding. They can also apply it to safe navigation or the manipulation of known objects, as well as unknown ones.

There are around 15 people working in the lab, including researchers, PhD students, and a few master's students working on their thesis. The team is very international, with members from various countries and backgrounds, and the working language is mainly English, which helped me integrate easily and feel comfortable in the lab.

I worked on-site at the lab most of the time, even though it is possible to work remotely, thanks to GitHub and other collaborative tools. I shared an office with two other researchers, although they were'nt always present, which gave me a lot of autonomy. However, I could always reach out for help or feedback from my two direct supervisors, Peter Hönig and Matthias Hirschmanner.

Peter is involved in the MANiBOT project [2], which aims to develop bimanual mobile robots capable of performing complex manipulation tasks in environments such as airports or supermarkets. His work focuses on cognitive functions and physical intelligence in robotics. Matthias, on the other hand, works in robotics, but more from a social and healthcare perspective. His project is called Caring Robots [3], and it explores how robotic technology can be designed to support rather than replace human interaction. This is why he collaborates with both caregivers and people receiving care.

The lab has access to various robotic equipment, including Sasha, a human support robot (HSR in short), used for manipulation tasks, and another platform currently being built by Matthias [4]. The team also publishes papers regularly, and attends major international conferences, so they are quite involved in the academic community.

I chose this lab for different reasons. First, on an academic level, the V4R group has a strong reputation in both robotics and computer vision. I wanted to challenge myself by discovering a domain I had less experience in, because even though I had worked on several robotics projects, they were mainly focused on control and navigation. I had very limited experience in vision-based approaches and applications of AI in general. Moreover, I was really interested in Vienna as a city, so it also had an influence on my choice of internship.

1.2 Project selection and objectives

Unlike most of my classmates, I arrived without any predefined subject. At first, I found it a bit intimidating because I didn't have a clear idea of what I was going to do or work on. I spent the first week of my internship discovering the lab and their different research areas. To help me do so, I was given a list of older master thesis subjects. They included different concepts and methods of vision analysis such as vision transformers (ViTs), that work by dividing images into patches and use self-attention to capture spatial relationships between the patches [5]. I also discovered semantic flow, which is used a lot in the lab. It finds pixel correspondences between two images that share semantically similar content. It's really useful to estimate the changes in view points for example. It's used for object pose estimation, which is the process of determining the position and orientation of an object in space, using usually 3D data from a sensor or from an image.

I was really interested in projects involving semantic flow, however, given the fact that I had no prior experience in computer vision, it wasn't really realistic to start with such an advanced topic given my level. The following week, we had a team meeting during which I introduced myself and my background and past projects. The team then gave me ideas of subjects I could work on, and I was happily surprised by how open they were to letting me choose a subject I was interested in. One suggestion was a navigation subject, however I was already familiar with this kind of topic and wanted to take the opportunity to work on something new that could combine both robotics and computer vision.

From all of their proposal, they suggested implementing a gaze detection module, and that's the subject I felt most interested in. Matthias had already started developping a pointing detector, so I could be inspired by the work he had already done and apply similar methods to gaze detection [6]. The goal would be to combine gaze direction with head rotation to get the 3d gaze direction. This could then be integrated into Matthias work with his healthcare robot for example, to improve human-robot interaction by allowing the robot to understand where a user is looking.

2 Presentation of the subject

The goal of my internship is to develop a gaze tracking algorithm, to verify its accuracy and to implement it in the existing detecting and grabbing pipeline so that it can be used in more complex robotics systems over time. The pipeline involves detecting objects, estimating their positions, and enabling the robot to grasp them or to interact with them. The main goal would be to detect with accuracy where the user is looking, so that by applying object detection we could for example determine what object the user intends to interact with, like finding which one the robot has to grab for example. The objective is to use the camera already installed on the robot, so in my case, as I am using Sasha (the HSR robot), it is an RGB-D camera. The sensor provides both the color images (RGB) and the depth images, which helps localizing objects in 3D space. The camera is mounted on the robot's head, which can turn to almost 360°.

To implement the gaze tracking algorithm, I chose to use MediaPipe Solutions, as it was used for the pointing detection, so I have a pre-existing idea of how to make it work. It is an open source project developed by Google, and it provides libraries and tools, such as pre-trained and ready-to-run models which I found more interesting and useful in my case. I will present it in more details in the next section.

Another idea we had for the next step of the project was to, once the gaze detection would work, combine both gaze and pointing detection to make a more complete detection algorithm that would be able to switch between methods of detection. For example, gaze tracking would be preferred when the user is further away, and pointing detection would be used when the whole body would be detected. It would make a more natural and useful human-robot interface.

3 Tools and methods

3.1 HSR robot (Sasha)

Sasha is a "Human Support Robot", made by Toyota [4]. It was designed for assisting people in daily-life environments. It has been at V4R since 2019, and it has been used for research in image processing and object recognition, and they use Sasha to test their algorithms, to grasp objects and to move around the lab.

Sasha is approximately 1 meter tall, and was built on an omnidirectional mobile base. To do so safely, it's equipped with different sensors, including a laser measuring range sensor and a magnetic sensor for stopping. It has a single arm equipped with a mutli-fingered gripper, with suction pad, a hand force sensor and a hand camera, which makes it capable of grasping and manipulating light objects.

FIGURE 1 – Toyota HSR used in the V4R lab.

Moreover, its head can rotate to almost 360°, and different sensors - and RGB-D sensor, stereo and wide angle cameras and a microphone - are mounted on it. Lastly, it has approximately 2 hours of battery autonomy under usual operating conditions, and a built-in display screen on the head.

The joints are all equipped with absolute-type joint angle encoders, which make the movements more precise and easily controlled. In total there are 11 controllable joints, each associated to its own coordinate system.

Although it is possible to use a simulator based on Gazebo, I didn't need to use it for my project and I was able to start experimenting directly with Sasha, using a ROS2 environment, detailed in the following section.

To interact with Sasha, it is also possible to use a web interface via its IP address. It provides a live overview of the camera feed as well as a control panel to directly command Sasha's movements. It's possible to control the head rotation angles, make the robot move in every direction, and to adjust the height of its head. It was very useful at the beginning to test Sasha's abilities and to determine what was possible without having to write code directly.

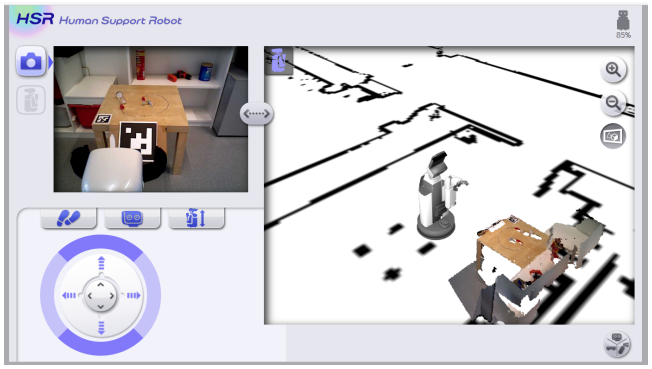

FIGURE 2 – Web interface to control Sasha's movements

3.2 MediaPipe Solutions for gaze tracking

MediaPipe Solutions is an ensemble of libraries developed by Google. It is part of the MediaPipe open source project, and all the code is available on GitHub [7]. It provides pre-trained models for a variety of solutions. For example, it is used for generative AI tasks such as image generation or function calling as well as vision tasks, which is what interested me for my project. It provides a wide range of models including image classification and segmentation, gesture recognition, face tracking, object detection, and more.

All the models are accompanied by code examples for Android, Python an Web applications. I used Python for the entirety of my project, as it was compatible with ROS2.

I used for the most part the face landmark detection. It is composed of three machine learning models : - a face detection model, which detects the presence of faces - a face mesh model , which adds a complete 3D mapping of the face - a blendshape prediction model, which is used for recognizing face expressions

These models can work both on single images and continuous stream of images, which was my case. This feature is particularly important for real-time applications like gaze-tracking.

I used the face mash model, which gives as an output the following mesh model :

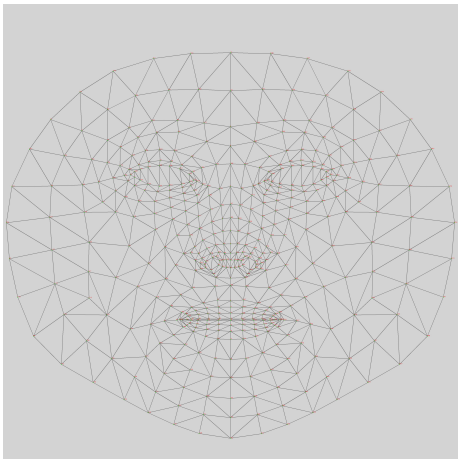

FIGURE 3 – Mesh model of the face with all the landmarks

FIGURE 4 – Mesh model applied on the detected face

I chose to use MediaPipe because it was already the model used by Matthias for his pointing gesture recognition algorithm. He gave me his initial code, so I could get familiar with how the module worked and get inspired for the gaze tracking pipeline. Moreover, MediaPipe, as opposed to other face recognition models, also provides iris tracking estimation. That was the most important decision point, because the face tracking on its own could only give me the head direction and not the gaze in itself. I used MediaPipe Iris [8], which is built on the MediaPipe Face mesh model, but adding other landmarks such as the iris, the pupil and the eye contours. The model also determines the metric distance between the model and the camera with a relative error of less than 10%, which is a feature that I didn't used for my project as it wasn't necessary.

Here is an example of the Iris detection model on the previous picture :

FIGURE 5 – Caption

3.3 Development environment

The first few weeks, I worked from my own laptop, and Peter gave me an RGBD camera (the intel RealSense Depth Camera D435) to test my algorithms and to get familiar with the different environments. To start, I used the MediaPipe module, to understand how the face detection worked. My first trial codes are in the mediapipe folder. I coded in Python, and at first I only used single frames and videos to understand how everything worked. Once that was done, I had to use Docker to work with the camera, because Sasha runs on ROS1 and not ROS2, which is the version I currently have on my laptop. Moreover, this setup gave me more freedom and flexibility for testing, and I could switch to another computer more easily if needed.

Once the algorithm gave me satisfying results, I started working directly with Sasha in more realistic conditions. In the office, there are two computers which are connected to the same local network as Sasha. By connecting to Sasha via SSH, I could control it and test my algorithm directly on Sasha's camera. It had already existing repositories with code from previous researchers and students, which I could adapt to my needs. For example, there were codes to put Sasha back to its neutral position, and to reset it. To make the switch easier between the different computers, I used GitHub to have access to all of my code synchronized and accessible all the time [9]. We were several people using Sasha, so we organized ourselves by using a shared Google Calendar to book time slots to say which of the two desktop computers we needed (named Robbie or Raufbold), and if we were going to use Sasha as well or not. This organization also helped me be more rigorous, especially in planning my work, because I had to make sure everything was ready on the right computer before my time slot with Sasha. It also made me stay really organized, as I often had to switch between computers.

Raufbold was more practical to use, because it had a USB-C port, so I could connect the RealSense camera as well as Sasha. Moreover, I had trouble running my algorithm on Sasha from Robbie, so most of the time I used Raufbold and only used Robbie when working with saved rosbags.

4 Implementation

4.1 Gaze detection algorithm

The RGBD sensor provides a pre-determined coordinate system, which is referred to as the main "camera frame" in which the depth cloud detected by the sensor is expressed. For my algorithm, I chose to create a frame in which the face would never move and would be the origin. I will refer to it as the "face frame" in this report.

To compute it, I had to choose 2 approximately perpendicular vectors that I could compute directly with the facial landmakrs detected by the MediaPipe module.

I chose these two vectors :

— the horizontal direction of the head defined as the vector between the right eye inner corner and the left eye inner corner.

— the vertical direction of the head is the vector going from the chin to the middle point between the inner corners of both eyes.

I normalized both of these vectors and translated them so that their origin would correspond to the nose bridge of the detected face. The results are shown on the following figure 6 as :

— the normalized horizontal vector is the x vector, in red

— the normalized vertical vector is the y vector, in green

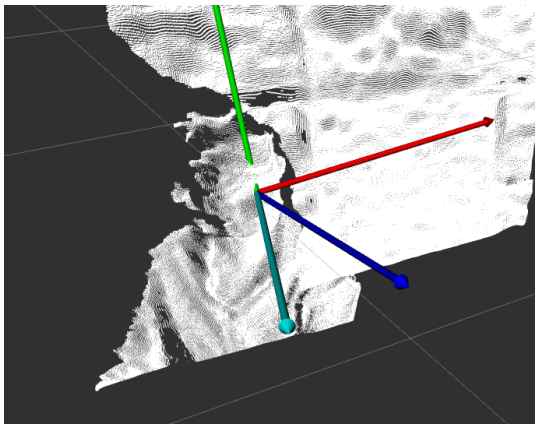

FIGURE 6 – The face frame ploted in `RVIZ` (The light blue vector is the computed gaze).

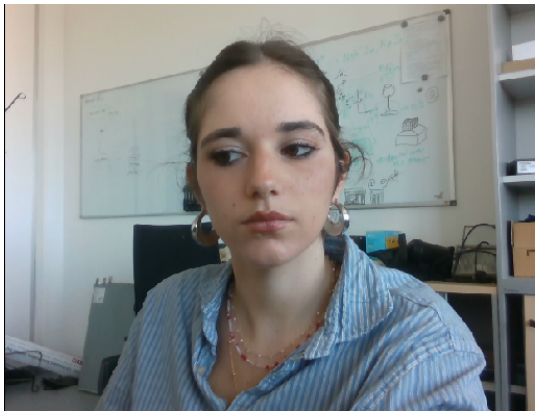

FIGURE 7 – The reference picture.

Then, I computed and normalized the cross product of these two vectors to find an estimation of the head direction in the camera frame. It is the z-vector, shown in blue in the figure.

All three of these vectors define the "face frame". Once that was done, I could then focus on gaze detection.

To detect the gaze direction, the last step was to compute the direction of the eyes relative to the neutral eye position (when the person is looking straight in front of them).

For example, the neutral position of the eyes, which is the middle point between the inner and outer corners of each eye, is shown in blue in the following figure, and the green markers are the real pupils of the detected face.

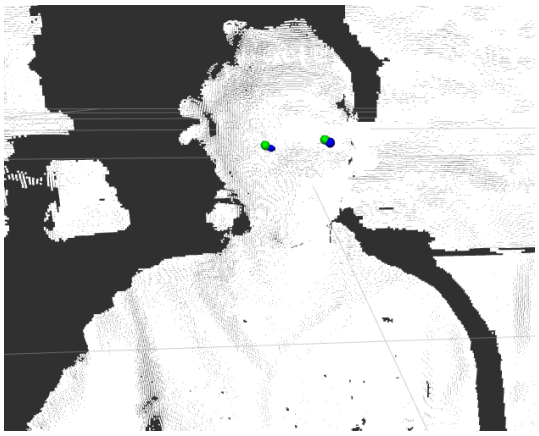

FIGURE 8 – The detected pupils (green) and the neutral eye position (blue).

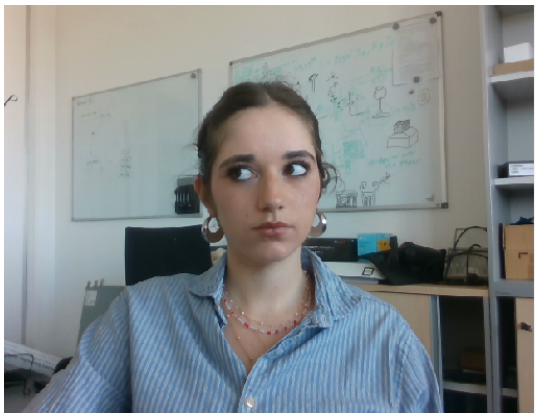

FIGURE 9 – The reference picture.

I computed the vector between the neutral pupil position and the detected pupil, and projected the result on the x and the y axis, to obtain the deviations dx and dy. This way, I could get the final gaze vector by adding the error detected to the neutral head direction, the z vector.

```
gaze_vector = x_vector * dx + y_vector * dy + z_vector
```

This gives an accurate estimation of the gaze direction expressed in the frame of the camera and in the face frame, as it combines both the overall head rotation and the eye movement relative to the face.

The expression of this vector in the camera frame was useful to plot the different markers in RVIZ and to be able to see them with the depth cloud. The one is the face frame was useful for the validation and testing part of the project, which I will explain in further details in another section.

4.2 Filtering the data

Once the gaze detection was working, I realized that the results were jittery and there was a lot of noise and occasional sudden errors. To improve the algorithm, I needed to filter the gaze. Peter, one of my supervisors, suggested I used the One Euro Filter. It is a low-pass filter, with an adaptative cutoff, which was developped by Géry Casiez in 2012 [10]. With a regular low-pass filter, you can either remove the noise but the gaze will react too slowly, creating lag issues, or you can filter less agressively, so the filter can stay really reactive, however id doesn't smooth out the noise and the result will stay jittery.

The advantage of the One Euro Filter is that the cutoff frequency adapts to the signal frequency, to filter more when the signal is stable to avoid jitter, and less when the signal varies significantly, to avoid lag.

The adaptative cutoff frequency is given by :

$$f_c = f_{c_min} + \beta * |d_s(t)|$$

where $d_s(t) = \frac{s(t)-s(t-dt)}{dt}$ is the derivative of the signal to filter $s(t)$.

To tune the filter, the two parameters that can be set are f_{cmin}, the minimum cutoff frequency, and β, with default values of 1 Hz and 0.

First, the filter is tested on a low speed movement detection, and f_{cmin} is adapted to remove jitter while keeping an acceptable lag (decreasing f_{cmin} amplifies the effect of the filter).

Then, it has to be tested on higher speed movement, this time adjusting the parameter β, to minimize lag as much as possible.

In my project, there were two elements to filter :

— the pupil movement (dx and dy) : it was very jittery, and it was interpreted as sudden gaze shifts even though it wasn't the case. It needed a very reactive filter, so I chose the One Euro Filter.

— the head direction vector (z vector) was smoother and more predictable than the pupils, so I applied a Kalman filter, which is more robust and worked better for this type of signal

After testing, I chose the following parameters for the One Euro Filter :

— $f_{cmin} = 5Hz$

— $\beta = 0.008$

4.3 Testing on Sasha

After developing and testing my algorithm with the RealSense Camera, the next step was to validate the algorithm and to test it in real conditions. To do so, I used the robot Sasha, which I presented in a previous part of this report. Other researchers working with Sasha had already developed algorithms that made it able to recognize QR codes. The position on the QR codes in the map frame could then be computed by analysing the depth frame if the camera.

The verification process was in two main parts:

— **Locating the QR codes :** Sasha had to rotate its head to scan its surroundings and detect the predefined list of QR codes I had given it. Once it had located the coordinates of each QR code in the map frame, it had to go back to its neutral position.

— **Gaze detection :** Sasha had to look up, at a chosen angle to ensure it could look at a person in front of it. Then, Sasha would instruct the person to look at the QR codes one by one, and the gaze detection would be applied during this part.

Then, I would compare the direction the person looked at and the "ground truth", i.e. the real QR code positions detected by Sasha, and possibly test different filters or techniques to see how the algorithm would work best.

4.3.1 Detecting the QR codes

Sasha already has basic modules developed. I was given the documentation to get familiar with how it worked, as well as code from a PHD student that was also working with Sasha during the time of my internship. First, Sasha has to rotate its head horizontally to scan its environment. Being unable to turn its head to 360° for mecanichal reasons, it has to turn its head from -220° to +100°, with a 20° step. As it turns, there is the `ar_marker` ROS topic running in the background that detects the QR codes and publishes their ID and position in the camera frame. Then, each detected marker is transformed from the camera frame to the map frame, using a tf listener. I had a list of markers I wanted to detect, that I had placed around the room at different distances from Sasha. Sasha would rotate its head until it had detected all the markers from the list. The detected markers positions where pusblished in a PoseArray to be able to visualize them in RVIZ. They were also savec in a .yaml file to be used later for the validation part of the gaze detection.

The following figure shows the detected markers in RVIZ in the map frame :

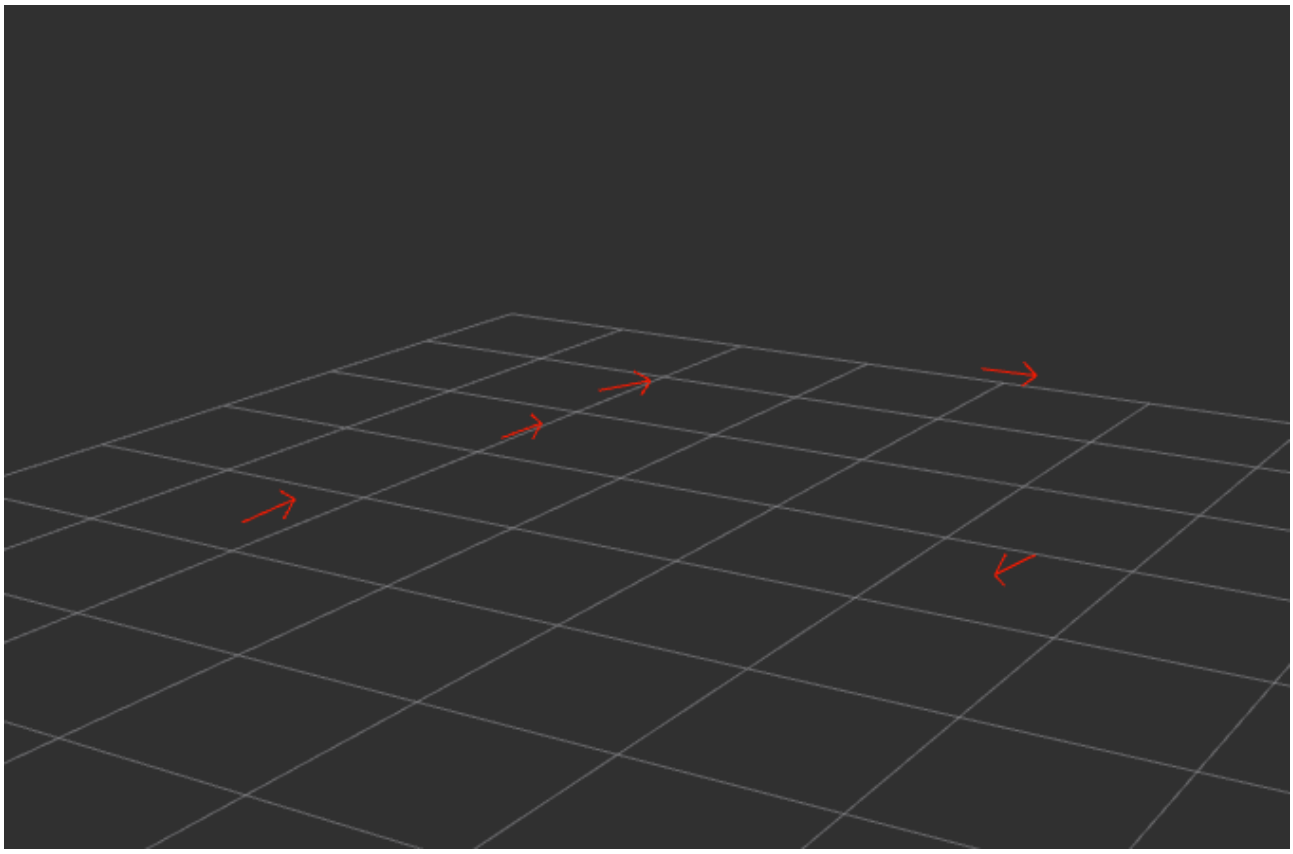

FIGURE 10 – Detected QR codes in RVIZ.

4.3.2 Gaze detection with Sasha

Once all the QR codes were detected, Sasha would go back to its neutral position, and then look up at a 20° angle, which was the best angle to be able to look at a person standing in front of it. Then, using the vocal commands module, Sasha would ask the person to look at each QR code one by one by saying the ID of the QR code. At the beginning, I wanted to apply the gaze detection algorithm directly during this part, however I wanted to record rosbags of all the data to be able to test different versions of the algorithm later. The rosbags would have been too heavy to record if I had my alogrithms running at the same time, so I decided to only record the camera data and the tf data.

4.3.3 Results

The QR code detection worked really well and I was able to visualize the detected markers in RVIZ. However, I had several issues with the gaze detection part.

My first issue was managing the tf frames. I couldn't record all the data for the rosbags or they would have been too heavy, so I had to make sure I was recording the right tf frames to be able to compute the gaze direction in the map frame later.

Once this issue was solved, I had trouble getting the gaze detection to work. The face wasn't detected by the MediaPipe module which made the whole algorithm fail. I tried to adjust the head tilt of Sasha to see if it would help, I also tried to get as close as possible to the camera. It didn't work, and it also wouldn't have been realistic to be that close to the robot in real life conditions.

FIGURE 11 – Camera view from Sasha : the face isn't detected.

I also tried adding more light to the room, by adding a lamp as well as placing my phone flashlight right above the camera to light up my face. None of it worked, the face was only detected for a few frames, so I couldn't get any satisfying results.

I also saved a few frames that I used to test the MediaPipe detection directly by using MediaPipe studio [11], which is a web interface where you can upload images to test the different models directly. The face wasn"t detected on most of the pictures, which confirmed that the problem came from the testing conditions and not from my code.

4.4 Testing with a dataset

To test my algorithm, one of my supervisors, Matthias, suggested I use the ManiGaze Dataset, which is a dataset developped by a group of researchers from the Institute Martigny in Switzerland [12]. They have recorded 4 different sessions :

— Markers on the table targets : The participant stood about 1m away from the robot, and followed its instructions to look at numbered markers, clicking a mouse when gazing at the target.

— End-effector targets : The participant looked continuously at one of the robots end-effectors, which was moving in different positions.

— Object manipulation : The participant had to move objects according to the robot's intructions.

— Set a table : The participant had to set a table while explaining their actions.

The dataset included RGB and depth videos, as well as the target positions, mouse events, and the robot end-effector positions. I used the first session, which was the most relevant to my project.

My original code was implemented directly in ROS1 nodes. To be able to use the dataset, I could either adapt the dataset to ROS1 or adapt my code to work independently from ROS. I chose the second option, as it would make my work easier to use in different context and experiment.

I rewrote my code to create an independant Python module, that would take as input only RGB and depth frames, and would output the gaze direction in the camera frame. I also added the filtering part directly in the module.

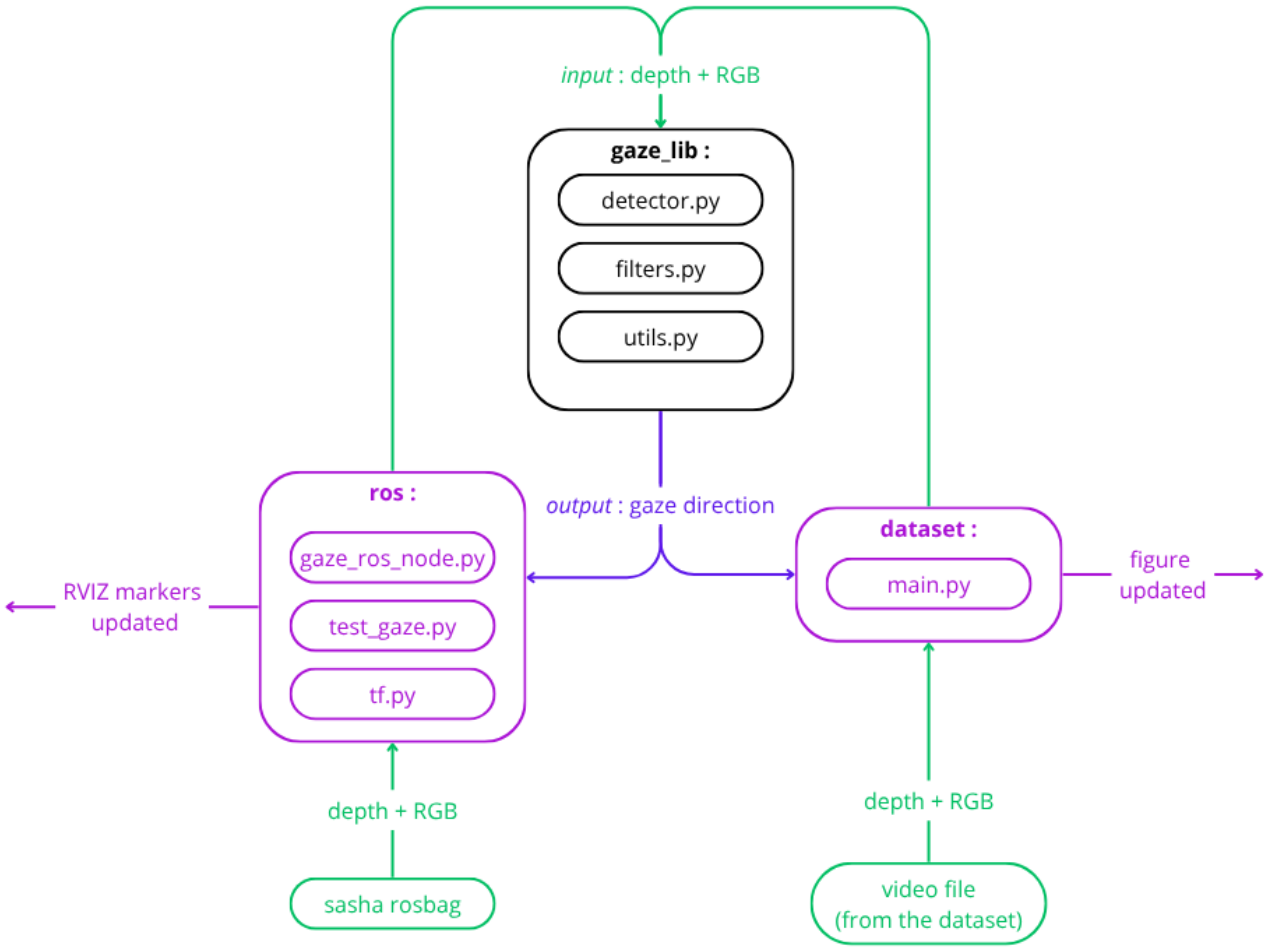

FIGURE 12 – New architecture of the gaze detection module.

After adapting my code, I tested it on the dataset. The results were once again not satisfying. The face wasn't detected on any frames, I tried different ones on MediaPipe studio and it didn't work there either. The video was not good quality enough, and the subject was a bit too far from the camera. Only the body landmarks were detected, which wasn't enough for my validation to work.

Despite not being able to validate my algorithm, this part of the project made me get familiar with Sasha and ROS1, as well as with working with datasets. It also forced me to reorganize my code and make it more modular, which is always more useful if the code is to be reused later for different applications.

I could have tried another model for the face detection, however I prefered to focus on the next part of the project, which was combining both gaze and pointing detection into a single module.

5 Combining gaze and pointing detection

5.1 Detection algorithm

As I couldn't validate my gaze detection algorithm, the next phase of my internship was to combine both gaze and pointing detection into a single module. The pointing detection algorithm worked in a similar way to the gaze detection one, using MediaPipe to detect the body landmarks. The pàoitning detection algorithm was already implemented by Matthias, so I could use it as a base to build my own module. It is applied on the right arm of the user. 33 landmarks are detected : the algorithm uses only three of them : the shoulder (12), the elbow (14) and the wrist (16).

— For each landmark, we find the corresponding pixel in the RGB image, and the corresponding depth value in the depth image.

— The algorithm then computes the 3D position of each landmark in the camera frame.

— If all three landmarks are detected, the algorithm computes the pointing direction as the vector from the shoulder to the wrist.

The goal of using both gaze and pointing detection is to be able to switch between the two methods depending on the distance of the user to the camera. If only the face is detected, the gaze detection will be used, and if the whole body is detected, the pointing detection will be used, or a fusion of both methods, which I didn't have time to implement.

The first step was to adapt the pointing detection algorithm to work with the gaze detection one. I used the same structure as the gaze detection module, so that both modules could be used independently or together.

The results are shown in the following figures :

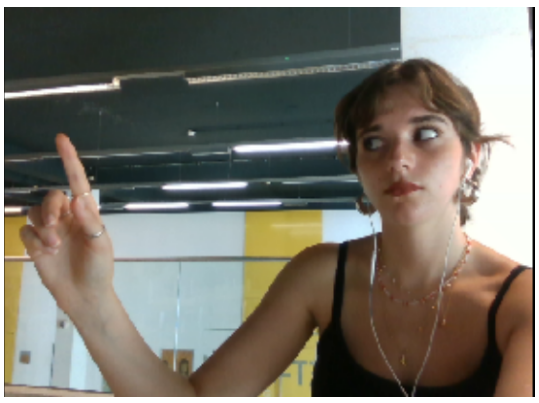

FIGURE 13 – The reference picture.

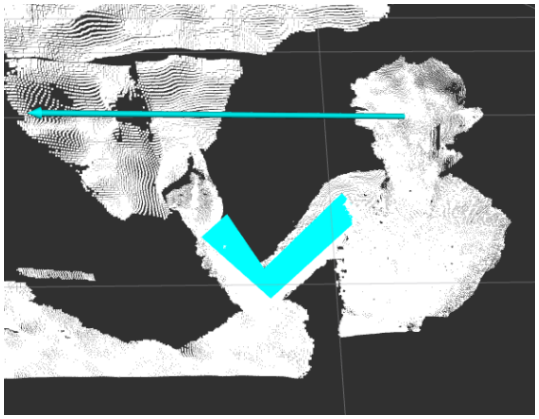

FIGURE 14 – The detected gaze and pointing vectors.

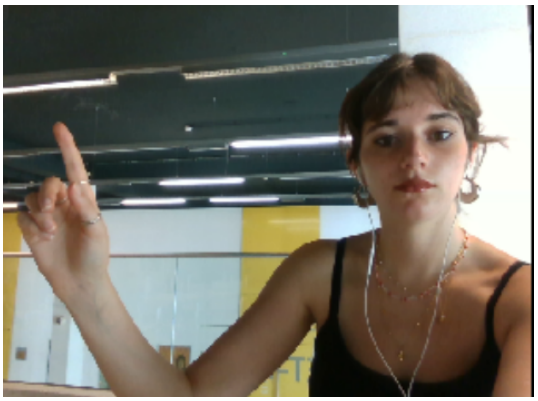

FIGURE 15 – The reference picture.

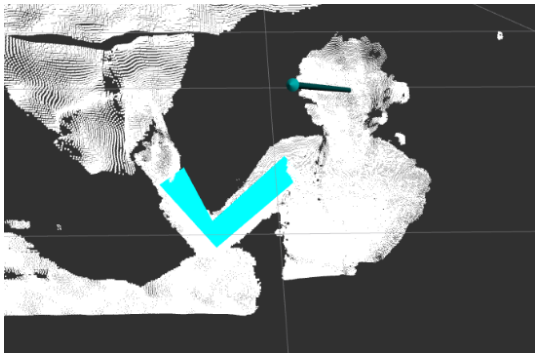

FIGURE 16 – The detected gaze and pointing vectors.

5.2 Final results

I tested the combined module with the RealSense camera. I wanted to analyse the consistency between both detection methods, as well as the effect of filtering on the gaze estimation. To do so, I performed series of test where I looked at different points in the room while pointing at them at the same time. I also moved my head to see how the gaze detection would react to different angles.

I calculated three different metrics to evaluate the results :

— **Raw-Filtered** The angle between the raw gaze vector and the filtered gaze vector in the horizontal plane (x-z plane of the face frame).

— **Raw-Pointing** The angle between the raw gaze vector and the pointing vector in the horizontal plane.

— **Filtered-Pointing** The angle between the filtered gaze vector and the pointing vector in the horizontal plane.

The results are shown in the following figures.

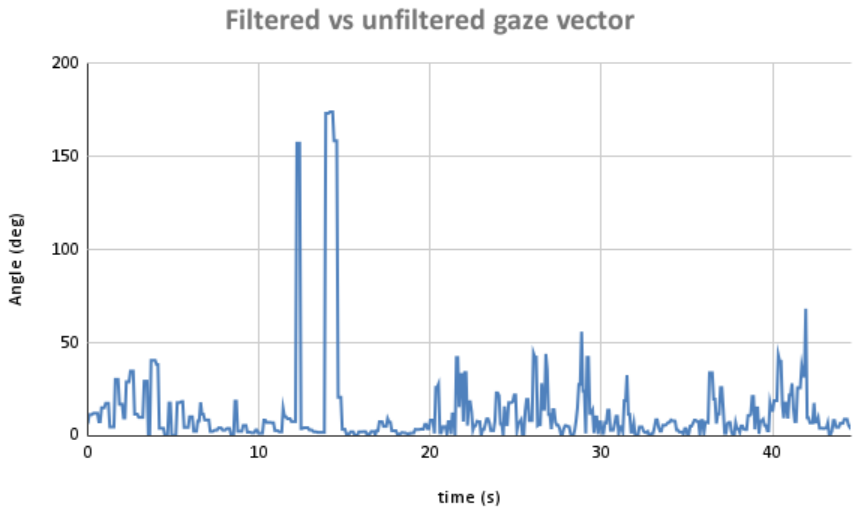

FIGURE 17 – Angle between the filtered and non-filtered gaze vectors.

TABLE 1 – Angular difference statistics between raw and filtered gaze vectors.

Angle	Mean (°)	Std (°)	Min (°)	Max (°)
Raw–Filtered	20.6	33.9	0.02	174.1

The filtered gaze vector remains relatively aligned with the raw gaze direction, with a mean angular difference of around 20°. It confirms that the filter smooths fast variations without altering the gaze orientation.

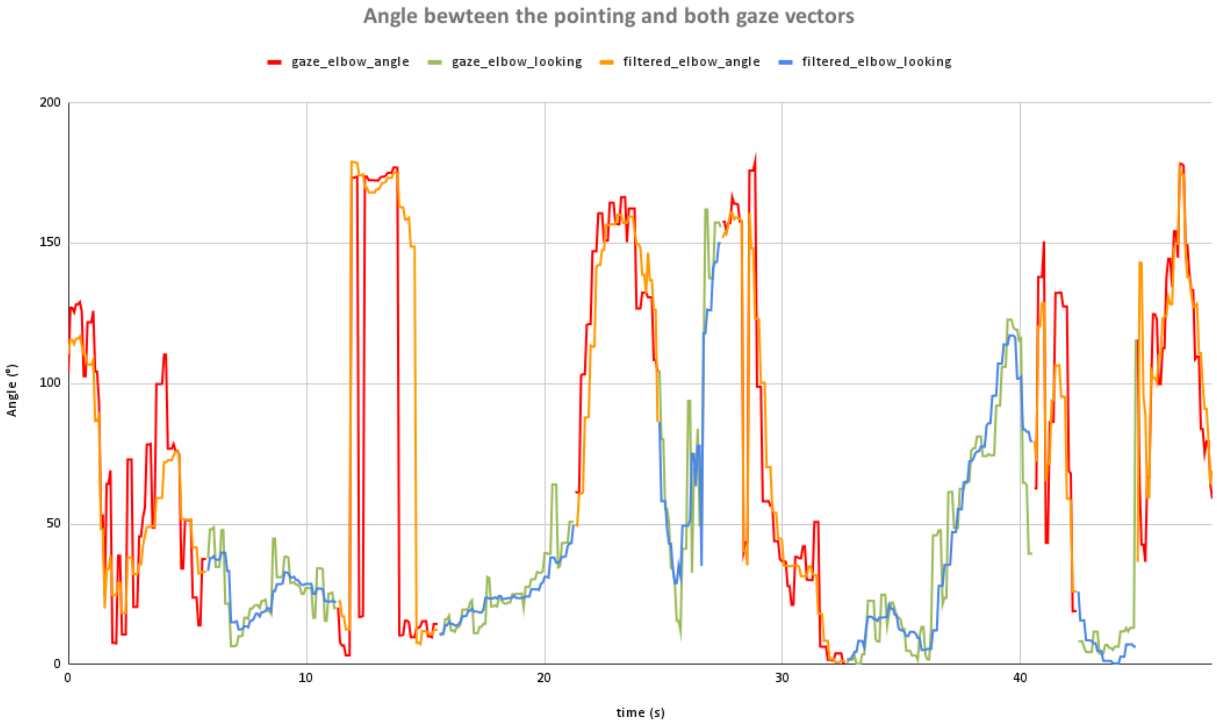

FIGURE 18 – Angle between the gaze vector and the pointing vector.

TABLE 2 – Angular difference statistics between the pointing and the gaze vectors.

Angle	Mean (°)	Std (°)	Min (°)	Max (°)
Raw-Pointing (when looking at target)	36.8	34.5	0.18	162
Filtered-Pointing (when looking at target)	35.4	32.5	0.36	150
Raw-Pointing (not looking at target)	86.7	58.1	0.5	179
Filtered-Pointing (not looking at target)	89.9	55.7	0.8	179

In contrast, the angles between the gaze and pointing directions are larger (around 36° on average), with higher variability. There can be several reasons for that :

— natural differences between head and arm alignment during pointing gestures,

— or residual noise in landmark detection, particularly when the subject is moving their head a lot.

Overall, the filtering process behaves as expected, and the system correctly differentiates between gaze and pointing movements. However, the fusion of both methods would require more work, especially taking into account the natural angle difference between the head and the arm, particularly when the user is standing close to the robot, which amplifies this effect.

6 Perspectives and future work

This internship allowed me to build the first version of a combined gaze and pointing detection module. Even though the results were not fully conclusive, the work opened several interesting directions for improvement and future development.

6.1 Improving robustness

The main limitation I faced was the lack of robustness of the face detection in real conditions. The MediaPipe model often failed when the lighting was not ideal or when the subject was a bit too far from the camera. Using a better camera setup, improving the calibration between the RGB and depth frames, or testing other models could make the gaze detection much more reliable. The filtering part could also be improved, for example by tuning the One Euro Filter differently depending on the distance or by combining it with another method.

6.2 Combining both detections

The next step would be to really combine the gaze and pointing detections instead of testing them separately. For instance, when the user is close to the robot, only the face is visible, so the gaze detection should be used, while for larger distances, the pointing direction is usually more reliable. It could also be interesting to fuse both results dynamically : for example by weighting them depending on the detection confidence or the movement smoothness.

6.3 Integration and testing in real conditions

Finally, once the detection part is more stable, the module could be integrated into a real human–robot interaction scenario. It could be tested on Matthias' robots, for example, to allow the robot to understand which object or area the person is referring to by looking or pointing, so that the robot could grab the object or navigate through its environment.

Overall, this project built a base that can be reused and improved for future work. With more robust detection and proper fusion between gaze and pointing, the system could eventually be used in real collaborative or assistive robotic contexts.

7 Conclusion

During this internship, I worked on developing and testing a gaze detection algorithm using RGB-D data, and on combining it with a pointing detection module for human–robot interaction. Even though I encountered several difficulties, especially with the robustness of the face detection and the testing conditions, I was able to design a complete pipeline (from data acquisition to filtering and visualization), and to adapt it for different use cases, both in ROS and in standalone Python.

This project helped me better understand how visual perception can be used to interpret human intentions, and how complex it can be to make algorithms work reliably in real-world conditions. I also learned a lot about filtering techniques, 3D geometry, and data synchronization between different frames of reference.

Beyond the technical aspects, I also discovered how to work within a research team, how to structure my work, and how to test and validate hypotheses step by step, which made me more rigorous and methodical.

Overall, even if the results were not as complete as I expected, this internship gave me a strong foundation and many ideas for future work on multimodal human–robot interaction.

Table of Figures

Annexes

A Detection module code

You can find the complete code of the entire detection modules in the GitHub repository: https://github.com/yasminerx/gaze_detection. The following sections include key parts of the gaze and arm detection code.

A gaze_lib/detector.py — GazeDetector

```
class GazeDetector:
def __init__(self, t0, camera_info, depth_encoding, depth_scale, model_complexity=1,
    static_image_mode=False):
# initialise the face detection model
self.mp_face_mesh = mp.solutions.face_mesh
self.depth_encoding = depth_encoding
self.depth_scale = depth_scale
self.camera_info = camera_info

# initalise the filters
self.kw = KalmanWrapper()
# min_cutoff : sensibilité aux changements rapides (haut = moins de lissage)
# beta : réactivité aux changements rapides (haut = plus de variations acceptées)
self.dx_filter = OneEuroFilter(t0, 0.0, min_cutoff=5, beta=0.008)
self.dy_filter = OneEuroFilter(t0, 0.0, min_cutoff=5, beta=0.008)

# Pose detection model complexity (0, 1, 2) can be set as rosparam
model_complexity = model_complexity
if model_complexity not in [0, 1, 2]:
model_complexity = 1
print(f"Model complexity = {model_complexity}")


# setup face detection model
self.face = self.mp_face_mesh.FaceMesh(static_image_mode=static_image_mode,
refine_landmarks=True,
max_num_faces=1)
# setup pose detection model
self.mp_pose = mp.solutions.pose
min_detection_confidence = 0.5  # can be set as rosparam
self.pose = self.mp_pose.Pose(static_image_mode=static_image_mode,
        min_detection_confidence=min_detection_confidence, model_complexity=model_complexity)
self.eye_detected = False

def get_eye_keypoints(self, face_results, rgb_image, depth_image):
''' Extract the x, y coords of both eyes from the detected face keypoints.'''
gaze_keypoints = {'right': None, 'left': None}
eye_detected = {'right': False, 'left': False}
image_height, image_width, _ = rgb_image.shape

#indices of the right and left pupil (in order)
pupil_indices = {"right": 468,
```

```
 "left": 473,
 "inner_right":362,
 "outer_right": 263,
 "inner_left":133,
 "outer_left": 33,
 "nose_bridge":168,
 "nose": 1,
 "chin": 200,
 "upper_right": 386,
 "lower_right": 374,
 "upper_left": 159,
 "lower_left": 145,}
if face_results.multi_face_landmarks:
for face_landmarks in face_results.multi_face_landmarks:
# get the x, y coordinates of the pupils
for side, i in pupil_indices.items():
landmark = face_landmarks.landmark[i]
x = int(landmark.x * image_width)
y = int(landmark.y * image_height)
gaze_keypoint = get_depth_coordinates(x, y, depth_image)
gaze_keypoints[side] = gaze_keypoint
eye_detected[side] = True
self.eye_detected = True
gaze_keypoints['right_eye_center'] = (np.array(gaze_keypoints["inner_right"])
            + np.array(gaze_keypoints['outer_right']))/2
gaze_keypoints['left_eye_center'] = (np.array(gaze_keypoints['inner_left'])
            + np.array(gaze_keypoints['outer_left']))/2

print("Face detected in the image.")

else:
print("No face detected in the image.")
return gaze_keypoints, eye_detected

def detect_eye_gaze(self, rgb_img, depth_img, t):
print("detect eye gaze started")

# is there a face in the imge ?
self.eye_detected = False

# check if the body is detected
try :
body_results = self.pose.process(rgb_img)
if body_results.pose_landmarks is not None:
print("Body detected in the image.")
else:
print("No body detected in the image.")
except Exception as e:
print(f"Error processing body landmarks: {e}")

# main face detection
```

```
face_results = self.face.process(rgb_img)
gaze_keypoints, eye_detected = self.get_eye_keypoints(face_results, rgb_img,
        depth_img)
if self.eye_detected :
# convert the pixel coordinates to camera coordinates
gaze_keypoints_cc = {}
for side, i in gaze_keypoints.items():
gaze_keypoints_cc[side] = pixel_to_camera_coordinates(i, self.camera_info)

try:
head_coordinate_system = self.update_head_coordinate_system(gaze_keypoints_cc)
dx, dy = get_eye_direction(gaze_keypoints_cc, head_coordinate_system)

print(f"dx: {dx}, dy: {dy}")

except Exception as e:
print(f"Error updating head coordinate system: {e}")
head_coordinate_system = None
dx, dy = 0.0, 0.0

print("returning values")
return gaze_keypoints_cc, eye_detected, head_coordinate_system, dx, dy
else:
return None, False, None, 0.0, 0.0
```

B gaze_lib/filters.py — filters and smoothing

```
class OneEuroFilter:
def __init__(self, t0, x0, dx0=0.0, min_cutoff=1.0, beta=0.0,
 d_cutoff=1.0):
"""Initialize the one euro filter."""
# The parameters.
self.min_cutoff = float(min_cutoff)
self.beta = float(beta)
self.d_cutoff = float(d_cutoff)
# Previous values.
self.x_prev = float(x0)
self.dx_prev = float(dx0)
self.t_prev = float(t0)

def __call__(self, t, x):
"""Compute the filtered signal."""
t_e = t - self.t_prev

# The filtered derivative of the signal.
a_d = smoothing_factor(t_e, self.d_cutoff)
dx = (x - self.x_prev) / t_e
dx_hat = exponential_smoothing(a_d, dx, self.dx_prev)

# The filtered signal.
cutoff = self.min_cutoff + self.beta * abs(dx_hat)
a = smoothing_factor(t_e, cutoff)
```

```
x_hat = exponential_smoothing(a, x, self.x_prev)

# Memorize the previous values.
self.x_prev = x_hat
self.dx_prev = dx_hat
self.t_prev = t

return x_hat


class KalmanWrapper:
def __init__(self, dt=0.1):
self.kf = KalmanFilter(dim_x=6, dim_z=3)
self.dt = dt
self.kf.F = np.array([[1, 0, 0, dt, 0, 0],
[0, 1, 0, 0, dt, 0],
[0, 0, 1, 0, 0, dt],
[0, 0, 0, 1, 0, 0],
[0, 0, 0, 0, 1, 0],
[0, 0, 0, 0, 0, 1]])
self.kf.H = np.array([[1, 0, 0, 0, 0, 0],
[0, 1, 0, 0, 0, 0],
[0, 0, 1, 0, 0, 0]])
self.kf.P *= 1000.0  # covariance matrix
self.kf.R *= 0.1
self.kf.Q *= 0.01

def update(self, z_measured):
self.kf.predict()
print("Kalman filter prediction step completed.")
self.kf.update(z_measured)
print("Kalman filter update step completed with measurement:", z_measured)
z_filtered = self.kf.x[:3].flatten()
if np.linalg.norm(z_filtered) == 0:
z_filtered = np.array([1.0, 0.0, 0.0])
print("Warning! Kalman filter output is zero, setting to default [1.0, 0.0, 0.0]")
else:
z_filtered /= np.linalg.norm(z_filtered)
return z_filtered
\end{lstlisting}

\subsection{gaze\_lib/utils.py --- helpers}
\begin{verbatim}
def get_depth_coordinates(x, y, depth_image, filtering=True, filter_width=3):
# Get the depth coordinate at desired x, y coordinate. Returns list of x,y,z
# Filtering the depth values in the neighborhood

# Unfiltered z-coordinate (depth) in meters
z = depth_image[y, x]

if filtering:
```

```
half = filter_width // 2

height, width = depth_image.shape

# Do not apply filtering if filter would be outside of image
if (x - half) < 0 or (x + half) > (width - 1):
filtering = False
elif (y - half) < 0 or (y + half) > (height - 1):
filtering = False

# Return the filtered depth value of the pixel neighborhood
if filtering:
z_list = depth_image[y - half : y + half, x - half : x + half]
z = np.median(z_list)

if z <= 0:
print("Invalid depth value at pixel coordinates ({}, {}): {}".format(x, y, z))

return [x, y, z]


def pixel_to_camera_coordinates(keypoint_pc, camera_info):
# Get intrinsic camera parameters from camera info and transform x,y,z from pixel
    coordinates to camera coordinates
fx = camera_info[0][0]
fy = camera_info[1][1]
cx = camera_info[0][2]
cy = camera_info[1][2]

# Transformation to camera coordinates
x_c = (keypoint_pc[0] - cx) * keypoint_pc[2] / fx
y_c = (keypoint_pc[1] - cy) * keypoint_pc[2] / fy

return [x_c, y_c, keypoint_pc[2]]


def get_eye_direction(keypoints_cc, head_coordinate_system):
right_eye_center = np.array(keypoints_cc["right_eye_center"])
left_eye_center = np.array(keypoints_cc["left_eye_center"])
right_eye_pupil = np.array(keypoints_cc["right"])
left_eye_pupil = np.array(keypoints_cc["left"])
upper_right = np.array(keypoints_cc["upper_right"])
upper_left = np.array(keypoints_cc["upper_left"])
lower_right = np.array(keypoints_cc["lower_right"])
lower_left = np.array(keypoints_cc["lower_left"])

# eye direction vector
# for the dx difference
x_vector = head_coordinate_system[0]
dx_right = (right_eye_center - right_eye_pupil)@ x_vector
dx_left = (left_eye_center - left_eye_pupil)@ x_vector
```

```
dx = 1 + dx_right/dx_left

# for the dy difference
y_vector = head_coordinate_system[1]
dy_right = (right_eye_center - right_eye_pupil)@ y_vector
dy_left = (left_eye_center - left_eye_pupil)@ y_vector
dy = (dy_right + dy_left)/2
return dx*34, dy*3
```

C arm_lib/detector.py — PointingDetector

```
class PointingDetector:
def __init__(self, t0, camera_info, depth_encoding, depth_scale, model_complexity=1,
    static_image_mode=False):
self.depth_encoding = depth_encoding
self.depth_scale = depth_scale
self.camera_info = camera_info

# Initialize mediapipe pose class
self.mp_pose = mp.solutions.pose

# Pose detection model complexity (0, 1, 2) can be set as rosparam
model_complexity = model_complexity
if model_complexity not in [0, 1, 2]:
model_complexity = 1
print(f"Model complexity = {model_complexity}")

# Setup pose detection model
self.pose = self.mp_pose.Pose(static_image_mode=static_image_mode,
        min_detection_confidence=0.9, model_complexity=model_complexity)

self.is_pointing = False

def get_arm_keypoints(self, pose_results, rgb_image, depth_image):
''' Extract the x, y coords of both the arm keypoints from the detected body landmarks.'''
arm_keypoints = {}
arm_detected = {}
image_height, image_width, _ = rgb_image.shape

arm_indices = {"right_shoulder": 12,
 "right_elbow": 14,
 "right_wrist": 16 }

if pose_results.pose_landmarks:
for arm_joint, i in arm_indices.items():
keypoint = pose_results.pose_landmarks.landmark[self.mp_pose.PoseLandmark(i).value]
x = int(keypoint.x * image_width)
y = int(keypoint.y * image_height)

arm_keypoint = get_depth_coordinates(x, y, depth_image)
arm_keypoints[arm_joint] = arm_keypoint
```

```
arm_detected[arm_joint] = True

if (arm_keypoints["right_shoulder"] is not None and
arm_keypoints["right_elbow"] is not None and
arm_keypoints["right_wrist"] is not None):
self.arm_detected = True
print("Right arm keypoints detected in the image.")
else:
self.arm_detected = False
print("No right arm detected in the image.")
return arm_keypoints, arm_detected

def detect_pointing_gesture(self, rgb_img, depth_img, t):
print("detect pointing gesture started")

self.arm_detected = False

# main arm detection
arm_results = self.pose.process(rgb_img)
arm_keypoints, arm_detected = self.get_arm_keypoints(arm_results, rgb_img, depth_img)

if self.arm_detected:
arm_keypoints_cc = {}
for side, i in arm_keypoints.items():
arm_keypoints_cc[side] = pixel_to_camera_coordinates(i, self.camera_info)

print("returning values")
return arm_keypoints_cc, arm_detected
else:
return None, False
```

D arm_lib/utils.py — arm helpers

```
def get_arm_angle(shoulder, elbow, wrist):
# Return the angle between vectors shoulder-elbow and elbow-wrist
# Angle can be used as a measure whether a person is pointing or not

shoulder = np.array(shoulder)
elbow = np.array(elbow)
wrist = np.array(wrist)

# Create the vectors
v_shoulder_elbow = elbow - shoulder
v_elbow_wrist = wrist - elbow

# Calculate the angle in degrees
angle_cos = np.dot(v_shoulder_elbow, v_elbow_wrist) / (np.linalg.norm(v_shoulder_elbow)
    * np.linalg.norm(v_elbow_wrist))
angle_cos = np.clip(angle_cos, -1.0, 1.0)
angle_degrees = 180.0 - np.degrees(np.arccos(angle_cos))

return angle_degrees
```

B ROS Nodes

A ros/gaze_ros_node.py — GazeRosNode (important parts)

```
class GazeRosNode :
    def __init__(self):
        self.camera_info = np.asarray(rospy.get_param('/pose_estimator/intrinsics'))
        self.depth_encoding = rospy.get_param('/pose_estimator/depth_encoding')
        self.depth_scale = rospy.get_param('/pose_estimator/depth_scale')
        self.frame_id = rospy.get_param('/pose_estimator/color_frame_id')

        # bridge and detector
        self.bridge = CvBridge()
        t0 = rospy.Time.now().to_sec()
        self.detector = GazeDetector(t0, self.camera_info, self.depth_encoding, self.depth_sca

        # service exposing gaze estimation
        self.service = rospy.Service("/gaze_callback", estimate_eye_position,
        self.gaze_callback)
        rospy.loginfo("Service /gaze_callback started")

        # several publishers for visualization (markers)
        self.pub_right_eye = rospy.Publisher("/pointing/right_eye", Marker, queue_size=10)
        self.pub_left_eye = rospy.Publisher("/pointing/left_eye", Marker, queue_size=10)
        self.pub_gaze = rospy.Publisher("/pointing/gaze", Marker, queue_size=10)

    def update_markers(self, dx, dy, head_coordinate_system, gaze_keypoints_cc):
        # compute non-filtered and filtered gaze vectors and publish as markers
        gaze_vector = head_coordinate_system[0]*dx + head_coordinate_system[1]*dy
        + head_coordinate_system[2]
        t0 = rospy.Time.now().to_sec()
        z_filtered = self.kw.update(head_coordinate_system[2])
        dx = self.dx_filter(t0, dx)
        dy = self.dy_filter(t0, dy)
        filtered_gaze_vector = head_coordinate_system[0]*dx + head_coordinate_system[1]*dy
        + z_filtered

        origin = gaze_keypoints_cc['nose_bridge']
        end_gaze = origin + gaze_vector * self.arrow_length
        point_origin = Point(origin[0], origin[1], origin[2])
        point_end_gaze = Point(end_gaze[0], end_gaze[1], end_gaze[2])
        self.gaze.points = [point_origin, point_end_gaze]
        self.gaze.header.stamp = rospy.Time.now()
        self.pub_gaze.publish(self.gaze)

    def gaze_callback(self, req):
        # convert ROS images to OpenCV and call detector
        rgb = req.rgb
        depth = req.depth
```

```
        depth.encoding = self.depth_encoding
        depth_img = CvBridge().imgmsg_to_cv2(depth, self.depth_encoding)
        depth_img = depth_img/int(self.depth_scale)
        rgb_img = self.bridge.imgmsg_to_cv2(rgb, "rgb8")

        detected_gaze_keypoints_cc, eye_detected, head_coordinate_system, dx, dy = self.detect
        depth_img, rospy.Time.now().to_sec())
        self.update_markers(dx, dy, head_coordinate_system, detected_gaze_keypoints_cc)
        # return response with 3D eye points
        res = estimate_eye_positionResponse()
        right_eye_point = Point(); set_point(right_eye_point, detected_gaze_keypoints_cc['righ
        left_eye_point = Point(); set_point(left_eye_point, detected_gaze_keypoints_cc['left']
        res.right_eye = right_eye_point
        res.left_eye = left_eye_point
        return res

if __name__ == "__main__":
    rospy.init_node('gazenodedetector')
     GazeRosNode()
     rospy.spin()
```

B ros/pointing_ros_node.py — PointingRosNode (important parts)

```
 class PointingRosNode :
     def __init__(self, arm_angle_thresh=120.0, arrow_length=2.0):
         self.camera_info = np.asarray(rospy.get_param('/pose_estimator/intrinsics'))
         self.depth_encoding = rospy.get_param('/pose_estimator/depth_encoding')
         self.depth_scale = rospy.get_param('/pose_estimator/depth_scale')
         self.frame_id = rospy.get_param('/pose_estimator/color_frame_id')
         self.arm_angle_thresh = arm_angle_thresh
         self.arrow_length = arrow_length
         self.bridge = CvBridge()
         t0 = rospy.Time.now().to_sec()
         self.detector = PointingDetector(t0, self.camera_info, self.depth_encoding, self.dept

         # service exposing pointing estimation
         self.service = rospy.Service("/pointing_callback", estimate_pointing_gesture, self.po

     def update_arm_markers(self, arm_keypoints_cc, arm_detected):
         # compute arm angle and publish arrows/lines when pointing
         if arm_detected['right_shoulder'] and arm_detected['right_elbow'] and arm_detected['r
             arm_angle = get_arm_angle(arm_keypoints_cc['right_shoulder'], arm_keypoints_cc['r
             if arm_angle >= self.arm_angle_thresh:
                 is_pointing = True
                 self.arrow_shoulder.action = Marker.ADD
                 self.arrow_elbow.action = Marker.ADD
             else:
                 is_pointing = False
                 self.arrow_shoulder.action = Marker.DELETE
                 self.arrow_elbow.action = Marker.DELETE

             # publish markers and return joint points in response
```

```
            res = estimate_pointing_gestureResponse()
            shoulder_point = Point(); set_point(shoulder_point, arm_keypoints_cc['right_shoul
            elbow_point = Point(); set_point(elbow_point, arm_keypoints_cc['right_elbow'])
            wrist_point = Point(); set_point(wrist_point, arm_keypoints_cc['right_wrist'])
            res.shoulder = shoulder_point
            res.elbow = elbow_point
            res.wrist = wrist_point
            return res
        else:
            return estimate_pointing_gestureResponse()

    def pointing_callback(self, req):
        rgb = req.rgb
        depth = req.depth
        depth.encoding = self.depth_encoding
        depth_img = CvBridge().imgmsg_to_cv2(depth, self.depth_encoding)
        depth_img = depth_img/int(self.depth_scale)
        rgb_img = self.bridge.imgmsg_to_cv2(rgb, "rgb8")
        arm_keypoints_cc, arm_detected = self.detector.detect_pointing_gesture(rgb_img, depth
        res = self.update_arm_markers(arm_keypoints_cc, arm_detected)
        return res

if __name__ == "__main__":
    rospy.init_node('pointing_node_detector')
    PointingRosNode()
    rospy.spin()
```

C ros/rosbag_analysis.py — PointingAngleCalculator (important parts)

```
class PointingAngleCalculator:
    def marker_to_vector(self, marker):
        # convert arrow marker points to a 3D vector
        if marker.points and len(marker.points) >= 2:
            p1, p2 = marker.points[0], marker.points[1]
            vec = np.array([p2.x - p1.x, p2.y - p1.y, p2.z - p1.z])
            return vec
        return None

    def angle_between(self, v1, v2):
        # compute angle (deg) between two vectors with safety checks
        if v1 is None or v2 is None:
            return None
        norm1 = np.linalg.norm(v1); norm2 = np.linalg.norm(v2)
        if norm1 < 1e-6 or norm2 < 1e-6:
            return None
        cos_theta = np.clip(np.dot(v1, v2) / (norm1 * norm2), -1.0, 1.0)
        return np.degrees(np.arccos(cos_theta))

    def compute_and_log(self):
        # compute angles between gaze, elbow and filtered vectors and save to Excel
        if self.gaze_vec is not None and self.elbow_vec is not None and self.filtered_vec
```

```
        is not None:
            angle_gaze_filtered = self.angle_between(self.gaze_vec, self.filtered_vec)
            angle_gaze_elbow = self.angle_between(self.gaze_vec, self.elbow_vec)
            angle_filtered_elbow = self.angle_between(self.filtered_vec, self.elbow_vec)
            self.results.append({
                "time": str(rospy.Time.now().to_sec()),
                "gaze_filtered_angle": angle_gaze_filtered,
                "gaze_elbow_angle": angle_gaze_elbow,
                "filtered_elbow_angle": angle_filtered_elbow
            })
            df = pd.DataFrame(self.results)
            df.to_excel("pointing_angles_01_09.xlsx", index=False)
```

D ros/tf.py — static TF broadcaster

```
def main():
    rospy.init_node('static_tf_map_to_camera')
    broadcaster = tf2_ros.StaticTransformBroadcaster()
    static_transform = geometry_msgs.msg.TransformStamped()
    static_transform.header.stamp = rospy.Time.now()
    static_transform.header.frame_id = "map"
    static_transform.child_frame_id = "head_rgbd_sensor_rgb_frame"
    # translation and rotation (example values used in the project)
    static_transform.transform.translation.x = 0.021
    static_transform.transform.translation.y = 0.016
    static_transform.transform.translation.z = 0.752
    static_transform.transform.rotation.x = 0.001
    static_transform.transform.rotation.y = -0.223
    static_transform.transform.rotation.z = 0.004
    static_transform.transform.rotation.w = 0.975
    broadcaster.sendTransform(static_transform)
    rospy.spin()

if __name__ == '__main__':
    main()
```

References

[1] V4R - Vision for Robotics Lab. https://www.acin.tuwien.ac.at/en/vision-for-robotics/. Accessed: 2025-07-29.

[2] MANiBOT. https://manibot-project.eu/. Pioneering Bimanual Mobile Robots for Challenging Work Environments, Accessed: 2025-07-29.

[3] Caring Robots - Home. https://www.caringrobots.eu/?lang=en. Accessed: 2025-07-29.

[4] SASHA - HSR Toyota Support Robot. https://www.acin.tuwien.ac.at/en/project/sasha/. Accessed: 2025-07-29.

[5] Yingzi Huo, Kai Jin, Jiahong Cai, Huixuan Xiong, and Jiacheng Pang. Vision transformer (vit)-based applications in image classification. In *2023 IEEE 9th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*, pages 135–140, 2023.

[6] v4r tuwien. Pointing Gesture Recognition. https://github.com/v4r-tuwien/pointing_gesture_recognition, 2025. Accessed: 2025-09-17.

[7] Google AI. Mediapipe solutions guide. https://ai.google.dev/edge/mediapipe/solutions/guide, 2025. Accessed: 2025-09-18.

[8] Google Research. Mediapipe iris: Real-time iris tracking & depth estimation. https://research.google/blog/mediapipe-iris-real-time-iris-tracking-depth-estimation/, 2020. Accessed: 2025-09-17.

[9] Yasmine Raoux.

[10] Géry Casiez. One euro filter explained. https://github.com/MKSharaf/OneEuroFilterExplained. Accessed: 2025-09-18.

[11] Google. Mediapipe studio: Face landmarker demo. https://mediapipe-studio.webapps.google.com/studio/demo/face_landmarker. Accessed: 2025-09-17.

[12] R. Siegfried, B. Aminian, and J.-M. Odobez. Manigaze: a dataset for evaluating remote gaze estimator in object manipulation situations. In *Symposium on Eye Tracking Research and Applications (ETRA)*, June 2020.