Paul Douat

Antonios Fritzelas

Development of Intelligent Systems

Report

University of Ljubljana, Faculty of Computer Science
2024

1- Introduction

This report aims to highlight and explain all the work that was carried out during this semester as part of the Intelligent System course, led by Professor Dr. Danijel Skocaj. Our solution takes advantage of the multitude of sensors provided by the Turtlebot4 robot, such as depth sensor, LIDAR (pointcloud), camera and odometer to accurately map and interact with the environment. It includes autonomous operation and path finding, image/feature recognition, SLAM navigation, positional tracking for fine maneuvering, 3D object, color and depth detection. In addition, we have used methods to detect faces and interact with them using natural language, and reasoning to process information in an intelligent manner.

2- Methods

Built-In Nodes

I) Map Goal

This method aims to enable the robot's movement using an OccupancyGrid map. Specifically, it provides a graphical interface via OpenCV to visualize the map and allows the user to click on specific points on this map to set navigation goals for the robot. When a user clicks on a point on the displayed map, the coordinates of this click are converted into real-world coordinates.

These coordinates are then used to generate a navigation goal message, which is sent to the NavigateToPose action server of ROS 2. The ROS 2 node created by this code subscribes to a topic containing map data (e.g., /map) and uses an action client to communicate with the navigation server. The received map is converted into a numpy image, suitable for easy visualization with OpenCV. The map values are adjusted to match a standard visual representation, where cells occupied by obstacles are marked differently from free cells. When a new goal is set by a user click, the ROS 2 node sends this goal to the navigation server. If the goal is accepted, the robot starts moving towards the specified position. The code also handles feedback from the navigation server, indicating whether the robot has reached its goal or if the goal has failed. This allows the user to intuitively and interactively set navigation goals for the robot using a visual map and mouse clicks.

In summary, this code provides a convenient user interface for defining navigation trajectories by simply clicking on a map. It transforms click interactions into precise navigation commands, leveraging ROS 2's capabilities to manage communication and control navigation actions. This greatly simplifies the autonomous navigation process for robots, making it easier for users to interact with the system and set movement goals. **By using this function, we were able to extract the points necessary to complete the traversal of the map.**

II) ArmMoverAction

The code implements a ROS2 node in Python named `ArmMoverAction`, designed to control the movements of a robotic arm based on commands received on a ROS topic. It uses an action client to send trajectory commands to the arm controller via the `FollowJointTrajectory` action. The predefined arm positions include specific configurations for tasks such as searching for parking (`look_for_parking1`, `look_for_parking2`, `look_for_parking3`, `look_for_parking4`) and other positions like `garage` and `look_for_monalisa`.

The node initializes subscriptions to command topics, waits for the action server to be available, and then sends the appropriate positioning commands. It checks if the commands are accepted and waits for the execution results. A periodic timer checks if new commands have arrived and executes the corresponding movements. In the case of manual commands, the node evaluates and applies the specified joint positions. **Using this mechanism we achieve precise management of the robotic arm's movements for various tasks, including those specific to parking**

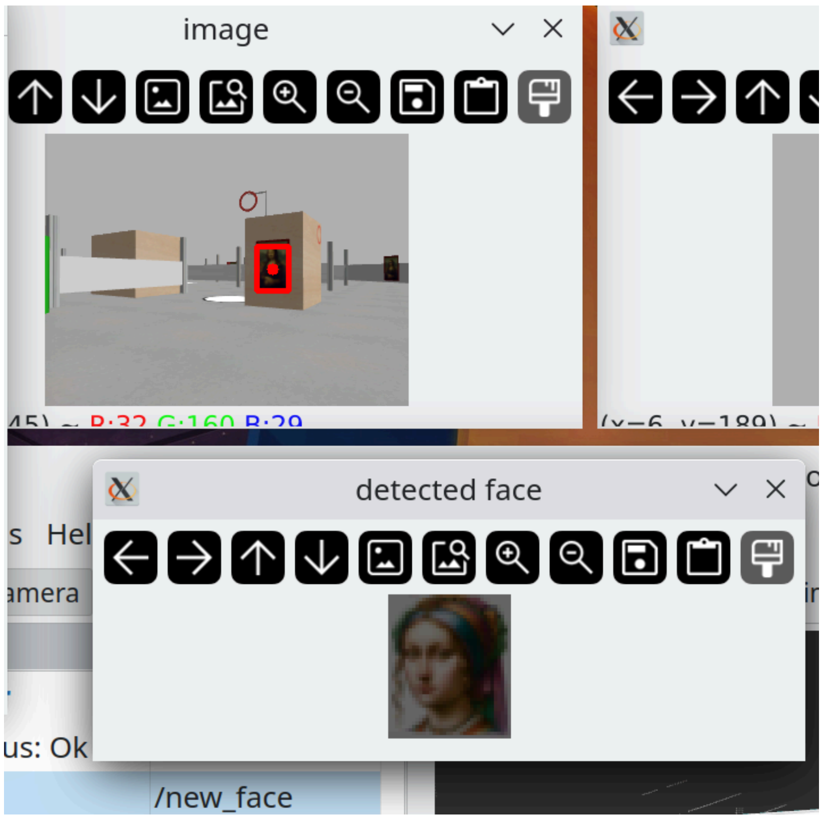
Modified Nodes

I) Detect Faces Node

The face detection method we have implemented aims to detect different faces present in paintings and artworks scattered throughout the map where the robot operates. To achieve this task, we used various Python libraries, including `sensor_msgs` and `std_msgs` to manipulate images, `PointCloud2` to use point clouds, `Marker` for visualization markers, and `Odometry` for odometry. This node also runs supplementary functions, switching between modes by using boolean variables set through callback functions of subscribed topics.

First, the image is converted from ROS format to OpenCV format using `CvBridge`. From this image, we use the YOLO (You Only Look Once) model to detect faces in an image, which returns bounding boxes around detected faces. For each detected face, a region of interest (ROI) is defined from the bounding box coordinates.

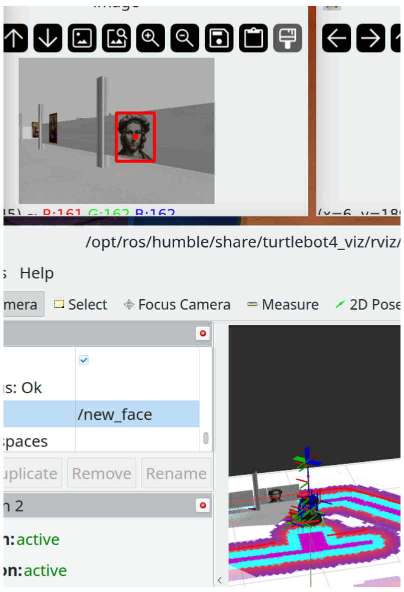
We then apply the SIFT (Scale-Invariant Feature Transform) algorithm to extract key points and descriptors of the faces. This allows us to create a list of all detected faces, thus avoiding detecting the same face twice. If the face is new, it is recorded, and a marker is displayed. **We differentiate paintings from people based on the number of white pixels** in the image, through a set threshold of over 100 pixels for faces.



Picture 1: Example of face detection and differentiation from painting

The 3D coordinates of the points corresponding to the detected face are then extracted from the point cloud and published to the robot commander node. The coordinates of the points, initially in the camera's frame, are transformed to be applied to the map's frame using `tf2_ros`.

This transformation is essential for obtaining accurate positions of the faces in the robot's global environment, in order to be able to walk up to them.



Picture 2: Approaching face through PointCloud coordinates

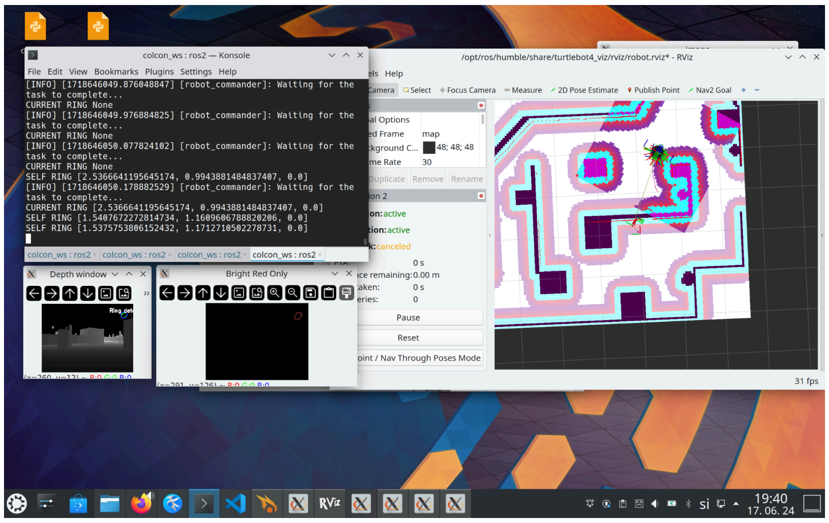
Face detection is only part of the functionalities of this ROS node. Additionally, the robot can read QR codes, download and verify reference images (such as the Mona Lisa), and perform parking maneuvers by analyzing images and calculating the necessary motion vectors.

Here are the main steps of the method:

1. Conversion of the image from ROS format to OpenCV.
2. Using YOLO to detect faces and generate bounding boxes.
3. Defining regions of interest (ROI) for each detected face.
4. Applying the SIFT algorithm to extract key points and descriptors of the faces.
5. Checking the uniqueness of the detected faces and recording new faces.
6. Publishing markers for newly detected faces.
7. Extracting the 3D coordinates of the faces from the point cloud.
8. Transforming the point coordinates from the camera frame to the map frame using `tf2_ros`.

II) Robot Commander

The methods implemented within the Python file `robot_commander` greatly facilitate the robot's navigation and actions, including functions for navigating to specific positions, rotating, docking and undocking, task management, and coordinate transformation. The `goToPose` method sends a command to the robot to move to a specific position using the `NavigateToPose` action, while `spin` allows the robot to rotate in place. This method is critical to move our robot in the coordinates of interest. Task management is handled by methods such as `isTaskComplete` and `cancelTask`, and coordinate transformation is performed via transformCoordinates using tf2_ros. We utilize the functions mentioned above to cancel existing goals and set new ones, as required by our navigation.



```
[INFO] [1718646049.876048847] [robot_commander]: Waiting for the
task to complete...
CURRENT RING None
[INFO] [1718646049.976884825] [robot_commander]: Waiting for the
task to complete...
CURRENT RING None
[INFO] [1718646050.077824102] [robot_commander]: Waiting for the
task to complete...
CURRENT RING None
SELF RING [2.5366641195645174, 0.9943881484837407, 0.0]
[INFO] [1718646050.178882529] [robot_commander]: Waiting for the
task to complete...
CURRENT RING [2.5366641195645174, 0.9943881484837407, 0.0]
SELF RING [1.5407672272814734, 1.1609606788820206, 0.0]
SELF RING [1.5375753806152432, 1.1712710502278731, 0.0]
```

Picture 3: Using navigation goals and viewing robot trajectory in RViz

The main function initializes the ROS2 system, waits for the navigation system to be ready, checks the docking status, and navigates the robot to a series of predefined points while searching for faces and colored rings. If new faces are found, their coordinates are appended to a list to visit, and are prioritized by the navigation (any current goals are canceled to first empty this list. Then, navigation proceeds where it had left off).

During navigation, the robot uses faceRecognizedCallback to receive a face location and newFaceCallback to know if it should go up to it, both from the detect_people node, and ring_callback to identify the position of the detected ring of a specified color. The script also uses speech recognition with speech_recognition to allow the robot to recognize voice commands using recognize_speech_from_mic and respond with speech synthesis via speak_text, enabling dynamic interaction with users. This combination of voice recognition, speech synthesis, and navigation and detection capabilities allows the robot to receive voice instructions to determine which ring to navigate to, thus effectively responding to one of the main tasks of the project. The way we achieve this is simple. Since only two persons have information regarding the rings, we query them as to which two rings do they think we should park under. We receive two possible answers from each. For each pair of colors the truth table is:

P	Q	$P \wedge Q$
T	T	T
T	F	F
F	T	F
F	F	F

Figuring out which one is the correct one is quite non-trivial. We append their answers to a list, and the duplicate answer is the correct ring.

Once found, it publishes the correct color to the detect_rings node, to switch from face to ring detection. The navigator also notifies the detector it should switch into parking mode and use the correct twist commands, and once complete check for anomalies.

III) Ring Detector

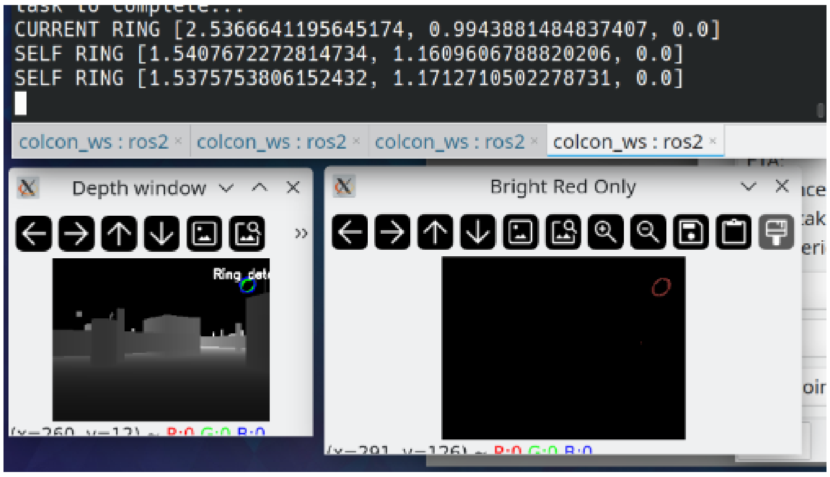
We have developed a method to detect rings and their position using various Python libraries, including `cv_bridge` for converting ROS images to OpenCV format, as well as numpy and OpenCV for image processing and `tf2_ros` for coordinate transformation. First, we convert the ROS images obtained from the depth and RGB camera to OpenCV format using the `cv_bridge` library. This allows us to process the images using the powerful features of the OpenCV library.

To verify if the object in question is indeed a ring, we have implemented a precise multi-factor detection method.

First, the `image_callback` method processes RGB images from the camera to detect rings of the specified color. We convert the image to HSV (Hue, Saturation, Value) format, which allows easier segmentation of specific colors. Then, we apply color masks to isolate the desired color (red, green, blue, etc.). The desired color is sent by the publisher in the robot commander node. Rings of the specified color are then detected by pattern search (color-black-color), and the coordinates (column) of the corresponding pixels are extracted to be compared to the depth sensor (in order to verify the coloured objected is a ring and not eg. a hollow square.

Concurrently, we are running depth image processing, where each pixel in the image contains information about the distance between the sensor and the observed object. We apply a binary threshold to isolate objects in the depth image and check the circularity of objects to detect rings. One challenge we faced was segmenting the pole of the ring from the image. We were able to overcome this by splicing the image into columns and creating merged images each time with one of the columns removed. This ensured that in at least one of the images, the pole is removed. Then, the pole is added back in and we save the column where the ring was detected. If the position of the ring (column) in the depth image matches the one found in the RGB image, then we have verified it is the correct ring.

Once a ring is detected, the method returns the average column and the coordinates of the detected ring's pixels, and paints it into a visualization.



```
CURRENT RING [2.5366641195645174, 0.9943881484837407, 0.0]
SELF RING [1.5407672272814734, 1.1609606788820206, 0.0]
SELF RING [1.5375753806152432, 1.1712710502278731, 0.0]
```

Picture 4: Detected ring, visualized

The `pointcloud_callback` method processes point clouds to obtain the global 3D coordinates of the ring. We extract the 3D coordinates of the pixels representing the ring, create visualization markers to represent the detected rings in the global space, and if a ring is detected, the markers are published for visualization, and the coordinates are sent through the topic "/ring_pub".

Then, the `transformCoordinates` method transforms the ring's coordinates obtained from Pointcloud to the map frame. This uses the reference frame transformations provided by `tf2_ros` to obtain the ring's global position in the environment.

3- Implementation and integration

Integration of the code occurs by correctly utilizing the subscribers - publishers model to correctly communicate between ROS 2 nodes. For the implementation, we have correctly combined all functionality in the simulation, and will be presenting the first task on the real robot.

I) Integration of components

The interoperability between navigation and detections requires that we store states in memory (eg. next navigation point to continue to after speaking to a face). Then, regarding face, ring and parking detection we use flags published to different ROS2 topics to differentiate the robot operation mode.

At startup, the Detect Faces node is configured to search for faces, and omit any Mona Lisa paintings in its path through the intelligent scanning described in the Methods section of this report. Then, when the robot has acquired enough information from speaking to faces (the correct ring color), the Robot Commander node publishes the color to detect. The ring detect node is configured not to detect rings if this variable has not yet been set. In order to successfully detect a ring, both the depth sensor and the RGB camera inference must agree to the existence of the ring.

Then, the coordinates of the ring, obtained through PointCloud, are pushed to the robot commander in order for the robot to move to its location. Once moved to its location, we stop looking for rings, and publish to the "/start_parking" topic that we are looking for parking.

The detect_people node automatically configures the correct position of the arm, if not set previously, and then proceeds to look for parking. The arm image is then analyzed to correctly park inside of the circle, by publishing twist commands to Robot Commander node aiming to achieve this goal

II) Execution - Implementation

We first launch the simulation by running the command:

```
ros2 launch dis_tutorial7 sim_turtlebot_nav.launch.py
```

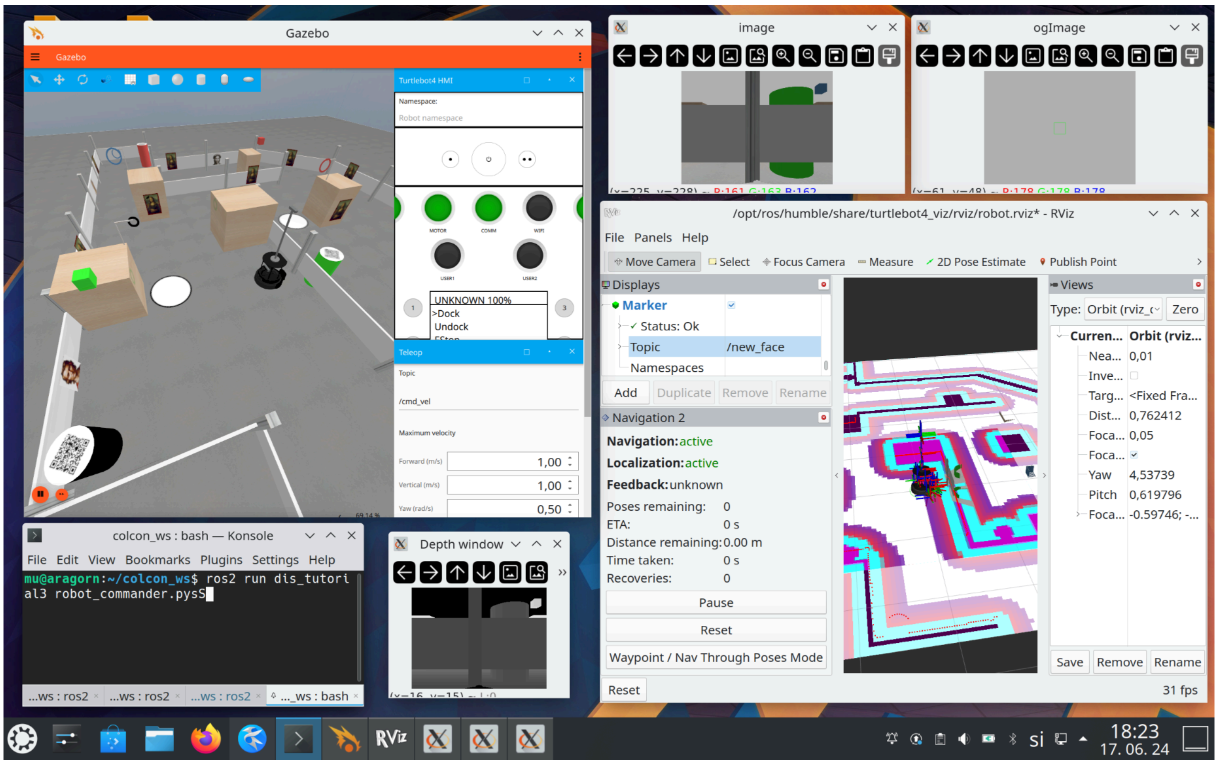
Then we run the three nodes, one for detecting faces, the other for parking and ring detection, and lastly for controlling the robot's navigation:

```
ros2 run dis_tutorial6 detect_rings.py
ros2 launch dis_tutorial3 detect_people.py
ros2 run dis_tutorial3 robot_commander.py
```

Correctly setting up the environment to capture log data (eg. using rosbag) and visualizing markers in RViz is also part of our implementation setup. From here we can also view the RGB camera and Depth sensor data.

Afterwards, we can set the position of the arm to be ready for parking by running the provided ArmMoverAction node and publishing to it's topic.

Now we can run all of our nodes together:



```
mu@aragorn:~/colcon_ws$ ros2 run dis_tutori
al3 robot_commander.pysS
```

Displays

Marker	✓
Status: Ok	
Topic	/new_face
Namespaces	

Navigation 2

Navigation: active
Localization: active
Feedback: unknown

Poses remaining:	0
ETA:	0 s
Distance remaining:	0.00 m
Time taken:	0 s
Recoveries:	0

Views

Type: Orbit (rviz_c

Curren...	Orbit (rviz...
Nea...	0,01
Inve...	☐
Targ...	<Fixed Fra...
Dist...	0,762412
Foca...	0,05
Foca...	✓
Yaw	4,53739
Pitch	0,619796
Foca...	-0.59746; -...

Teleop: Forward (m/s) 1,00; Vertical (m/s) 1,00; Yaw (rad/s) 0,50

Gazebo
Turtlebot4 HMI
Teleop
image
ogImage
Depth window
/opt/ros/humble/share/turtlebot4_viz/rviz/robot.rviz* - RViz
File Panels Help
Move Camera Select Focus Camera Measure 2D Pose Estimate Publish Point
Add Duplicate Remove Rename
Pause
Reset
Waypoint / Nav Through Poses Mode
Save Remove Rename
31 fps
colcon_ws : bash — Konsole
File Edit View Bookmarks Plugins Settings Help
...ws : ros2 ...ws : ros2 ...ws : ros2 ..._ws : bash
18:23
17. 06. 24

Picture 5: Workflow of execution

4- Results

In this section, we will discuss the different results we obtained for the various tasks assigned to us.

1) 1st Task

   The robot is able to autonomously navigate to set points in the virtual world. SLAM navigation is performed. The face recognition algorithm works as expected, it detects the correct faces, differentiating them from paintings and can isolate them in the images, and approach. It detects the exact number of faces required. In general, it is capable of providing us with images and 3D information.

2) 2nd Task

The algorithm is able to detect all the rings in 3D space. Thanks to our color isolation method, it is also able to identify the color of the detected ring. Therefore, it can detect the green ring as required and can park underneath. Fine maneuvering is possible for calculating the parking space, with a few issues sometimes related to the circle's boundary (sometimes exceeding it).

3) 3rd Task

The robot is capable of moving towards each of the detected faces and asking them in natural language where it can find the Mona Lisa. Then, thanks to speech recognition, the robot is able to isolate the 2 colors given to it. It will then go to a second character who will also give it 2 colors. One of the colors in each response is the same. It is then capable of finding this color, moving towards the ring with that color, and parking. The robot is also capable of reading a QR code using a reader and retrieving all the information from it, and also contains unfinalized methods for anomaly detection.

5- Division of Work

Concerning the division of the work:

- Stefan Andjekovic - 40% of the work (parking, approaching, half of navigation, part of ring detection, part of face detect, ~~anomaly detection~~)
- Antonios Fritzelas - 35% of the work (other half of navigation, most of ring detection, half of speech synthesis, part of face detect, face-painting differentiation)
- Paul Douat - 25% of the work (other half of speech synthesis, part of ring detection, part of face detect)

6- Conclusion

Throughout this semester, our work on the Development of Intelligent Systems course has led to the successful development and implementation of a robust solution utilizing the ROS2 platform. This project harnesses the diverse sensor array of the Turtlebot4, including depth sensors, LIDAR, cameras, and odometers, to enable comprehensive environmental interaction and autonomous navigation.

Key accomplishments of our project include:

1. **Navigation and Path Finding**: We developed code to correctly plot the map of the virtual world. This method enabled precise movement and pathfinding capabilities, essential for autonomous operation.

2. **Face Detection and Interaction**: Implementing YOLO and SIFT algorithms for face detection, we achieved accurate recognition of faces in paintings and artworks. The system was capable of extracting 3D coordinates of detected faces, enabling the robot to navigate to these points and interact intelligently.
3. **Ring Detection and Parking**: By processing RGB and depth images, we developed a reliable method to detect and identify rings of specific colors in 3D Space. This allowed the robot to perform the complex task of parking under the correct ring, demonstrating effective integration of vision and navigation systems.
4. **Voice Interaction**: Enhancing user interaction, the robot employed speech recognition and synthesis to follow voice commands and respond appropriately.

The integration of these capabilities was achieved through the ROS 2 framework, ensuring efficient communication between nodes and seamless operation. The system's ability to dynamically switch between different detection modes and manage tasks was key in addressing the various challenges posed by the project.

Overall, the project successfully met its objectives, showcasing the potential of intelligent robotic systems in complex environments. The results from the assigned tasks highlight the robustness and effectiveness of our solution, though some minor improvements, such as precise parking alignment, and the integration of the anomaly detection/QR code reading could be added.

Our approach provided us with valuable hands-on experience in autonomous systems and robotics, combining key technologies and areas of computer science interest into one project.