1 | Experimental Setups and Demonstrations

This chapter will describe the experiments conducted to demonstrate the capabilities of the mobile manipulation robotic system. The experiments consist of three demonstrations named Aruco Follower, Button Presser, and Object Picking. These demos are meant to showcase the capabilities of the robotic system in performing various tasks such as following a marker, pressing buttons, and picking objects. The first demo is meant to be a preview of the robotic arm's autonomous control software. The second demo is meant to showcase the capabilities of the entire system in performing high-level tasks in industrial environments, such as pressing buttons on an industrial control panel. The third demo is meant to showcase the mobile manipulation capabilities in an agricultural environment, such as picking fruits from trees.

The experiments are conducted in a controlled but challenging and realistic environment to test the robustness and reliability of the system. The demos have been tested inside the Artificial Intelligence and Robotics Laboratory at Politecnico di Milano. The laboratory has enough space to allow testing the efficiency of the autonomous navigation software while testing also the obstacle avoidance algorithm in a cluttered and changing environment.

The objectives of these demonstrations are to stress-test the robotic system and to evaluate the performance of the various software components and how well they integrate with the hardware components. The experiments will also highlight the challenges faced during the development and testing of the robotic system and how they were overcome. The demos are meant to be a proof of concept of the robotic system's capabilities in simulated scenarios that are as close as possible to more realistic environments. They are also meant to show the potential of mobile manipulation robots in performing various complex tasks that are currently performed by humans. The objective is not to replace humans but to assist them in performing tasks that are dangerous, repetitive, or require high precision.

1.1. Aruco Follower Demo

The Aruco Follower demo is a simple demonstration of the robotic arm's autonomous control software. The demo consists of the robotic arm following an Aruco marker with the end effector. The demo is meant to showcase the motion planning and execution libraries with MoveIt2 and the integration of the Aruco marker detection and pose estimation algorithms with the robotic arm's control software. The demo tests the autonomous control software of the robotic arm in tracking and following a moving target. Figure 1.1 shows the demo in execution, with *MountV1* as end effector (but without the cylinder presser). The cardboard shown to the camera displays an Aruco marker, tracked by the robotic arm.

In this demo, the end effector, equipped with the camera, will move in a position and orientation that points toward the center of the Aruco marker. The algorithm for **tracking the marker** computes the end effector's target position such that the arm can always reach it, and the orientation of the end effector is always pointing towards the marker. The position of the end effector will be exactly the marker's position if the marker is sufficiently close to the arm, otherwise, the end effector will be positioned in a way such that the camera points toward the marker. If the marker is not visible, the end effector will remain in the last known position until the marker is visible again. If the marker moves to a different position in the camera frame and stays visible, the algorithm will compute the new target pose and the end effector will move to it. The computation of the end effector's target pose is based on geometrical calculations from the marker's position in the camera frame. The marker's pose is transformed in the robot arm's base frame, and the target pose is computed based on the robotic arm's workspace. The target pose is then sent to the motion planning and execution libraries to compute and execute the trajectory to reach the target pose.

I tested also the **MoveIt2 Servo** node to control the end effector's position and orientation in realtime using the Aruco marker's pose as the target pose. This feature is available only in ROS2 Iron, at the moment of writing this thesis. Unfortunately, the MoveIt2 Servo node was not stable enough to be used in the demo, in fact, the generated trajectories were very jerky and the end effector was not able to reach the target pose in a short time. The problem exists because the MoveIt2 Servo node is still in development and it is not yet stable enough to be used in real-time applications. It also requires a precise PID tuning of the closed loop joint trajectory controller to work properly, which is not easy to do in a short time. So eventually I gave up on using the MoveIt2 Servo node and I used the standard MoveIt2 planning and execution libraries.

Figure 1.1: Aruco Follower demo during execution. The cardboard shows an Aruco marker.

1.2. Button Presser Demo

The "Button Presser" demo is a complex demonstration that showcases the potential of mobile manipulation robots in performing high-level tasks in industrial environments. The demo consists of the robotic arm pressing buttons on an industrial control panel, in an autonomous way. The objective is not to have the fastest and most efficient solution for pressing buttons but to show the capabilities of the robotic system in performing tasks that are currently performed by humans. The demo is meant to be a proof of concept of the robotic system's integration of multiple software components and the orchestration of the robotic arm and mobile base to interact with the environment without human intervention.

This demo was a request by a company that is interested in using mobile manipulation robots in their industrial production plant, for monitoring sensors and various equipment, and intervening in case of emergency, while avoiding human intervention in dangerous environments. The objective was to demonstrate the feasibility of the mobile manipulation robot in interacting with the control panel autonomously and effectively. It was also important to demonstrate the system's reliability and robustness in performing such a

complex task, even though it was not required to be the fastest or most effective solution, in terms of accuracy in pressing buttons and the time taken to press all the buttons.

1.2.1. Buttons Setup Box and End Effector Setups

For this demo, the *MountV1* was employed, mounted on the robotic arm's end effector. The *MountV1* is a custom-designed 3D-printed mount attached to the cobot's flange, which allows the installation of the stereo camera and the cylinder presser. The cylinder presser is a cylinder-shaped tool used for pressing buttons. In *MountV1* the camera is placed in front of the flange, and this resulted in a reduced field of view of the camera. Despite this non-ideal camera position, the camera was able to detect the Aruco markers on the control panel and the buttons to be pressed. Since the *MountV1* proved to be effective in the demo, I didn't apply any structural changes to the design. The only change that I made was to shorten the cylinder presser to prevent the mobility of the end effector from being reduced due to the length of the tool.

Further in the development of the demo, I switched to *MountV2*, which is the next version of *MountV1*. The biggest improvement of *MountV2* is the camera's position, which is placed on the wrist of the cobot, allowing a greater field of view of the camera. This proved more effective in finding the Aruco markers on the control panel from a vicinity.

Using *MountV1*, the end effector was slipping during the execution of the linear trajectories, due to the roundness of the button caps. With *MountV2*, instead of having a cylinder presser, the end effector presents a vacuum suction cap on its tip, which can be also used to press the buttons. This solution is more effective in pressing the buttons, thanks to the greater friction between the plastic button cap and the silicon suction cup. The vacuum suction cap presses the buttons effectively, and reliably, with less slipping.

The Realsense camera was used only for Aruco markers detection and pose estimation. So the software component related to the perception of the control panel didn't make use of the infrared cameras for depth estimation. The position of the markers is in fact computed using the RGB image as input and the equations for 3D camera projection on a 2D matrix for the pose estimation.

The control panel is a custom-designed box with 3 buttons of different sizes and 7 Aruco markers (having dictionary 4x4) placed around the buttons. Figure 1.2 shows the control panel. The control panel is mobile, meaning that it can be moved around the laboratory to test the robustness of the system by pressing the buttons in an arbitrary location and orientation. The control panel has also 3 different lights, one for each button, that indicate whenever a button is pressed. The control panel is connected to the power supply that

Figure 1.2: Control panel with 3 buttons and Aruco markers.

can be plugged into a power outlet. I also had to create the internal circuitry to link the buttons to the lights and connect everything to the power supply. The control panel is also equipped with a power switch that can be used to turn on and off the lights.

The reason for having multiple Aruco markers instead of just one or three for the three buttons, is to have a more robust and reliable detection of the buttons' positions. Having multiple markers allows the system to detect the buttons' orientation in a more precise way that is also robust to noise. The markers are placed around the buttons in a way that the plane estimation algorithm can compute the plane of the markers effectively, relying on a greater number of points to estimate the plane.

1.2.2. End Effector Positioning and Linear Trajectories

The implementation of the algorithms for pressing buttons and planning the trajectories is handled inside ROS2 nodes. One ROS2 node subscribes to the Aruco markers detection topic containing the estimated markers' poses and their IDs. Given the markers' poses, and the relative positioning of the markers with respect to the buttons, the node computes the position of the buttons in the robot's base frame. The node also computes the orientation of the end effector, such that the tip of the end effector faces the plane of the markers orthogonally, and this is needed to compute the orientation required to press the buttons.

The **algorithm** for computing the sequence of target poses that the end effector must reach uses the **estimated position and orientation of the buttons**. For each button, the algorithm computes the target pose that "sits" above the button, meaning that the end effector faces the button orthogonally from a fixed distance. Then the algorithm generates the linear path that the end effector must follow to reach the pose where the button is pressed. The linear path is then reversed to get the path required to lift the end effector from the button in order to release it and return to the starting point above the button. This sequence of target poses computation is repeated for each button on the control panel. The ROS2 node uses these target poses and linear paths to plan and generate the trajectories for the arm to reach the target poses and follow the linear paths. Once the trajectory plans are generated, they are executed.

The linear motion trajectory generation is the most probable point of failure for the trajectory planner, because the algorithm that computes the trajectories must take into account several constraints, such as the static collisions with external bodies and the robotic arm's self-collision mapping. MoveIt2 is a library that interfaces directly with the motion planners, which in turn use the inverse kinematic solvers to compute the joint configuration required to move the end effector to the target pose. There are two main methods to generate a linear trajectory for the end effector in cartesian space:

- **Cartesian Linear Motion Planning generation via a sequence of waypoints**: this method generates a sequence of pose waypoints (sequence of positions and orientation) that the end effector must follow to reach the target pose. The trajectory is generated by interpolating the sequence of waypoints in space and time. The trajectory planner generates a trajectory that follows the waypoints with maximum deviation defined as the *end effector jump*, which is the maximum total deviation in joint space that each joint can have from the average joints' positions. This parameter controls how much the joints can move from their average configuration during the execution of the linear trajectory. Setting the end effector jump to zero means that the joints will not move from their average configuration, and the end effector will follow the waypoints exactly.
- **Constrained Cartesian Motion Planning**: this method creates a motion plan request with the addition of position and orientation constraints on the end effector that must be respected while moving toward the final pose. The constraints are defined as a position box constraint, meaning that the end effector must stay within a box (parallelepiped) in the cartesian space, and an orientation constraint, defined as a quaternion representing the orientation and the maximum angle of deviation from the orientation axis defined by the quaternion.

Both methods were tested and implemented successfully to be used with the Igus Rebel cobot. The constrained cartesian motion planning method proved to be unreliable due to the high probability of not being able to generate a valid trajectory. Adding only one constraint to the motion plan made the trajectory planner able to find a solution most of the time. But adding two constraints, one for the position and one for the orientation, resulted in failed trajectory generation almost every time. After some research, I found out that the constrained cartesian motion planning method is usable and tractable only for 7-DoF robotic arms. The Igus Rebel cobot has only 6 DoF, and this makes the constrained method not suitable for this robotic arm. The reason behind this is that constraining a motion plan with two constraints for a 6-DoF robotic arm results in an overdetermined system of equations, and the trajectory planner is not able to find a feasible solution, because the constraints add more equations than the number of DoF of the robotic arm, resulting in a system of equations with no feasible solution. This is mostly due to a limitation of the MoveIt2 library and the trajectory planners used since they work well with 7-DoF robotic arms, such as the Franka Emika Panda.

Due to this limitation, only the cartesian linear motion planning method was used to generate linear trajectories following a sequence of waypoints. The end effector's jump was set to zero to make the end effector follow the waypoints exactly, to avoid random jumps in the cobot's configuration. The linear trajectories were generated successfully and executed correctly most of the time. Sometimes the trajectory planner failed to find a solution, even when attempting to create a trajectory plan multiple times. This was due to the complexity of the environment that the planner must take into account. After many attempts with different motion planners, I found that no planner can guarantee to find a feasible solution, it exists, 100% of the time. The Pilz industrial motion planner proved to be the most reliable and robust planner for generating linear trajectories.

1.2.3. Mobile Button Presser Demo

The "mobile button presser" demo is a complex demonstration aimed at showcasing the mobile manipulation robot's capabilities in pressing buttons on an industrial control panel autonomously. The demo consists of the mobile manipulation robot navigating to the control panel's location, detecting the control panel, and pressing the buttons in a predefined sequence.

The **complete demo** consists of the following steps:

1. The mobile manipulation robot starts from a random location in the laboratory and does not know where the control panel is located.

2. The robotic arm moves around the camera mounted on the end effector to search for a specific Aruco marker in the laboratory, which signals where the control panel is located.
3. Once the specified marker is detected, and its distance is computed, the computer algorithm computes the position of the control panel in the map frame of reference.
4. The robotic arm parks itself to not obscure the field of view of the LiDAR, which is essential for localization and autonomous navigation.
5. The mobile base navigates to the control panel's location autonomously, while avoiding obstacles in the environment.
6. The mobile robot parks in front of the control panel at a fixed distance, facing the opposite side of the control panel, so that the robotic arm can move freely without colliding with the mobile base and reach the buttons. Figure 1.3 shows this stage of execution of the demo.
7. The robotic arm moves around the camera mounted on the end effector to search for the Aruco markers on the control panel, indicating the locations of the buttons to be pressed.
8. Once the camera detects the Aruco markers, the computer computes the position and orientation of the buttons placed on the panel in the robot's base frame.
9. For each button, it computes the end effector target poses in the cartesian space required to press the buttons. It also computes the linear paths for the end effector necessary to press and release the buttons.
10. The robotic arm executes the computed trajectories, and the end effector presses the buttons in the predefined sequence. Figure 1.4 shows the end effector pressing one of the buttons, at this stage of execution of the demo.
11. The control panel lights up the corresponding light for each pressed button.

This complex demonstration is handled by one ROS2 node (action client) that orchestrates the actions of the mobile base and the robotic arm. The client node sends the goal to three different action servers:

- **Parking Action Server**: this server is responsible for computing the parking algorithm, given the position of the control panel in the map frame. After the parking pose is computed, the server sends a navigation goal to the nav2 stack to navigate autonomously to the parking pose. The action returns the result containing

information about how close the robot is to the parking pose when the autonomous navigation is completed.

- **Button Finder Action Server**: this server is responsible for searching for a specific Aruco marker in the room. The server executes a "searching movement", which is a sequence of predefined poses that the robotic arm must follow to search for the marker. The server also executes all trajectories until the marker is detected. The server returns once the marker is detected, and the distance from the cobot to the marker is computed.
- **Button Presser Action Server**: this server is responsible for pressing the buttons on the control panel. It first computes the "searching movement" to search for the Aruco markers on the control panel. Once the markers are detected and the control panel is located, the server computes the target poses and linear paths for pressing the buttons. The server then executes the trajectories to press the buttons in the predefined sequence. The server keeps track of how many trajectories have been planned and successfully executed, with respect to the total number of trajectories to be executed. The server returns the success rate of the executed trajectories. The server terminates its execution once all the buttons have been pressed.

1.2.4. Experimental Challenges and Solutions

One of the main challenges faced during the development of the "Button Presser" demo was the **reliability of the trajectory planner** in generating linear trajectories for the end effector. The planner failed to find a feasible solution frequently even after multiple attempts. I did not manage to overcome this problem, but I optimized the trajectory planner's parameters to increase the probability of finding a feasible solution, by incrementing the tolerances and the number of attempts to generate a trajectory plan.

I used 4x4 Aruco markers for locating the control panel from a distance, instead of the 7x7 markers that I initially used because the 4x4 markers require fewer pixels to be represented, therefore the minimum area of pixels required for detection is smaller. This implies that the 4x4 markers can be detected from a greater distance than the 7x7 markers.

Another issue encountered was the **imprecision of the robotic arm's motors' encoders**, which caused the end effector to not reach the target pose with the desired precision. This problem was partially overcome by adding a function that artificially compensates for the error in the end effector's position, by adding a small offset to the target pose, based on empirical measurements of the error. This solution was not ideal, but it was effective in increasing the precision of the end effector's position.

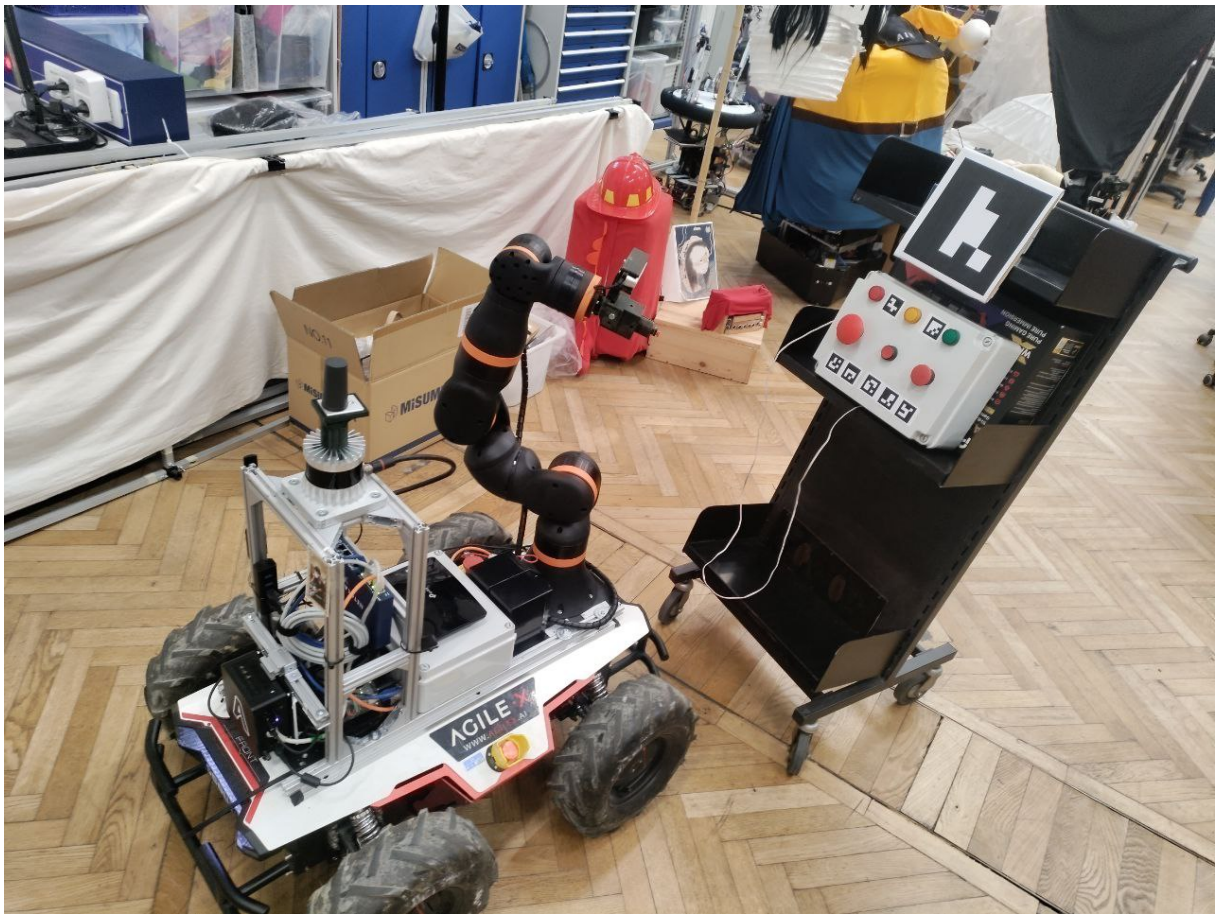


Figure 1.3: Mobile Button Presser Demo in execution

Despite the **parking algorithm** being effective in most cases, there were some cases where the robot parked too close or too far from the control panel. This was due to the imprecision of the localization algorithm and the local planner's inability to reach the exact parking pose. This problem resulted in a precision error in the final position of the robot, in the range of ±15cm from the desired parking pose. This problem was critical because an error of just a few centimeters could result in the robot being too close to the control panel to find it or interact with it, or too far to reach the buttons orthogonally. Since no ideal solution exists that can compensate for the localization and navigation errors, no further improvements were made to the parking algorithm.

1.3. Object Picking Demo

The "Object Picking" demo is a more complex demonstration that showcases the mobile manipulation robot's capabilities in picking objects from the environment autonomously. The demo consists of the robotic arm picking colored balls or apples from a fake plant tree,

Figure 1.4: Cobot pressing buttons on the control panel using the *MountV1* end effector

in an autonomous way. The objective is to demonstrate the robotic system's integration of multiple software components and the orchestration of the robotic arm and mobile base to interact and grasp objects of different shapes and sizes, using a soft robotic hand gripper. The soft gripper enables the robotic arm to grasp fragile objects without damaging them, and it is also able to grasp objects of different shapes and sizes, thanks to the flexibility and adaptability of the silicone fingers. For this demo, only *MountV2* is used, since it is the only one that supports the soft gripper.

The colored balls used in the demo are small plastic balls of different colors. The balls are used as a simple test ground for the grasping capabilities of the soft gripper, in fact, the balls are small enough to be easy to grasp and the plastic material enables high grip friction between the fingers and the ball. Instead, the fake plastic apples are a little more challenging to grasp, because of their irregular shape (not as ideal as the sphere). The fake apples are used to simulate a realistic and more complex scenario, where the objects to be picked are closer to objects appearing in real-world environments, such as fruits on trees.

The objective of the demo is to apply the mobile manipulation robot in an agricultural environment, which is often more challenging and irregular than in industrial environments. The demo is meant to be a proof of concept of the autonomous control of a robot to pick and place objects in a realistic environment, such as a plantation.

For this demo, three different versions were developed, each with different levels of complexity and challenges. The first version is used as a test for the algorithm for object pose estimation and the grasping capabilities of the soft gripper. The second version is used to test the robot's navigation and obstacle avoidance capabilities in conjunction with the object detection neural network and the perception algorithms. The third version is a simplified version of the second one because it is not a pick and place task with the targets in two different locations, but a picking task where the placing location is on the robot itself.

1.3.1. Plants, Colored Balls, and Apples Setup

The first setup for testing and demonstrating the demo uses a small fake plant tree with colored balls attached to it, as shown in figure 1.7. The second more realistic setup uses a big flat surface with fake apples placed on it, simulating a more realistic espalier apple tree, as shown in figure 1.5a. This sort of tree is used in agriculture to grow apples flatly and vertically, to save space, and to make the apples more accessible for picking. Figure 1.5b shows an espalier apple tree, grown in a yard. The apples are placed on the tree at different heights and distances from each other, to simulate a more realistic scenario where the apples are not all at the same height and distance from the robot. The apples are also placed in a way that the robot must move around the tree to reach all the apples, and this is meant to test the robot's navigation and obstacle avoidance capabilities.

The colored balls and apples are attached to the plant with a nylon string. At the extremity of the string, there is a small magnet that is attached to the string with hot glue. In the case of the apples, the magnet is attracted to another magnet placed on a screw that is screwed into the plastic apple. In the case of the colored balls, the magnet is attracted to another magnet glued onto the ball. To make this solution more robust, the balls have two magnets: one inside it, and one glued onto the external surface. Figure 1.6 shows the magnetic attachment of an apple to the nylon string. This solution is effective in attaching the objects to the plant and making them detachable, without the need to exert too much force to detach them so that the robotic arm can easily detach them without stressing the motors at all. The magnets are small and lightweight, and they do not affect the object's weight and shape. The nylon string is thin and transparent, and it

(a) Simulated espalier apple tree (b) Real espalier apple tree

Figure 1.5: Espalier apple trees

is not visible in the camera's field of view, so it does not affect the object's appearance.

1.3.2. Algorithm for Grasp Pose Estimation

Picking objects requires the software to know where the object is located in the environment and how to grasp it. Since explicit programming of how to grasp objects is quite difficult and not extensible to different objects, the software neglects the grasping strategy and focuses on the object's position estimation. The algorithm for computing the optimal grasping pose focuses solely on the object's barycenter, which is computed from the object's perceived pointcloud data. This simplification allows the software to be fast in computing the grasping pose, even though the generated grasping poses are not optimal for all objects. The algorithm is based on the assumption that the object can be approximated as a sphere and that getting the end effector sufficiently close to the object's surface is good enough to grasp it.

The algorithm used for computing the grasping pose from the object's barycenter is described in algorithm 1.1. The object's barycenter is estimated using algorithm 1.2, using the depth and RGB images. The algorithm generates a list of possible candidate grasping poses based on the object's barycenter and the object's known radius. For each candidate, it checks whether an inverse kinematic solution exists for the end effector at the candidate grasping pose. If a solution exists, the candidate grasping pose is added to the list of feasible grasping poses. The algorithm then uses a heuristic to choose which candidate grasping pose to use, based on the list of feasible grasping poses. It selects the candidate at one fourth of the list's size. The first candidate in the list will correspond to grasping poses of the object from the top, while the last candidate will correspond to

Figure 1.6: Magnetic attachment of an apple to the nylon string

grasping poses from the bottom. This heuristic is used to choose a grasping pose from a position closer to the top because it is usually easier for the robotic arm to reach it, compared to the ones closer to the bottom, due to the kinematic constraints of the robotic arm placed on the mobile robot base.

If no feasible grasping poses are computed, the algorithm returns an error message, and the robot does not attempt to grasp the object. Otherwise, the robot moves to the selected grasping pose and attempts to grasp the object.

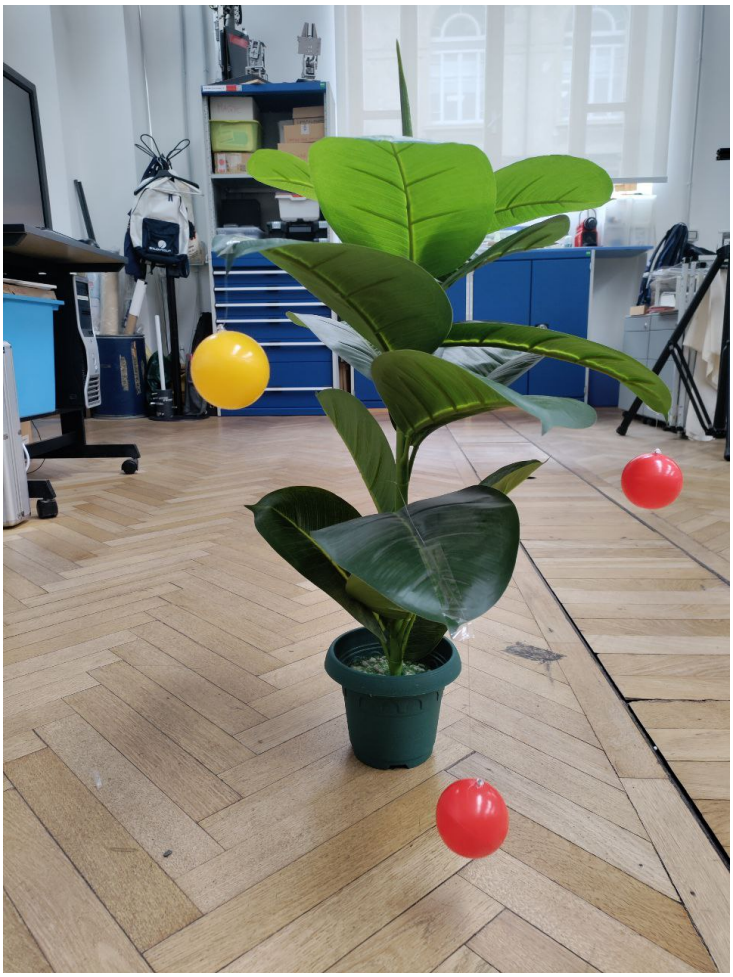


Figure 1.7: Simulated plant tree with colored balls

Algorithm 1.1 Grasp Pose Estimation from Object's Barycenter

Require: $p = (x, y, z)$ estimated object's barycenter in the camera frame

$C \leftarrow \emptyset$ ▷ Set of candidate grasping poses

$n_{candidates} \leftarrow 50$ ▷ Number of candidate grasping poses

$\theta_{min}, \theta_{max} \leftarrow -\pi, \pi/3$ ▷ Range of angles for the orientation of the end effector

$g \leftarrow 0.05$ ▷ Grasping distance from the object's barycenter in meters

transform p into the robot's base frame

for θ in linspace$(\theta_{min}, \theta_{max}, n_{candidates})$ **do**

  $v_c = \frac{(x,y,z)}{||(x,y,z)||}$ ▷ Vector from the base to the object's barycenter

  $v_l = -v_c \cdot g \cdot \cos(\theta)$ ▷ Longitudinal component v_l

  $p_v = \frac{(x,y,0)}{||(x,y)||}$

  $v_v = (v_c \times p_v) \times v_c$ ▷ Vertical component v_v

  $v_v = v_v \cdot g \cdot \sin(\theta)$

  $v_{grasp} = v_v + v_l + v_c$ ▷ Grasping vector v_{grasp}

  $v_{grasp,x} = \frac{-v_{grasp}}{||v_{grasp}||}$

  $plane_{xy} = (0, 0, 1)$

  $v_{grasp,y} = plane_{xy} \times v_{grasp,x}$

  $v_{grasp,z} = v_{grasp,x} \times v_{grasp,y}$

Considering the increased complexity of a soft gripper, compared to a more traditional rigid mechanical gripper, it becomes extremely difficult to simulate the grasping process accurately. The soft gripper's fingers are flexible and deformable, and they can adapt to the object's shape and size. This makes it difficult to simulate the fingers' behavior when in contact with the object, and how the fingers adapt to the object's shape. Since no accurate simulation of the soft gripper is available, the algorithm neglects the deformation of the fingers when actuating them, and it assumes that if the fingers are close enough to the object's surface, the object will be grasped. This is a strong assumption, but it is the only feasible solution to grasp objects with a soft gripper without simulating the complex behavior of the fingers.

An important note is that the algorithm 1.1 does not guarantee finding a feasible grasping pose for a certain object, even if it exists. The algorithm is based on a simplified model of the object and of the end effector and it restricts the range of search for a grasping pose to the vertical plane passing through the object's barycenter and the robotic arm's base. This simplification is necessary to make the algorithm fast and robust, but it also results in the impossibility of guaranteeing the successful computation of a feasible grasping pose for any object, so it comes at the expense of less reliability.

1.3.3. DemoV1 with Manual User Input

This first version of the demo (*DemoV1*) is a test for the grasping capabilities of the soft gripper and the autonomous control of the robotic arm, without any advanced perception algorithm implemented. This demo is also meant to be used when the robotic arm is fixed in a specific location. The demo consists of the robotic arm picking colored balls from the fake plant tree, with manual user input for selecting the target object to be picked. The user selects the target object by clicking on the object in the camera's field of view, and the robotic arm moves to the target object's position and grasps it. The user can select only one object at a time. The choice of the object is rather simple, as it does not use any neural network for detecting the objects. Instead, it relies on the user's input to select the pixel on the image corresponding to the center of the object to be picked.

When starting the demo, an image window appears on the screen, showing the camera's RGB image feed. The user can **click on the image** and the coordinates of the mouse click are recorded, and broadcast on a ROS2 topic. The algorithm for object position estimation computes the object's position in the camera frame using the pixel coordinates, the camera's intrinsic parameters, and the depth image from the infrared camera. The result is a (x, y, z) position vector of the visible point in the camera frame. This is the

point of the pointcloud on the surface of the object, which roughly corresponds to the object's visible surface center. By projecting a ray from the camera's optical sensor to the computed point, the algorithm computes the object's approximate barycenter in the camera frame. The barycenter is then transformed into the robot's base frame, and used as input to the grasping pose estimation algorithm to compute the target pose for the end effector required to position the robot where the object can be grasped.

Once the grasping pose is computed, the robot moves to the target pose and the end effector grasps the object. The robot then moves to a standard pre-defined position and drops the object, assuming that a basket is placed exactly underneath the robot's end effector in the dropping position. The robot then returns to the starting position and waits for the user to select the next object to be picked. Figure 1.8 shows this demo during its execution, using colored balls as objects to be grasped. The demo worked well both with the balls attached to the simulated tree and with the ball in the hands of human volunteers, as in 1.8.

1.3.4. DemoV2 with Object Detection Neural Network

In demo version 2 (*DemoV2*), the demo does not require any user input for selecting the object to be picked. Instead, it relies on a neural network for detecting the objects in the camera's field of view. The neural network is trained to detect the colored balls and the apples, and it outputs the object's bounding box and class label with the confidence score. The predicted bounding box is used as the starting point to obtain a pointcloud containing the points on the entire object's surface. The pointcloud is then used to compute the object's barycenter in the camera frame, and the grasping pose is computed as in the previous version of the demo. The algorithm makes the strong assumption that the object can be approximated as an ideal sphere of known radius. This algorithm is very robust for the colored balls, since they are spheres, and less robust and reliable for the apples, because of their irregular shape. The algorithm is able to compute the object's barycenter even from a small portion of the object's surface, and this is a strong point of the algorithm because it does not require the entire object to be visible in the camera's field of view.

The algorithm for the object's barycenter estimation from the surface pointcloud is the one described in 1.2. This algorithm uses a random sample consensus (*RANSAC*) algorithm to estimate the sphere's center.

The algorithm uses *MLESAC* (Maximum Likelihood Estimation Sample Consensus) to estimate the sphere's center from the segmented pointcloud. The function used for sphere

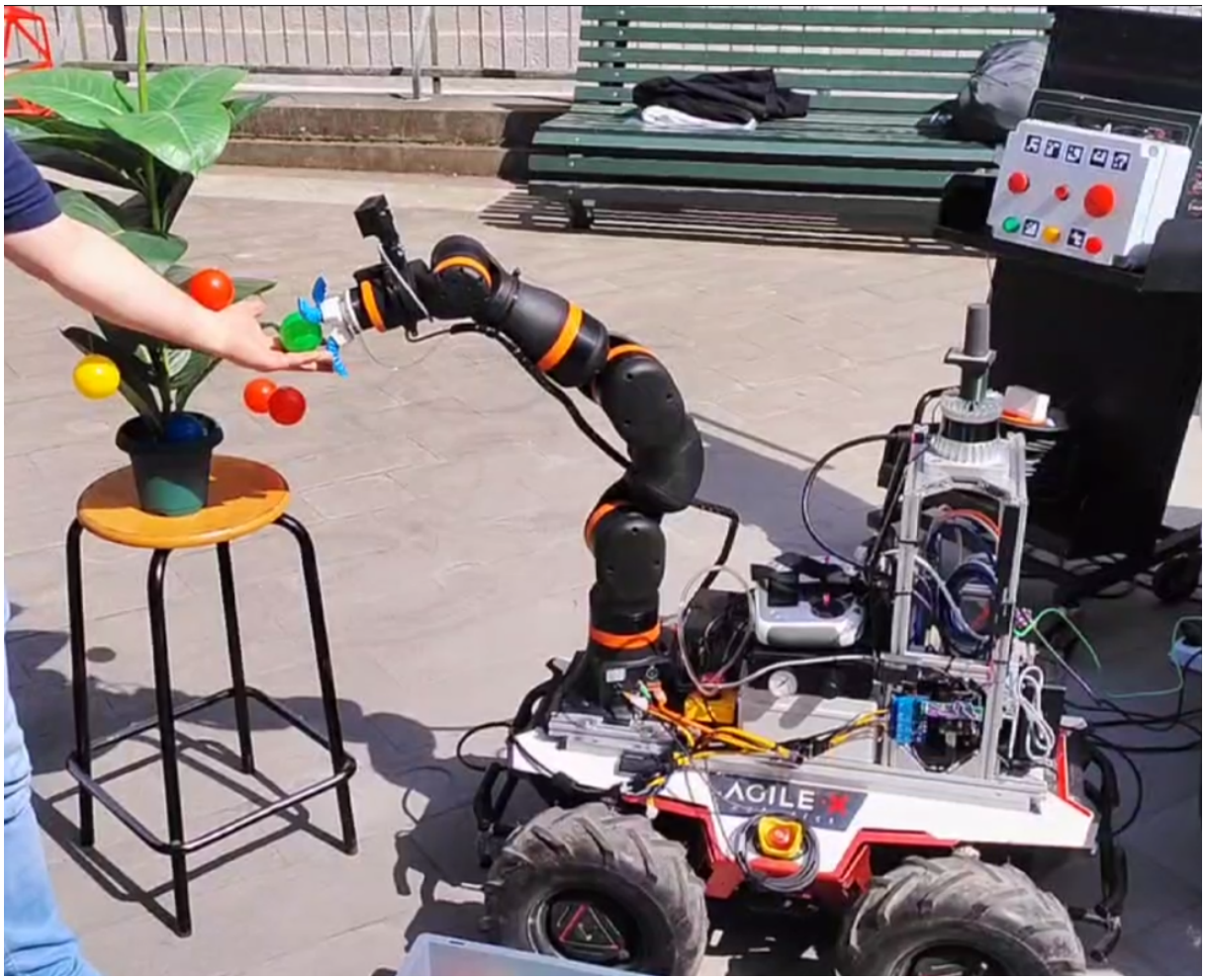


Figure 1.8: Cobot grasping a colored ball from the hand of a volunteer

fitting to a pointcloud is provided by *SACSegmentation* package from *Pointcloud Library*. This package provides a robust and fast implementation of the RANSAC algorithm for fitting geometric shapes to pointcloud data.

The assumption that the algorithm 1.2 makes is that the barycenter (center of mass) of the object is the same as the object's center. This is not always true, especially for irregularly shaped objects, but it is a good approximation for objects that are not too irregular. The algorithm computes the object's center and uses it as the barycenter, which is then used to compute the grasping pose.

Algorithm 1.2 Sphere Barycenter Estimation from Object Detection

Require: RGB image I, Depth image D
Require: predicted bounding box $B = (x, y, w, h)$, predicted class label $\hat{y}$
sphere radius range r_{min}, r_{max}, tolerance ϵ
$d_{max} \leftarrow 1.5$ ▷ maximum depth of useful points in meters
$I_{crop} \leftarrow I[y : y + h, x : x + w]$ ▷ Crop the image to the bounding box
$D_{crop} \leftarrow D[y : y + h, x : x + w]$ ▷ Crop the depth image to the bounding box
$P \leftarrow get\_pointcloud(D_{crop})$ ▷ Get the pointcloud from the depth image
filter pointcloud P by removing points with $z \geq d_{max}$
$colormask \leftarrow get\_colormask(\hat{y})$ ▷ Get the color mask based on predicted class label
$P_s \leftarrow \emptyset$ ▷ Object's surface segmented pointcloud
for each pixel $p \in I$ **do**
    if color of p is within $colormask$ **then**
        $P_s \leftarrow P_s \cup P(p)$ ▷ Apply color mask filter and add point to surface pointcloud
    end if
end for
$n_{min} \leftarrow 4$ ▷ minimum number of points passing through a unique sphere
for a fixed number of iterations **do** ▷ Random sample consensus algorithm
    $S \leftarrow$ random subset of n_{min} points from P_s
    $c, r \leftarrow$ center and radius of sphere fit to subset S
    if $r_{min} \leq r \leq r_{max}$ **then**
        Calculate *inliers*: points within a distance threshold ϵ of the sphere's surface
        Calculate *MLESAC* score based on the number of inliers and the residual error
        if $score > best\_score$ **then**
            Update $best\_model \leftarrow (c, r)$, $best\_score \leftarrow score$
        end if
    end if
end for
Refine $best\_model$ by fitting the *inliers* using least squares method
return $best\_model$

1.3.5. Mobile Fruit Picking Demos

The **complete demo** is based on DemoV2 and extended to include the mobile base's navigation and obstacle avoidance capabilities. The demo consists of the mobile manipulation robot navigating to the apple tree's location, detecting the apples using the neural network, and picking the apples in a predefined sequence. Figure 1.9 shows different

stages of the execution of this demo, where the robot is picking an apple from the tree. The robot then drops the apples into a basket. There are two versions of the complete demo:

1. **DemoV2 with pick and place**: the robot picks the apples from the tree and navigates to a predefined location to drop the apples in a basket. The basket is located in a position separate from the tree, and the robot must navigate to the basket's location to drop the apples. The robot must avoid obstacles in the environment while navigating to the basket's location. This version is more complex and challenging because the robot must navigate to two different locations. Once the mobile robot parks next to the basket's location, the robotic arm searches with the stereo camera an Aruco marker, which signals the position of the basket. The basket with the Aruco signal marker is shown in 1.10. Then an algorithm finds a feasible pose for the end effector such that it gets placed immediately above the basket, where it can release the grip on the grasped object and drop it, as shown in 1.11.

2. **DemoV2 with pick and place on robot**: the robot picks the apples from the tree and drops them on a basket that is positioned on the mobile robot base. The robot does not need to navigate to a separate location to drop the apples, but it is sufficient for the robotic arm to move to a predefined position so that the end effector is immediately above the basket. This version is simpler and less challenging than the first one, but it was implemented to have a demo that is faster to execute and more appealing to the public.

The **object picking routine**, explained in algorithm 1.3 is the routine that the robot follows to pick up objects from a plant tree. This algorithm is suitable for both apples and colored balls since they are both approximated as spheres. This routine defines the sequence of algorithms and actions to be executed by the robot to search for an object to pick, grasp, and drop in a basket, which can be located either on the robot itself or in a separate location far from the tree. The picking routine differs based on the version of the demo. When the robot must place the object in the basket positioned on the mobile robot itself, or just next to the robot itself, the robotic arm moves to the dropping pose and drops the object, as explained in the algorithm 1.3. Instead, when the robot must pick and place in separate locations, the robotic arm moves to its parking position and doesn't release the object until the mobile base has not reached the dropping location and the camera recognizes the basket using the Aruco marker.

Algorithm 1.3 **Object Picking Routine**

Require: Neural Network for object detection NN
Require: Depth Image feed D
max object distance $z_{max} = 1.5$ in meters
Generate a list of pose waypoints W
for each waypoint w in W **do**
    plan and execute trajectory from current pose to w
    check if there are objects detected by NN
    for each object o detected by NN **do**
        compute **segmented pointcloud** using algorithm 1.2
        compute centroid point P from segmented pointcloud
        compute distance z from centroid P
        get confidence score c of object o from vector C
        compute **priority score** $s = c \cdot 1/4 + (z_{max} - z)/z_{max} \cdot 3/4$
    end for
    Choose object o having highest score s
end for
Estimate **object's barycenter** using algorithm 1.2
Estimate **grasping pose** G using algorithm 1.1
if $\exists$ feasible pose G **then**
    compute pre-grasping pose G'
    plan and execute trajectory to G'
    open the gripper
    plan and execute linear trajectory from G' to G
    close the gripper to grasp the object
    plan and execute linear trajectory from G to G'
    move back to waypoint w
    move to dropping pose d
    open the gripper to drop the object
    turn off the gripper
else
    move to next waypoint
end if

The algorithm 1.3 is the main algorithm that the robot follows to pick objects from the tree. The algorithm is executed for each waypoint in the list of poses that the robot must follow to search for objects to pick. If there are multiple objects detected by the neural

network in a unique searching pose, the algorithm chooses the object with the highest priority score, which is computed based on the object's distance from the robot and the confidence score of the object's detection. The priority score is used to choose the object that is closest to the robot and has the highest confidence score. The algorithm then computes the object's barycenter and the grasping pose, and it executes the trajectory to grasp the object. The robot then moves to the dropping pose and drops the object into the basket.

This complete demo requires the **integration of multiple software components**, such as the neural network for object detection, the perception algorithms for computing the object's barycenter, the grasping pose estimation algorithm, the trajectory planning algorithms, and the navigation and obstacle avoidance algorithms. The demo is a complex orchestration of multiple algorithms wrapped in ROS2 action servers. The action servers are responsible for executing the actions and the algorithms required to pick the objects. The action client node is responsible for orchestrating the actions of the mobile base and the robotic arm, and for sending the goals to the action servers. The action client node also handles the sequence of actions in case of any failure of any software component and logs the feedback data received from the action servers. The ROS2 action servers used in the demo are:

- **Parking Action Server**: this server is the same as the one used for the other complete demo. The only difference with the other demo is that the location from which the robot must compute the parking position is not given by the result of another action, but is read from a configuration file. Both poses where to pick objects and the dropping location are hardcoded in a configuration file for simplicity and ease of use.
- **Picking Action Server**: this server is responsible for executing the first part of the object-picking routine. It is responsible for searching for objects to pick and checking whether there exists a feasible grasping pose for the objects in the camera frame. If it finds a feasible grasping pose for an object, it plans and executes the trajectories to grasp the object. The server returns the result of the planning and execution of the trajectories. It also provides logging feedback about the objects detected and the grasping poses computed.
- **Dropping Action Server**: this server is responsible for executing the second part of the object-picking routine. It is responsible for moving the robot to the dropping location and dropping the object into the basket. This server is called only if the dropping location is separate from the picking location. It searches for an Aruco

marker signaling the dropping location and computes a feasible pose for the end effector to drop the object. Then it moves to that pose and drops the object. The server returns the result of the execution of the trajectories and the dropping action. It also provides logging feedback about finding the Aruco signaling the dropping location.

1.3.6. Experimental Challenges

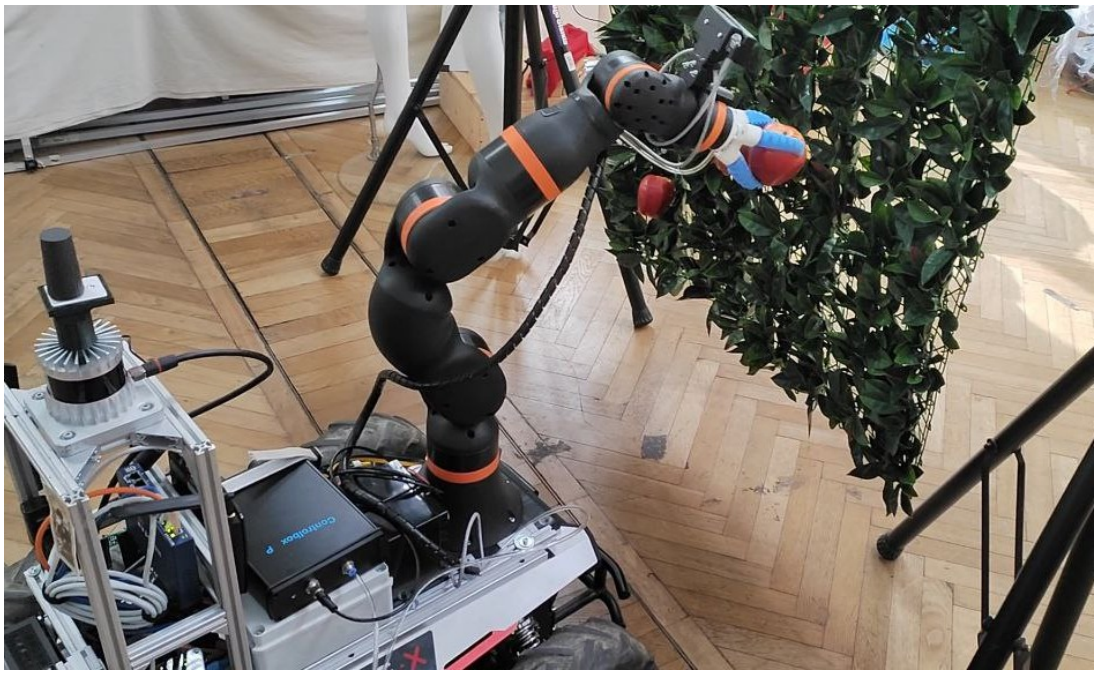
One of the issues faced during the development of the DemoV1 for ball picking was the **reflectivity** of the balls in the **infrared spectrum** in the presence of strong sunlight. The balls were reflecting the sunlight and the infrared camera was not able to compute the depth in the spots of the balls' surface with the highest reflectivity. This resulted in the balls' surface appearing as holes in the depth image. So if the user clicks exactly in the spot where the ball is very reflective, the depth value computed would be either wrong or infinite and would result in the algorithm crashing or computing an infeasible grasping pose. This effect was more pronounced with yellow balls. The immediate solution is to click in the least reflective spots, but this is a temporary workaround and not a definitive solution to the problem. This problem was first encountered during the presentation of the project at the Open Day of Politecnico di Milano, where the robot was tested outside, with direct sunlight, as shown in 1.8. On this occasion, some balls were not picked correctly due to the strong sunlight affecting the infrared sensors.

DemoV2 solved this problem because it relies on the entire pointcloud of the object's surface, and not just on a single point. This means that even if there is a hole in the pointcloud, the algorithm is still able to compute the object's barycenter correctly (even though likely less precisely). In the cases where the pointcloud presents points with wrong depth values, the algorithm 1.2 is still able to find the correct solution, as RANSAC is robust to outliers and noise in the data.

One problem faced during the development of the DemoV2 was the **high false positive rate** of the object detection neural network. The neural network was trained on a relatively small dataset of images, and it was not able to generalize well enough to new images. The neural network was detecting objects that were not present in the image, and this resulted in the algorithm computing grasping poses for objects that were not balls or apples. The obvious solution to the problem is to retrain the neural network on a larger dataset of images, but this is a time-consuming process and due to the time constraints of the project, there was not enough time to collect a larger dataset of images and manually label all of them.

Another problem faced during the development of the DemoV2 was the **slow performance of the neural network** on the CPU. The neural network was running on the CPU, and it was not able to process the images fast enough to provide real-time object detection. The neural network takes around 0.5 seconds to process a single image on the Intel NUC onboard computer, when other algorithms and ROS2 nodes are running in parallel, especially during the execution of the complete demos. This issue can be resolved by running the inference on the GPU, but it is currently not available on the robot. So the only feasible solution is to write code with multiple parallel threads that can handle and synchronize the data flow between the neural network and the other algorithms. This does not solve the problem but makes sure the software uses the predictions from the neural network as soon as they are available, coupled with the depth and RGB images taken at the moment of the prediction.

(a) The cobot opens the gripper in the proximity of the apple (at the computed grasping pose) to grasp it



(b) The cobot grasps the apple from the plant

Figure 1.9: Mobile Fruit Picking demo during execution

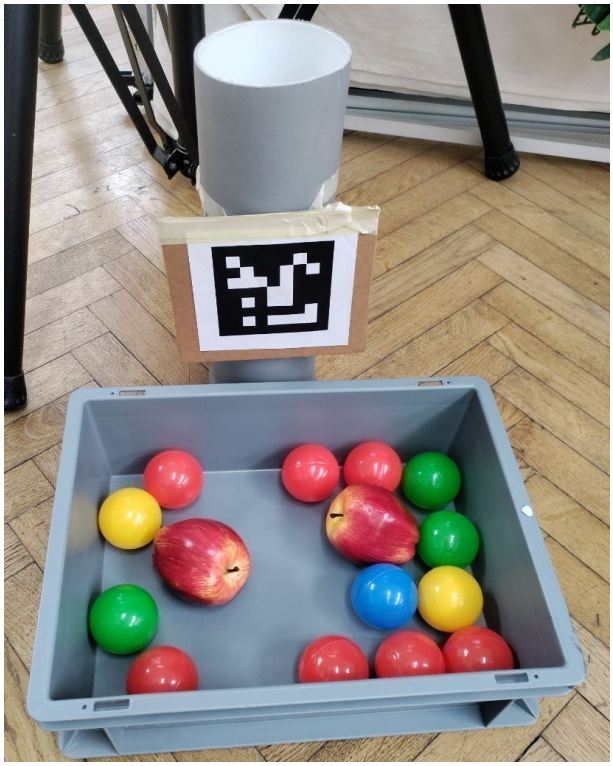


Figure 1.10: Basket at the dropping location

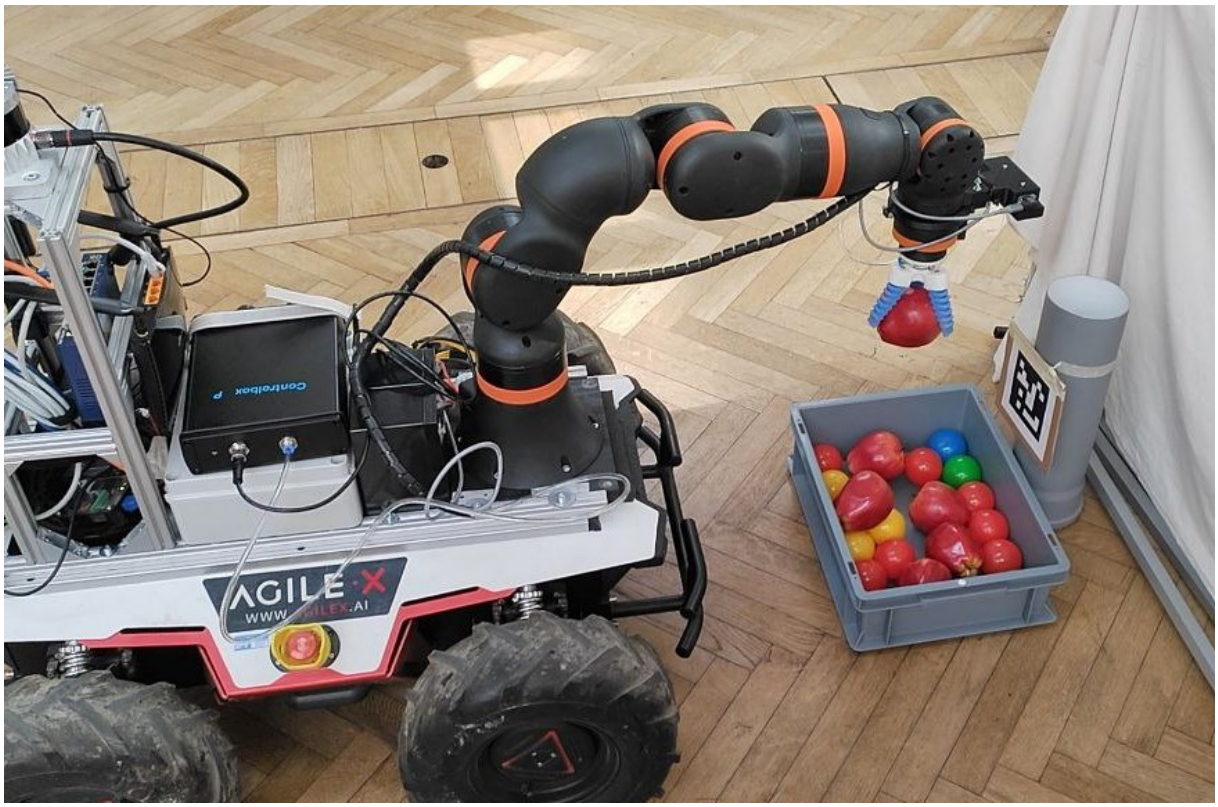


Figure 1.11: Cobot dropping the apple in the basket, at the dropping location, during the mobile fruit picking demo