University of Tartu
Faculty of Science and Technology
Institute of Technology

Steven Tsienza

Camera-Based Machine Vision for End-effector Guidance using ROS

Practical Experiences in Computer Engineering · LOTI.05.043

Supervisor:
Karl Kruusamäe

Tartu 2023

1. Introduction

This report presents my main work for the Practical Experiences in Computer Engineering (LOTI.05.043) [1] course at IMS-Lab. This course is intended to enable students to gain professional work experience in a company or a university research laboratory.
As part of this course, my main task was to find, integrate and assess different camera-based machine vision ROS (Robot Operating System) packages to guide a manipulator end-effector during a pick-and-place task.

This work delves into the functionalities and usage of three crucial ROS (Robot Operating System) packages: **ar_track_alvar** [2], **vision_opencv** [3], and **find_object_2d** [4]. These packages are instrumental in enabling camera-based machine vision for end-effector guidance in robotic applications. They provide a range of capabilities to detect, track, and identify visual markers or objects, essential for object manipulation, navigation, and pose estimation.

Code and documentation are available on GitHub.

2. Problem statement

ROS has become a popular framework for robotics applications, including camera-based machine vision for guiding manipulator end-effectors (eefs) during pick-and-place tasks [5]. ROS is a flexible and scalable platform for developing and deploying software for robots.

However, various ROS camera-based machine vision approaches can be used for guiding a manipulator end-effector during a pick-and-place task, each with its advantages and limitations. The problem is that it can be challenging to determine which approach is best suited for a particular application or task, given the complexity and variability of the environment.

Therefore, this study aims to assess different ROS camera-based machine vision approaches for guiding a manipulator end-effector during a pick-and-place task. This study will evaluate the performance of various ROS packages, for different machine vision solutions.

By assessing these different ROS camera-based machine vision approaches, this study aims to provide a better understanding of the strengths and weaknesses of each approach and to identify the most effective method for guiding a manipulator end-effector during a pick-and-place task within the ROS framework. This can improve machine vision solutions in manufacturing and industrial environments, increasing productivity and decreasing costs.

3. Software packages

3.1. ar_track_alvar

The *ar_track_alvar* package is designed to detect and track Augmented Reality (AR) markers in camera images. These markers are physical patterns that a camera can recognise and track to provide information about their relative pose in the camera frame [2].

Figure 1: Example image of an alvar AR tag marker

The ar_track_alvar package subscribes to a ROS topic for camera images, typically named /camera/image_raw. It processes the input images to detect and identify AR markers. The package also requires camera calibration parameters, which are often provided through ROS parameters or YAML files.

Once a marker is detected, the package publishes the pose (position and orientation) of detected AR markers as ROS messages. The pose information is usually provided through the "/ar_pose_marker" topic. The message type for the detected AR markers is "ar_track_alvar_msgs/AlvarMarkers".

Configuring the package with appropriate parameters for marker sizes, camera calibration, and other settings is essential to ensure accurate pose estimation.

3.2. vision_opencv

The *vision_opencv* package integrates the OpenCV computer vision library with ROS to provide various image processing and computer vision functionalities [3].

vision_opencv provides the *cv_bridge* package that acts as a bridge between ROS image messages that have the message type "sensor_msgs/Image" and OpenCV image formats. The *cv_bridge* package helps convert images between these formats, enabling the use of OpenCV functions on ROS image messages.

3.3. find_object_2d

The *find_object_2d* package focuses on detecting and recognizing 2D objects in images using a bag-of-words approach [4], which is particularly useful for identifying objects with distinctive visual patterns.

find_object_2d package subscribes to an image topic, often named “/camera/image_raw” to receive images from the camera which has ROS messages of type “sensor_msgs/Image”. Additionally, a set of reference images or descriptors that represent the objects to be detected are required. These references are often stored in a YAML file or provided through ROS parameters.

Once an object is detected, the find_object_2d package generates the positions and orientations of the object in the camera’s field of view. This information is usually provided in the form of “find_object_2d_msgs/ObjectsStamped” messages published on the “/objects” topic.

The package’s performance, however, can be influenced by the quality of reference images and the choice of detection parameters. Thus, factors such as lightning conditions and object variability should be considered to achieve reliable object recognition.

4. Experiment setup

4.1. Requirements

Tested on:

- Lenovo laptop Intel® Core™ i3-6006U CPU @ 2.00GHz × 4, 8GB, 1 TB HDD with its integrated 720p HD (1280 x 720 pixels) and 30 fps Webcam
- Ubuntu 20.04.5 LTS
- ROS Noetic Ninjemys
- MoveIt

4.2. Software pipeline

To simplify the task of object tracking, vision packages are used to obtain the 3D Cartesian coordinates (XYZ) of the target. The tf (transform) is then used to maintain a tree of coordinate frames representing the position and orientation of the object of interest in the robot's workspace relative to its base frame. Once the coordinates of the target are found, MoveIt, a popular open-source robotics framework for motion planning and manipulation [6], then utilises this information to compute a trajectory to reach the target.

The camera is mounted separately from the robot's end-effector and remains fixed in space relative to the robot's base frame (link_base), see Figure 2. As a result, the position and orientation of the camera are constant relative to the robot's base frame. This makes it easier to determine the position of an object in the robot's workspace relative to the robot's base frame.

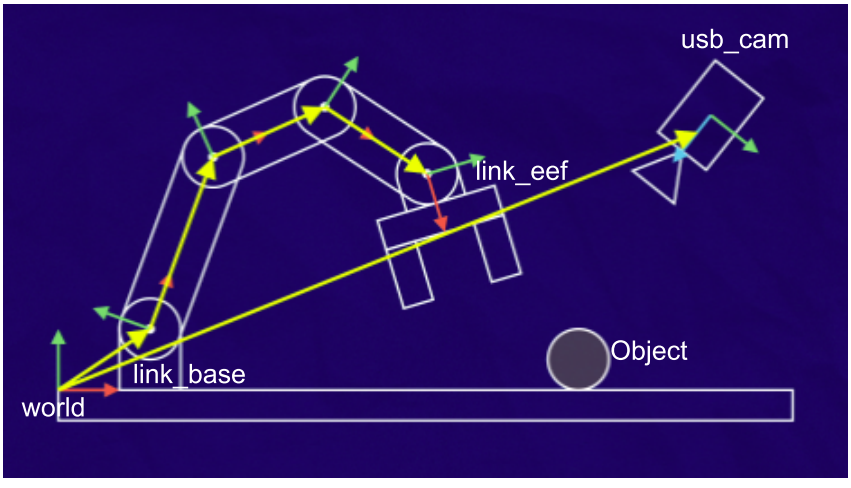


Figure 2. Eye-on-base configuration [7].

In this work, the ROS xArm7 robot package was used in the testing process. To control the xArm7 robot with the target locations of a detected object, a ROS package called *ar_arm_package* was developed. The following launch files are available in the developed package:

- xarm7_manipulator.launch: This launch file is used for configuring and initializing the xArm7 manipulator robot arm. It also includes parameters and settings required for the vision systems and the node responsible for controlling the robot.
- find_object.launch: This launch file is used to register a template image of the object that needs to be detected.
- track_find_object.launch: This launch is responsible for detecting the registered template image.

The ar_arm_package serves as the main package in controlling the robot. This package also contains Python scripts for the previously discussed vision systems. The scripts, namely *artag_detector.py*, *findObject.py,* and *opencv_detector.py* are present in the 'src' folder of the package. Details about the vision systems are discussed in subsections 4.2.1 - 4.2.3.

Additionally, the package has a script named "move_arm.py" for controlling the robot's arm using the MoveIt framework. This node listens to the /motion topic for the detected object pose and controls the EEF of the robot to reach the target's pose. See Figure 3, which illustrates the pipeline for this work.

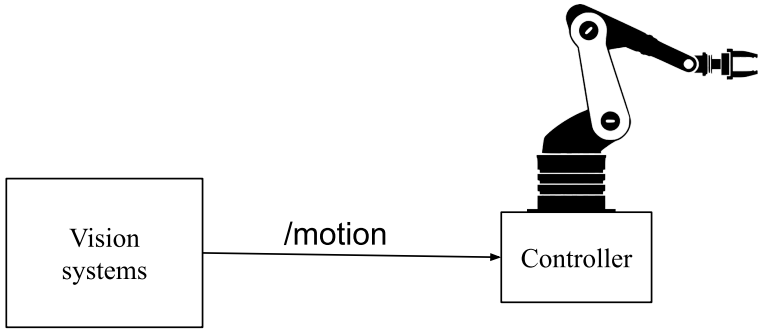


Figure 3. Flowchart of the implementation pipeline. The vision systems send geometry_msgs/Pose messages to the /motion topic. The node *move_arm.py* (known as Controller) subscribes to this topic, enabling it to receive the target pose details for effectively guiding the robot towards the identified target.

4.2.1 artag_detector.py

This node is responsible for detecting the AR markers in a camera image and sending the pose of the marker to the robot's main controller node, *move_arm.py*.
To initialize the node, you can use the command: *rosrun ar_arm_package artag_detector.py.*

This node subscribes to the /ar_pose_marker topic and listens for ar_marker_0 (AR marker) in the camera image. When the AR marker is detected, the pose is published to the /motion topic. A quick overview of how the node works can be seen in Figure 4.

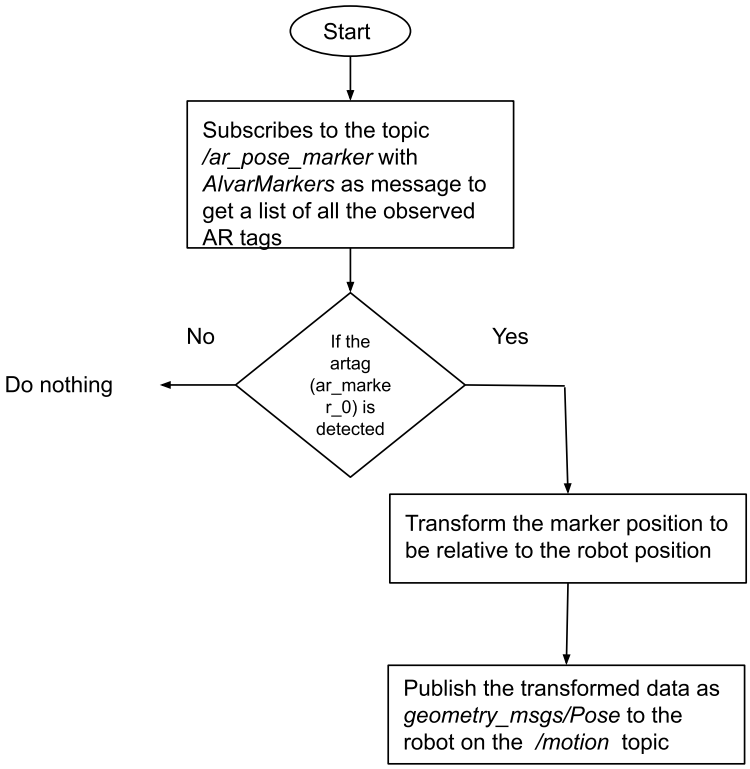


Figure 4. The flowchart for the artag_detector node

4.2.2 opencv_detector.py

The opencv_detector.py node detects objects using several computer vision approaches using the OpenCV framework. This node subscribes to the /usb_cam/image_raw topic, which contains ROS sensor_msgs/Image message type. To be able to use the camera data from ROS, the cv_bridge node helps convert the ROS camera images to a readable format(such as bgr8) for the OpenCV framework.

The node then converts the image from BGR to HSV colour space and applies colour detection thresholds to isolate specific colours.
It then detects keypoints (blob) in the resulting mask, calculates their positions relative to the camera frame, and broadcasts these positions as transforms between the camera and object frames using TF2 in ROS.

```
# Keypoint detection
keypoints = detector.detect(img_mask)
# …
# Object Transformation and Broadcasting
```

```
for i, kp in enumerate(keypoints):
```

```
  # …
  br.sendTransform(blob_pose)
```

The node continuously looks up for these transforms, handling exceptions, and publishes the detected object's target pose to the /motion topic.

```
while True:
        try:
            trans = self.tf_buffer.lookup_transform("link_base", "object", rospy.Time())
            # …
            self.pub_.publish(target_pose)
                                    except     (tf2_ros.LookupException,     tf2_ros.ConnectivityException,
tf2_ros.ExtrapolationException):
            # …
           continue
```

In this task, the object to detect was a green ball. To use this node for object detection for the manipulator you can use the command: *rosrun ar_arm_package opencv_detector.py*

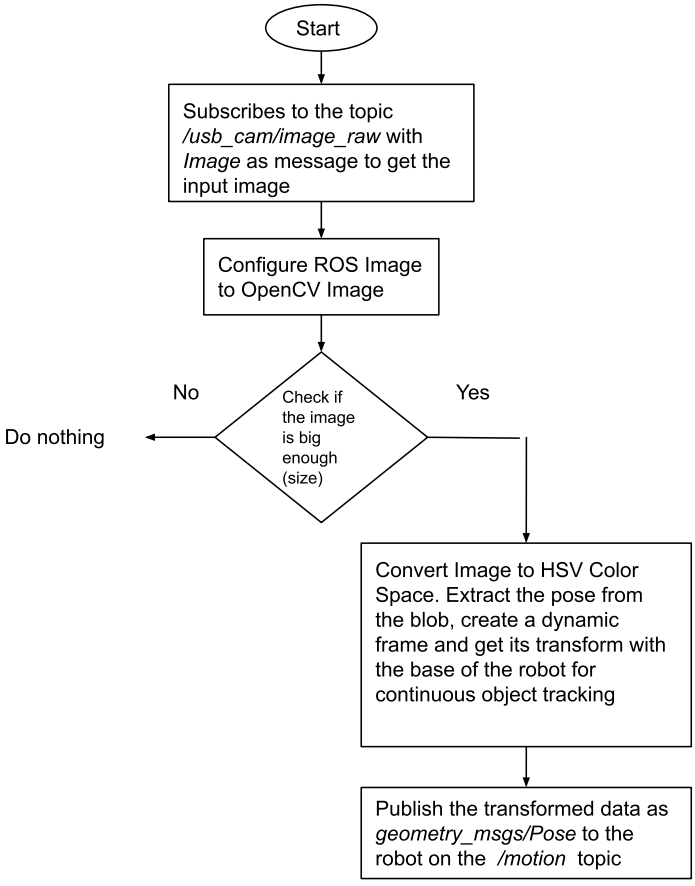


Figure 5. The flowchart for the opencv_detector node

4.2.3 findObject.py

The findObject.py node is responsible for publishing the pose of a target object to the manipulator via the /motion topic. The script subscribes to the /objectStamped topic, which contains messages of type find_object_2d_msgs/ObjectsStamped. The message data contains feature-detected points of our target object. Therefore to use this node, one has to launch the dependent launch file *track_find_object.launch* before running this script. Below are the complete launching steps:
roslaunch ar_arm_package track_find_object.launch
rosrun ar_arm_package findObject.py

The *track_find_object.launch* file launches the *find_object_2d* node, which is responsible for finding objects in images using a feature-matching algorithm. Feature matching is a technique that finds distinctive points in two images and matches them up. This allows the find_object_2d node to identify objects in a new image by comparing it to a reference image. In my case, I used the KAZE algorithm, which seemed to be the best feature descriptor as compared to SIFT and SURF since KAZE is robust to changes in illumination and viewpoint, however, comes with an additional computational cost [8].

After the features/corners are detected:

```
# Find corners Qt
# QTransform initialization with the homography matrix
qtHomography = QTransform(data[i+3], data[i+4], data[i+5],
                          data[i+6], data[i+7], data[i+8],
                          data[i+9], data[i+10], data[i+11])
# Map the template coordinates to the current frame with the shape of the template and homography
qtTopLeft = qtHomography.map(QPointF(0,0)) # Top left coordinates
qtTopRight = qtHomography.map(QPointF(objectWidth,0)) # Top right coordinates
qtBottomLeft = qtHomography.map(QPointF(0,objectHeight)) # Bottom left coordinates
qtBottomRight = qtHomography.map(QPointF(objectWidth,objectHeight)) # Bottom right coordinates
qCenter = qtHomography.map(QPointF(objectWidth/2,objectHeight/2)) # Centroid Coordinates
```

The findObject.py node extracts the object's location in the image frame and publishes it as a geometry_msgs/Pose message to the /motion topic.

```
W = abs(((TopLeftX+BottomLeftX)/2) - ((TopRightX+BottomRightX)/2))
H = abs(((TopLeftY+BottomLeftY)/2) - ((TopRightY+BottomRightY)/2))

area = H*W
diameter = math.sqrt((4*area)/math.pi)

object_pose.position.x = CenterX
object_pose.position.y = CenterY
object_pose.position.z = diameter
object_pose.orientation.x = 0.0
object_pose.orientation.y = 0.0
object_pose.orientation.z = 0.0
```

```
object_pose.orientation.w = 1.0

self.pub_.publish(object_pose)
```

The implementation of findObject.py is inspired by [9].

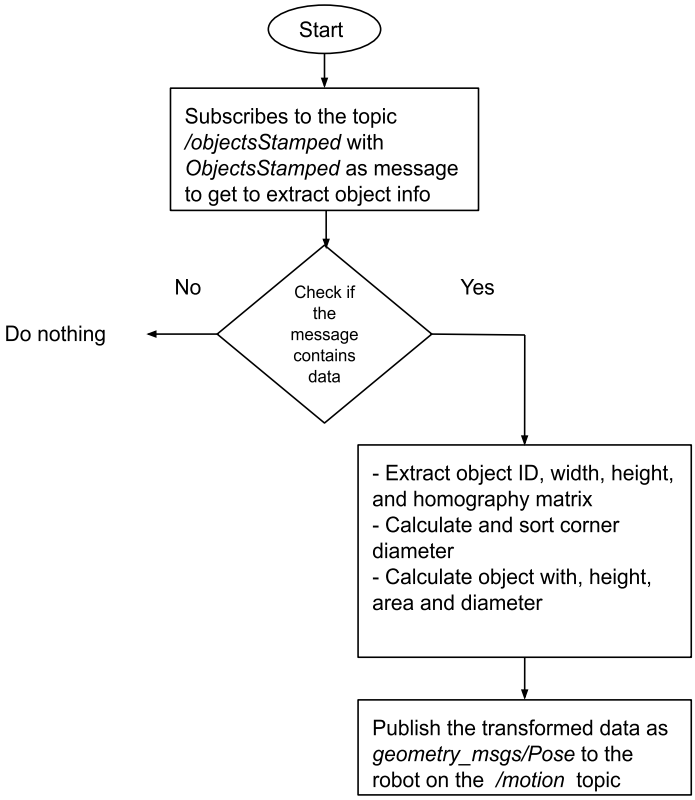


Figure 6. The flowchart for the findObject node

4.2.4 Reaching the target (move_arm.py)

Until now, the pipeline has discussed the vision systems that can potentially send the pose of an object to the /motion topic. The move_arm.py node is responsible for controlling the robot's EEF to reach the goal (detected object) pose. The node subscribes to the /motion topic, which provides the pose information of the object.

To reach the goal position, the robot must first find a solution to the inverse kinematics problem. This will tell the robot the joint angles it needs to move to in order to reach the goal. Once the robot finds a solution to the inverse kinematics problem, it can plan a motion to reach the goal. Fortunately, the MoveIt framework can help the robot with these tasks [6].

The move_arm.py script initializes the moveit_commander interface, which is a Python wrapper to communicate with the MoveIt framework. To control the robot's arm, it can be made possible using the MoveGroupCommander module from the moveit_commander package. Thus instantiating the move_group for the xArm7 is as follows:

```
move_group = moveit_commander.MoveGroupCommander("xarm7")
```

We can now use the callback function described below to send motion plans to the robot. Each pose message received from the /motion topic can be represented as a waypoint.
The move_group interface has a compute_cartesian_path command that computes a trajectory that takes into consideration a set of waypoints to reach the final waypoint. Note that waypoints are basically an array of 3D points that a robot can follow.

```
def pose_callback(pose_msg):

  # Plan and execute a Cartesian path to the target pose
  waypoints = [move_group.get_current_pose().pose, pose_msg]
  fraction = 0.0
  while fraction < 1.0:
     (plan, fraction) = move_group.compute_cartesian_path(
        waypoints,   # List of waypoints
        0.01,        # Step size for interpolation
        0.0          # Jump threshold
     )

  # Execute the planned trajectory
  move_group.execute(plan, wait=False)

  # clear the trajectory after execution
  move_group.clear_pose_targets()
```

The function compute_cartesian_path() takes as input waypoints of end effector poses and outputs a joint trajectory that visits each pose. The path is interpolated at a resolution of 1 cm thus 0.01, and the jump_threshold, 0.0, determines the maximum allowed distance between consecutive points in the resulting path. The function returns a tuple containing the fraction of the successfully followed path and the resulting RobotTrajectory.

Once a plan has been formulated, the robot executes and reaches the detected object pose given by any of the vision pipelines discussed earlier.

5. Results and Discussion

The results of using the artag_detector and opencv_detector nodes for guiding the robot arm have been shown in Figures 7 and 8, respectively.

It was observed that the artag_detector.py, based on the ar_track_alvar package, exhibited notable advantages in terms of integration ease and robust detection capabilities, including orientation awareness. In contrast, the opencv_detector.py algorithm, relying on the vision_opencv package, proved to be more complex to integrate, requiring an understanding of the OpenCV library, and displayed sensitivity to environmental factors such as lighting. In addition, it did not account for object orientation.

Additionally, a link to short animations can be accessed here (https://drive.google.com/drive/folders/1ZgR4-9qg4ERAmuBKxgtBkh4z-GxahpcV)

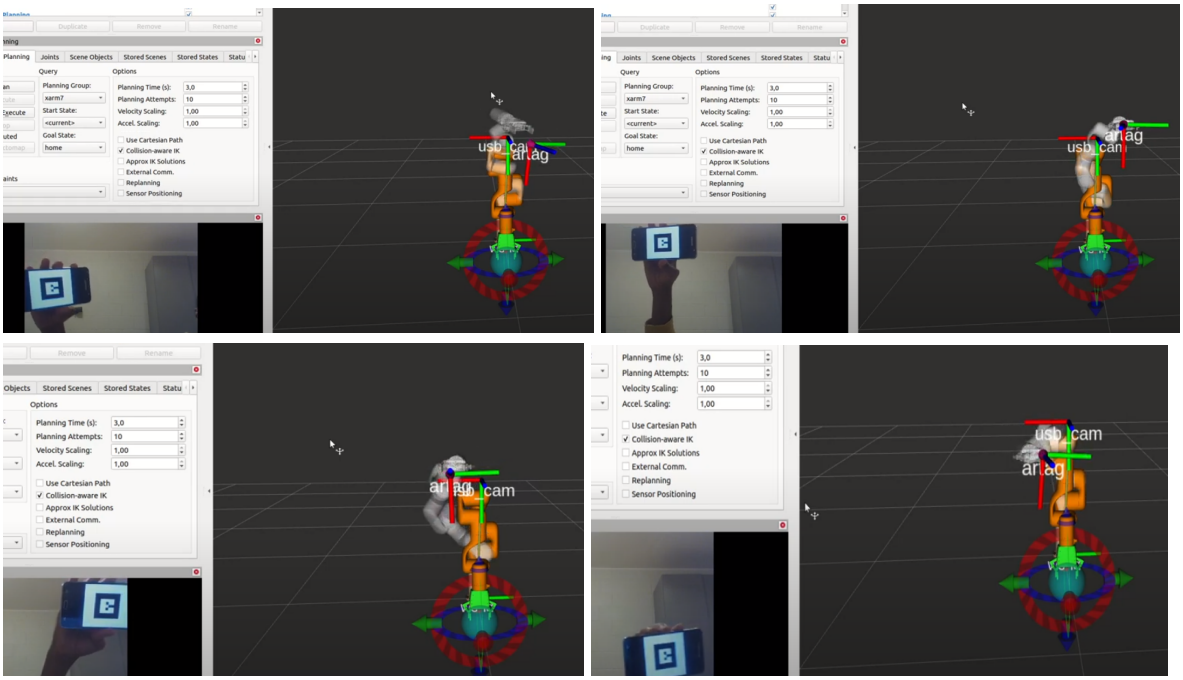


Figure 7. Artag is detected and tracked by the manipulator

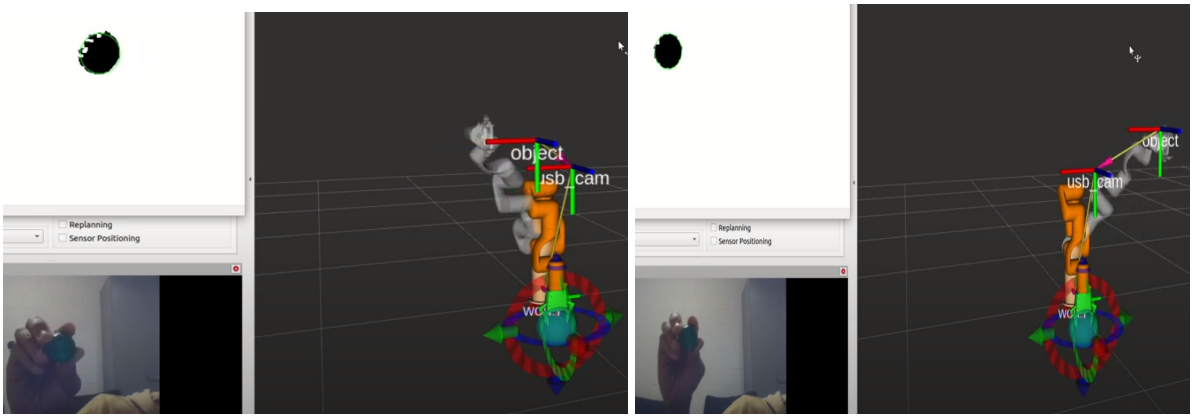
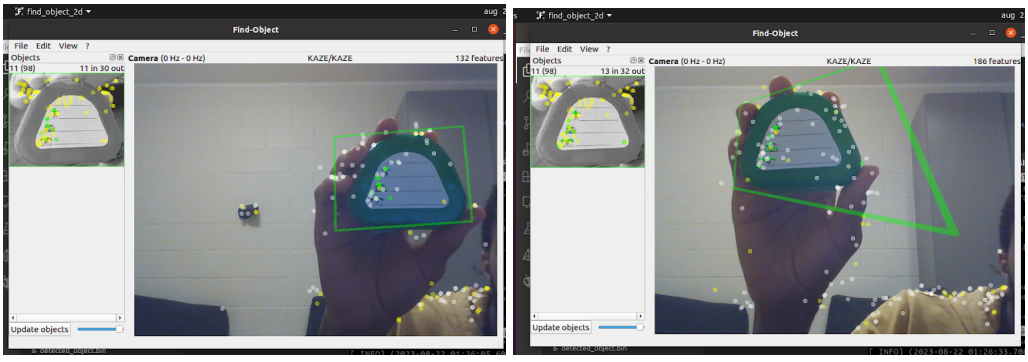


Figure 8. Object is detected and tracked by the manipulator

As you may have noticed, the implementation of the findObject node to guide the robot arm was not completed. This is because the values received on the /motion topic were not useful or within reach of the robot arm to reach the target. Additionally, several outliers were noticed when detecting the object, affecting how the bounding box was created around the object (see Figure 9b). However, the *find_object_2d* node itself is computationally expensive. As a result, further steps in using the package were discontinued because running the node and other ROS applications simultaneously rendered my computer unusable.



(a) (b)

Figure 9. An object is detected using find_object_2d. (a) bounding box aligns well with the object. (b) bounding box fits poorly

6. Conclusion

In this project, two vision pipelines were successfully developed and can be tested in real-world applications. Future work could focus on improving the accuracy and efficiency of the OpenCV blob detection method. Additionally, a thorough investigation for the inefficient find_object_2d method could be analyzed, and the potential outliers reduced to a higher degree. Moreover, an advanced Computer would be used in future as computer vision algorithms tend to be computationally expensive, and a lower-end computer such as mine is not the best option for multitasking.

Overall, this experience has been a great learning opportunity, and I am grateful for the knowledge and skills I gained through this course. The code and documentation for my work are available on my GitHub. I am confident that they will be valuable resources for anyone looking to work on similar solutions.

7. References

[1] Practical Experiences in Computer Engineering (6 ECTS) LOTI.05.043. (n.d.). Ois2.Ut.Ee. Retrieved August 2022, from https://ois2.ut.ee/#/courses/LOTI.05.043/details

[2] ar_track_alvar - ROS Wiki. (n.d.). http://wiki.ros.org/ar_track_alvar

[3] vision_opencv - ROS Wiki. (n.d.). http://wiki.ros.org/vision_opencv

[4] find_object_2d - ROS Wiki. (n.d.). http://wiki.ros.org/find_object_2d

[5] Khan, K. A., Konda, R., & Ryu, J. (2018). ROS-based control for a robot manipulator with a demonstration of the ball-on-plate task. Advances in Robotics Research, 2(2), 113–127. https://doi.org/10.12989/arr.2018.2.2.113

[6] MoveIT Motion Planning Framework. (n.d.). https://moveit.ros.org/

[7] J. (2021, September 24). Getting Ready for ROS Part 6: The Transform System (TF). Articulated Robotics. https://articulatedrobotics.xyz/ready-for-ros-6-tf/

[8] Alcantarilla, P. F., Bartoli, A., & Davison, A. J. (2012). KAZE Features. In European Conference on Computer Vision (ECCV) (pp. 214-227). URL: https://www.doc.ic.ac.uk/~ajd/Publications/alcantarilla_etal_eccv2012.pdf

[9] MikeS. (n.d.). *autonomous_landing_uav/object_detector/src/corners_detector.cpp at 4cb8de553f441c63126a2ba31a6db9772ade9ac9 · MikeS96/autonomous_landing_uav*. GitHub. https://github.com/MikeS96/autonomous_landing_uav/blob/4cb8de553f441c63126a2ba31a6db9772ade9ac9/object_detector/src/corners_detector.cpp