Robotics Project

Eleonora Bardelli, Ian Abuaf Pelo, Simone Tarenzi
{eleonora.bardelli, ian.abuaf.pelo, simone.tarenzi}@usi.ch

Abstract

Our initial project idea was to make a Robomaster EP grab an object marked with an ArUco marker, and then follow a Thymio, which goes in specific path where at the end it instructs the Robomaster to stop and put the object down.
The project uses a Thymio II robot, a Robomaster EP by DJI, robot simulator CoppeliaSim, Robot Operating System ROS, and the ArUco markers library.

1 Introduction

For our robotics project we wanted to implement a way for a Robomaster EP to grab an object identified with an ArUco marker, and then make it follow a Thymio robot.

We thought it would be interesting to combine the detection and recognition of designated identifiers with the more dynamic capabilities of a robot such as the Robomaster, mimicking how Kiva Robots from Amazon Robotics use QR codes to navigate warehouses, locate and then transport inventory [1].

The project's implementation required a series of well-defined steps: detecting and reading ArUco markers, alignment of the Robomaster to the object through visual servoing and PID control, and finally grabbing the object. The completion of these steps would then also allow us to easily make the Robomaster follow the Thymio.

We will provide a detailed discussion of each step in the following sections.

2 Theoretical Approaches

2.1 ArUco markers detection

The ArUco marker detection and identification was implemented using the ArUco class from the OpenCV library 3.4, which also offers extensive documentation [2].

ArUco markers are 2D binary-encoded fiducial patterns designed to be quickly located by computer vision systems [3]. They are composed of a wide black border and an internal binary black and white matrix, which can be of different sizes. Each unique configuration identifies a distinct number, which can then be read by a camera.

In order to use the functionalities offered by the library, we created a helper class containing all the methods needed for our ArUco detection implementation.

By using OpenCV, we can have access to all the markers seen by the camera, including the coordinates of their corners. The library also allows us to estimate the marker's pose in the world, which will later be explained in Section 2.3.

2.2 Camera alignment

An important step of implementing the project is to get the robot aligned with the an object marked by an ArUco marker so that the grabber is able to pick it up.

We looked at and partially implemented various ways of accomplishing this task, such as:

- Moving the sides of the marker towards the center of the camera;
- Using a pure visual servoing approach, computing velocity based on the uv coordinates of the marker corners and three stacked interaction matrices;
- Estimating the marker's pose and the camera's pose, and calculating velocities to align the camera to the marker. We eventually settled on this implementation.

2.2.1 PID implementation At first, we started implementing a more naïve approach for solving the problem of aligning to the ArUco marker. This versions consisted in a standard PID with values tuned heuristically. The aim was to align the centre of the ArUco marker - which was calculated starting from the position of the marker corners- to the centre of the camera. At every step the error was used to set the angular velocity around the z axis whereas another function was setting the direction of the spin.

Note that we decided not to include it in the final code since it was an intermediate step which presented some bugs solved by the refactoring of the visual servoing.

2.2.2 Visual servoing implementation The main idea of visual servoing is to use the camera of a robot to identify *visual features* and to move it towards a *target feature*.

For example we want to identify a marker in the camera and align it so that it stops at the center of the camera's vield of view. We can do this by using the marker's corner points as visual features. Figure 1 shows an example of such features. This is known as **Image Based Visual Servoing** (IBVS).

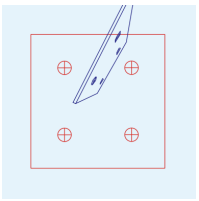


Figure 1. Illustration of visual features captured by the camera in blue and the target features in red [4].

In doing so we move the robot in an optimal spot where it can then move forward and grab the object that the marker is on.

Our visual servoing implementation started by trying to replicate the standard IBVS approach wherein we compute an interaction matrix L that, when applied to the desired velocity of the visual feature, gives us the velocity that should be applied to the robot [4].

This approach is however limited by the degrees of freedom of the robot. Since the Robomaster can't rotate around the X axis, part of the velocity information was lost.

In the end, we decided to instead estimate the marker's pose to then use that as a visual feature, and use the current camera pose as the target feature. This is known as **Position Based Visual Servoing** (PBVS), which in contrast to IBVS has a simpler velocity control law but requires pose estimation.

Thus we need to reconstruct the marker's pose, which is detailed in the next section, as well as the camera's pose, which is detailed in Section 2.4.

2.3 Marker pose estimation

As mentioned earlier, we used OpenCV to compute the marker's corners that we used in the previous version of visual servoing. OpenCV also allows us to estimate the pose of the markers, provided we supply a **camera matrix**.

So in order to effectively estimate a detected marker's pose, we first need to compute the camera's matrix, which is composed of the following:

$$\underbrace{\begin{pmatrix} f/\rho_w & 0 & u_0 \\ 0 & f/\rho_h & v_0 \\ 0 & 0 & 1 \end{pmatrix}}_{\text{Intrinsic matrix}} \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}}_{\text{Projection matrix}} \underbrace{\begin{pmatrix} R & & & T \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{Extrinsic matrix}}$$

The matrix transformations are applied from right to left:

- The **Extrinsic matrix** defines the pose of the camera: its translation and rotation with respect to the world frame.
- The **Projection matrix** takes the coordinates of an object and removes the Z coordinate, mapping them to the 2D plane from the camera's view.
- The **Intrinsic matrix** maps coordinates into screen coordinates, given a camera with focal length f and a screen with central point (u_0, v_0).

We implemented two approaches to compute this matrix. Initially, we used the methodology suggested and provided by OpenCV, which includes calibrating the camera with a chessboard pattern (see Section 2.3.1), but afterwards we decided to manually compute the optimal camera matrix from the camera's specifications (see Section 2.3.2) available in the Robomaster EP's user manual [5].

2.3.1 Camera calibration OpenCV provides a way to calibrate a camera, which is the recommended way to compute the camera matrix for real robots [6].

For this purpose, we built a separate ROS node that handles calibration and a CoppeliaSim scene with a chessboard pattern. The code is mostly inspired by the OpenCV calibration tutorial [7].

Figure 2 shows the calibration scene. It includes a script in the chessboard plane that allows it to move during calibration. The calibration script will periodically send a message to a ROS topic that will randomize the rotation and position of the chessboard, allowing for a more accurate calibration.

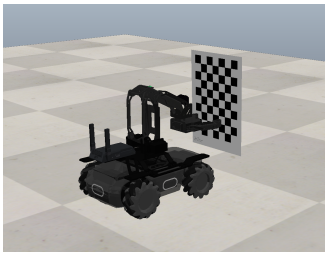


Figure 2. Initial CoppeliaSim calibration scene setup.

This approach has the advantage of not only giving the intrinsic matrix, but also the extrinsic matrix, from which we can then extract the camera pose.

Additionally, it gives us some distortion coefficients which are useful to have for real cameras, but are not really useful for the simulated environment we are testing.

The downsides are that the estimation is not always perfect and that it requires an additional step to compute (running the calibration script on the calibration scene).

2.3.2 Optimal camera matrix The optimal camera matrix can be easily computed from the camera's field of view and image size.

See this link for a more detailed explanation.

When using this process we compute only the camera intrinsic matrix. We simply consider the camera to be at the origin of the world frame which removes the need for the extrinsic matrix.

2.4 Calculating the camera pose

Our first implementation was done by moving the camera closer to the robot's gripper. This simplified our calculations, as we could assume the camera's pose to be roughly equivalent to the gripper's.

It also had the advantage of putting the camera on a surface that is kept level to the ground thanks to the gripper's gimbal. Thus, no need to keep track of the camera's pitch.

This however, was undone later into the project for two reasons:

1. Once the gripper picks up an object, the camera's vision is blocked.
2. It allows our code to run on the original RoboMaster EP without any additional external modification to the robot.

The default position of the camera is near the beginning of the second bracket of the robot's arm. Therefore in practice the camera's position and rotation with respect to the robot's base is dictated by the rod, which itself is attached to the first bracket of the arm.

This means that aligning the camera to the object now takes an additional step. We must find the transformation from the robot's camera to the gripper. Figure 3 is a diagram from the OpenCV documentation that shows what we want to compute, the transformation ${}^cT_g$.

While OpenCV does provide calibration functions for these transformations, we instead compute the transformations ourselves given the joint states.

Specifically, we have a custom script in the CoppeliaSim scene that broadcasts to a ROS topic the transformation ctg

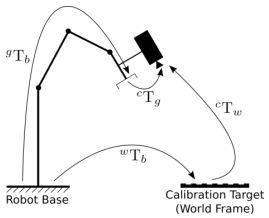


Figure 3. Camera calibration blueprint from OpenCV [7].

(gripper to camera) in the form of a vector for translation and quaternion for rotation. This script is found attached to the "camera_attach" node in the Robomaster.

3 Implementation

The general setup is structured with three states. Each state is defined by a class. Each class is a ROS node and whenever the state completes the node is destroyed and replaced with the new state.

This ensures that every state has its own separate set of timers, subscribers and publishers.

The states which have been created are:

- Align state: where the robot aligns with the ArUco. For more details see Section 3.1.
- Move Forward state: where the robot moves forward and grabs the object. For more details see Section 3.2.
- Follow Thymio state: where the robot searches for the Thymio and follows it. For more details see Section 3.3.

3.1 Aligning to a marker

In the aligining state, the Robomaster tries to align to the marker using the implemented version of the Visual Servoing (described in Section 2.2.2). As soon as the alignment is satisfactory, the state is switched to the move forward one. The stopping condition of the alignment has been set as a threshold on the speed, since the slower the robot moves, the closer to the correct alignment it is. We tested this implementation in many scenarios to make sure the alignment was robust at various different angles.

3.1.1 Target Pose As described in Section 2.2.2, we use the marker's pose as the visual feature to align and the camera pose (origin) as the target feature. In practice, this means that the camera will attempt to *move into* the marker rather than remain just in front of it.

So we added an *offset* to the marker pose so that we end up at a fixed point in front of it.

In Section 2.4 we also describe how to get the gripper to camera transformation. We apply this transformation to the marker pose as well in order to move the gripper to the marker rather than the camera.

The marker pose therefore is computed as follows:

$$\underbrace{\begin{pmatrix} R_m & & & T_m \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{Marker pose from OpenCV}} \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0.05 \\ 0 & 0 & 1 & 0.32 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{Offset}} \underbrace{{}^{c}T_{g}}_{\text{Gripper to camera}}$$

3.2 Move forward and grab

The move forward state contains an heuristic approach to get close to the object and grab it. The odometry presented some minor issues (see Section 4.1.1) but it has been used to get closer to the object. After the alignment phase, the heuristic we employed consisted in moving forward from the starting position (which is the position after the alignment) for 0.14 meters. This allows the gripper to be wrapped around the object to grab when the robot stops moving forward. Even this state has been tested in a couple of different scenes in order to set a suitable stopping value. After the Robomaster has moved forward, we send an asynchronous call to the action client of the gripper*, setting as a goal to close it. In order to be sure the action is executed, we also insert a delay of one second in the code. At this point, since in the next stage the Robomaster needs to follow the Thymio, we used another heuristics to try to avoid objects to hinder the Robomaster's path. This second heuristics consisted in the use of odometry to move backward for 0.25 meters. Note that for the grabbing part we had some limitations 4.1.4 but we also tried with objects of different dimensions as shown in Figure 4.

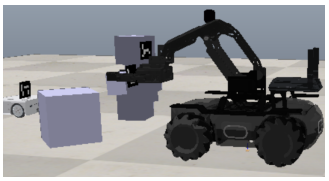


Figure 4. The Robomaster using the gripper to grab a small cuboid.

3.3 Spot and follow the Thymio

In this state the Robomaster follows the Thymio as soon as it is spotted. The identification is carried out with its ArUco which has been posed on a tile (see Section 4.1.3). Subsequently, the Robomaster, with the object still in its gripper, tries to align to the Thymio using visual servoing 2.2.2. Once it is close enough to the Thymio, it drops the object. As mentioned before, the proximity to the Thymio has been implemented with another heuristics which combines the current velocity given by the visual servoing (if low it means that the Thymio is close). The Robomaster then drops the object after three seconds.

3.4 Experimental setup

For our experiments, apart from the standard ROS libraries for developing robotics application and the CoppeliaSim simulator, we used OpenCV. In order to correctly run our code we needed to uninstall the current version of OpenCV and installing opencv-contrib-python version 4.9.0, as described in the README file.

3.4.1 Robomaster EP The Robomaster is set up as shipped from DJI. Initially, as mentioned in Section 2.4, we moved the camera on top of the gripper, but we moved it

*The ActionClient for the gripper [8] has type GripperControl [9]. The GripperControl takes as argument the goal (which can be: close, open or pause) and the force to apply.

back to its default position after the issues encountered with blocked vision.

We didn't make use of every part and sensor available on the Robomaster (like the built-in hit detectors or controller antenna, see Figure 5), but we still managed to obtain the results we aimed for.

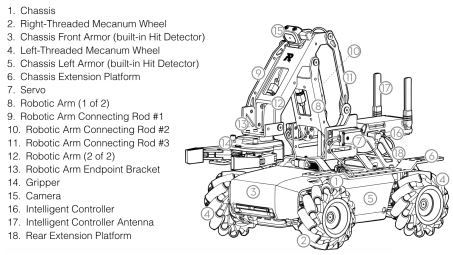
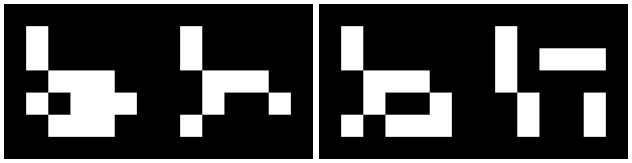


Figure 5. Parts list of the Robomaster EP by DJI as illustrated in its official manual [5].

3.4.2 ArUco markers We used four different ArUco markers in our setup, as they can be seen in Figure 6. They all use a 5x5 matrix.



(a) ArUco ID 55. **(b)** ArUco ID 56. **(c)** ArUco ID 57. **(d)** ArUco ID 74.

Figure 6. ArUco markers used in our setup. The ones with ID number 55, 56 and 57 are used to identify the boxes, while 74 is used for the Thymio.

3.4.3 Coppelia scene The starting Coppelia simulation scene is composed of: a Robomaster EP, three boxes textured with a distinct ArUco marker, each one positioned at different heights on top of cuboids of various sizes, and a Thymio II robot, also marked with an ArUco marker (see Figure 7).

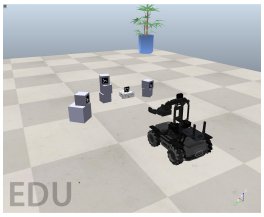


Figure 7. Starting environment of the simulation, with a Robomaster EP, three ArUco marked boxes at different heights, and a Thymio II.

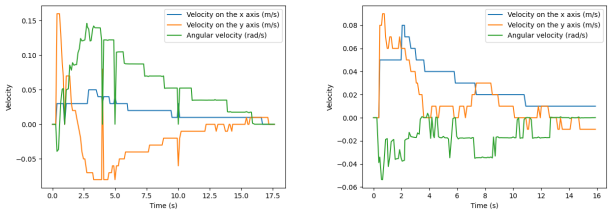
3.4.4 Usage The project comes with a `README.md` file which contains instructions on installing dependencies, running the (optional) calibration node and running the main node, describing also the possible parameters.

4 Results

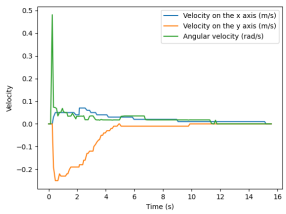
The following tests were done all on the provided scene, using the optimal camera matrix (no calibration). We look specifically at the performance of the VS controller while aligning with the marker to pick up the object. The tests were done on all three marked objects in the scene.

First, we see in Figure 8 the three main velocities of the robot (x, y and angular) over time as they converge to 0. Since the updates to the goal pose and thus of the velocities are done every time a new image is received from the camera, we see the staircase effect.

These velocities are taken from the robot's odometry, thus they are not fully accurate.



(a) Targeting marker id 55. **(b)** Targeting marker id 56.



(c) Targeting marker id 57.

Figure 8. Velocity over time on the provided scene, during first alignment.

We also plotted the robot's gripper position and orientation as it moves in Figure 9. This time this data is taken directly from CoppeliaSim (with a script that publishes it to ROS inside the Robomaster).

The most interesting behaviour we observed is when going for marker id 55 (the leftmost one), the robot will start by moving left but then quickly readjusts to the correct direction. This can be observed in both Figure 8a and Figure 9a.

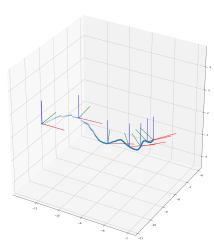
4.1 Issues and Limitations

4.1.1 CoppeliaSim The CoppeliaSim simulated environment has presented some limitations to the implementation. In the first place, the odometry was sometimes a bit unprecise or not updating properly. In order to partially solve this problem we decreased the dynamics dt parameter in the simulation time steps setting from 0.0050 to 0.0045.

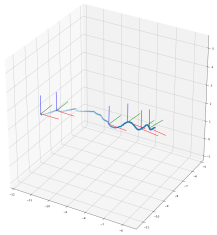
4.1.2 Obstacles We did not implement any form of obstacle avoidance, also due to the lack of proximity sensors on the Robomaster.

This causes the robot to sometimes get stuck while aligning with a marker.

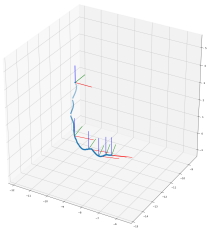
4.1.3 The ArUco tile on the Thymio In order to implement the *Follow Thymio state* 3.3 we needed to place the tile vertically as shown in Figure 10b. We tried to modify the code in order to fix this problem, but this resulted in the marker being too far away to be detected due to the low camera resolution.



(a) Targeting marker id 55.

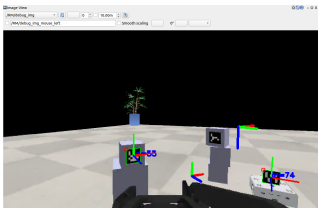


(b) Targeting marker id 56.

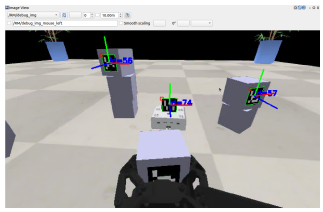


(c) Targeting marker id 57.

Figure 9. Gripper pose during first alignment.



(a) View from the Robomaster EP camera, while the robot is aligning to the selected ArUco marker.



(b) View from the Robomaster EP camera, while the robot is following the Thymio.

Figure 10. View of the Robomaster's camera during execution

Due to the way the camera is positioned on the robot, it works best when the marker is upright on the ground then when it's laying flat.

Sometimes, the marker would also be assigned an incorrect identifier when positioned farther away.

4.1.4 The object to grab Since the code was implemented to grab an object with an ArUco marker tile placed on the surface, we could only test the grabbing part cubic object.

5 Discussion and conclusion

In conclusion, our work mainly focused on exploring the possibilities and limitations of a visual servoing implementation 2.2.2, combined with the grabbing of a cubic object. We also aimed at making our work as robust as possible in accordance to the time and restrictions 4.1 we had. Possible future work could involve the modification of the heuristic parts in Section 3.2 and Section 3.3, exploiting different strategies or even mounting a proximity sensor on the Robomaster EP to avoid obstacles. Furthermore, a more pure and stable visual servoing could improve the speed and fluidity of the alignment.

6 Appendix

The complete project code is accessible at the following GitHub repository: `https://github.com/Bluxen/Robotics-Project.git`.

Additionally, videos showcasing successful runs and various failure cases are available at: `https://drive.google.com/drive/folders/1vxGOCeYLTzSv8-Fjd0xYIYAxzXamwBTp?usp=drive_link`.

References

[1] Evan Ackerman. Meet the drone that already delivers your packages: Kiva robot teardown, 2013. URL `https://robohub.org/meet-the-drone-that-already-delivers-your-packages-kiva-robot-teardown/`.

[2] OpenCV Development Team. Detection of aruco markers. URL `https://docs.opencv.org/3.1.0/d5/dae/tutorial_aruco_detection.html`.

[3] S. Garrido-Jurado, R. Muñoz-Salinas, F.J. Madrid-Cuevas, and M.J. Marín-Jiménez. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition*, 47(6):2280–2292, 2014. URL `https://www.sciencedirect.com/science/article/pii/S0031320314000235`.

[4] François Chaumette and Seth Hutchinson. Visual servo control. i. basic approaches. *IEEE Robotics & Automation Magazine*, 13(4):82–90, 2006.

[5] DJI. *RoboMaster EP User Manual*, April 2022. `https://dl.djicdn.com/downloads/ROBOMASTER_EP/20220429UM/RoboMaster_EP_User_Manual_v1.2_EN_1.pdf`.

[6] OpenCV Development Team. Camera calibration and 3d reconstruction, 2023. URL `https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html`.

[7] OpenCV Development Team. Camera calibration with opencv, 2023. URL `https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html`.

[8] Open Robotics. Writing an action server and client (python), 2024. URL `https://docs.ros.org/en/foxy/Tutorials/Intermediate/Writing-an-Action-Server-Client/Py.html#writing-an-action-client`.

[9] Jérôme Guzzi. robomaster_msgs: Grippercontrol action, 2024. URL `https://jeguzzi.github.io/robomaster_ros/robomaster_msgs.html#robomaster_msgs.action.GripperControl`.