Professors:

Luigi Palopoli, Niculae Sebe, Michele Focchi, Placido Falqueto

Project Report: Recognition and movement of Object with UR5 robot

Authors:

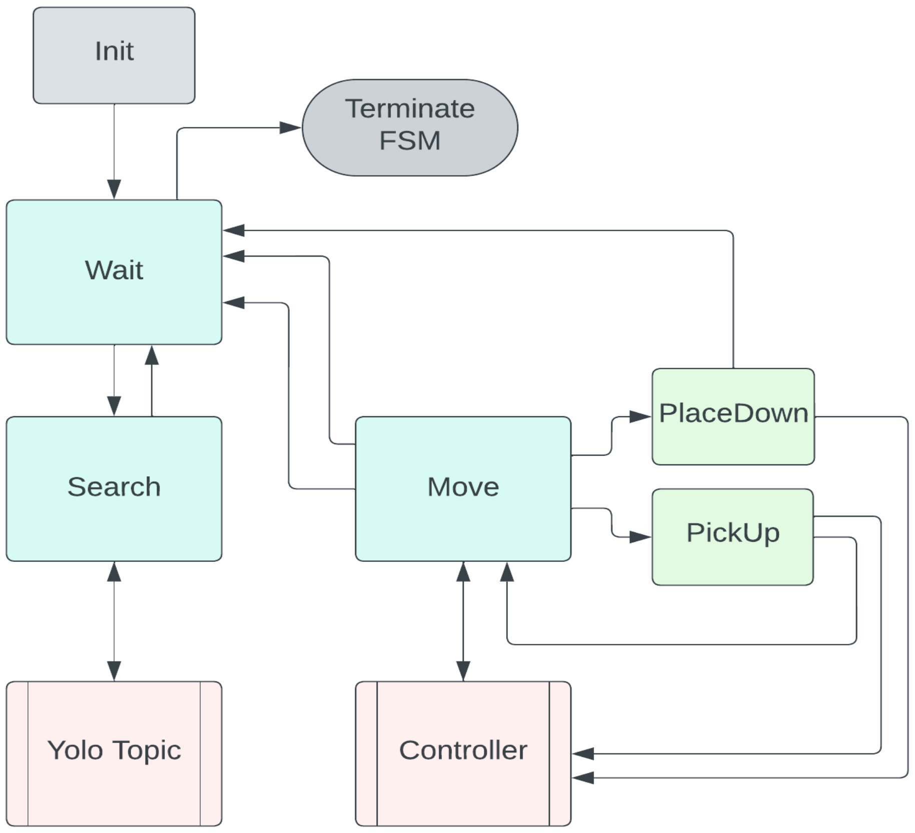
Emanuele Ricca, Jacopo Tomelleri, Emanuele Maccacaro

Introduction

The 'Fundamentals of Robotics' course presented our group with a series of assignments. Specifically, the tasks involved the identification and manipulation of blocks strategically positioned on a tabletop. The objective was to employ the UR5 robotic arm to not only detect these blocks but also execute precise movements to pick them up and relocate them to predefined areas on the table. The project is divided into four assignments of increasing difficulty.

FSM

The logic of the robot's movement is implemented through a Finite State Machine (FSM). This FSM is connected to a ROS node that works with Yolo, extracting information about the blocks on the table, and a robot controller implemented in C++, which handles trajectories and robot movements. Thanks to this architecture, the Finite State Machine cycles through various states, receiving information about the blocks, passing it to the movement controller, which then moves the robot to the required positions and opens and closes the gripper. The FSM repeats this simple cycle until all block movements are completed.



The FSM implements various states:

- Init: an initial state where the FSM is initialized, variables, ROS topics, and objects are set up, such as the movement controller and the connection to the YOLO node.
- Wait: a state used for state management; in this state, based on the conditions of the robot and the table, the machine selects a state among Search, Move, or program termination.
- Search: a state where the FSM wants information about the blocks, sends a signal to the -YOLO node, waits for a few seconds, then checks if information about the blocks to move and the coordinates of their silhouettes has been shared on the YOLO node's topic.
- Move: a state in which the FSM requests movements to the coordinates obtained from the -Search state to the movement controller. Depending on the FSM state, after the movement request, it returns to the Wait state in case of an error or goes to the PickUp or PlaceDown state, depending on whether the robot has already picked up the block.
- PickUp: a state where the gripper closure is requested, then it returns to the Move state.
- PlaceDown: a state where the gripper opening is requested, then it returns to the Wait state.

Controller

The robot controller is responsible for managing all robot movements, interfacing with the ROS node that controls the robot and requesting movements. Through various kinematics functions, it calculates joint values and trajectories necessary to reach various required positions, also handling the complex task of verifying all trajectories that different movement strategies implement.

To find the necessary trajectories to move to the required points, it uses various techniques. For simpler movements, it directly calculates the trajectory using inverse kinematics to determine joint values. For more elaborate movements, such as those near the wall or close to the table's back, it employs less direct and more sophisticated techniques:

- ResetMainJoint: Technique developed in order to avoid possible error when large rotations of the first joint are requested, an issue that can occur during this type of movement is the collision with the wall in the back of the table. With this technique the robot firstly moves to a defined configuration that prevents all collisions when rotations on the first joint, and then the robot spins on the first joint until above the requested position, then it can continue with the movement to the desired position.
- UpAndDown: This technique is used in order to fix movement that otherwise will be too low, and get invalidated. This technique keeps adding a new trajectory from the current position to a position above the current recursively until it’s high enough that the next movement won't cause troubles, once it has calculated the position above the current it calculates a move to the desired position, if it isn’t reachable it calculated before a move to the position above the desired and then to the desired position.

- NearAxis: This technique tries to fix some peculiar trajectories that can't be reached easily, it tries to divide the move in 2 moves, on the X axis and one of the Y axis, essentially it tries to move before to the nearest axis and then move on the other axis reaching the desired position.
- MoveThroughHoming: This technique can be considered the slow and safe option for when the other techniques aren't able to calculate a valid trajectory, this technique is based on safe check points that we called “Homing positions”. There are 6 “Homing positions”, those are generated from dividing the table in 6 parts then the center of each of those parts, at a fixed height, is considered a “Homing position”. When requesting a move, the technique tries to move the robot to nearest “Homing position” and the moves through the “Homing positions” in anticlockwise order until the reached “Homing position” is the one that is closer to the desired position, then calculates the trajectory to the requested position.

These techniques are necessary as various controls on trajectories are implemented to ensure that the robot does not collide with itself, does not hit the table, and does not collide with the blocks. For example, the robot cannot move below a certain height unless a specific flag is enabled to prevent collisions with blocks on the table.
We developed over 20 various checks for the trajectory in order to both optimize the moving space and to be always sure that the calculated trajectories are valid, giving us a solid method to always be able to move the robot even in the most difficult spots.

Optimizations

In order to optimize the time and to minimize the number of movements we developed a method to calculate trajectories that are more complex that simple trajectories from a point to a point, using the the movements techniques that we developed we are able to calculate the trajectory from a point to multiple points with different rotation matrices in a defined order even using multiple movement techniques in a recursive way generating a single trajectory than can include all the movement that one can want, all without ever moving the robot.
This enables us to be always able to tell if we reach a point before even requesting a move to the robot and always have an optimized trajectory to do so.
This is only possible due to our solid checks of the trajectory that discards all the invalid trajectory right away.
This may seem a little underwhelming at first but given the strict checks of the trajectory that we created being able to calculate the an elaborate trajectory without moving saves a ton of time, for example a movement to a point may not be possible given a peculiar position of the robot, which will require an additional movement before hand, such movement may result in another error resulting in a wasted movement. So such architecture prevents wasted movement and enables us to reiterate and combine all the possible movement tactics without never wasting movements.

In order to decrease the time even more we implemented the curve of a normal distribution with the aim of using it for the settling time of each step of our trajectory.
In other words, by dividing the settling time by the value of the normal distribution based on the number of the steps of the trajectory we became able to accelerate the movement of the

robot in the middle of the movement and decrease its velocity in the start and in the end for better precision.

Another issue we had to address was the fact that the each joint has a limited range of motion, for example the last joint can move between -2PI to 2PI, this cause us an issue due to the fact that we expected the joint to have no limit of movement and due to this false assumption we developed a optimization for the joint values, essentially trying to normalize them, when possible, with the movement that decreases that total distance, this cause issue because we were trying impossible values. For example when you have a joint that is a 2PI and you have to reach 0.1 rad it's way faster to go to (2PI + 0.1 rad), but such an angle isn't possible due to limitations of the joint.
So we adapted our joint optimization to work only when the calculated joint values are in reachable range, resulting in an optimization most of the time when possible.

Singularity Points

In the implementation of the control system, it became evident that not all positions were accessible when the gripper was directed downward for block grasping. Notable challenges were observed when the robot was positioned at a considerable distance and around the shoulder joint.

To tackle the issue of inaccessible positions, we explored the implementation of a side picking approach. This strategy involves attempting to grasp the block from one of its sides rather than from above. The primary motivation behind this alternative approach is to mitigate the impact of singularities at problematic points, specifically those occurring in the vicinity of the shoulder joint.

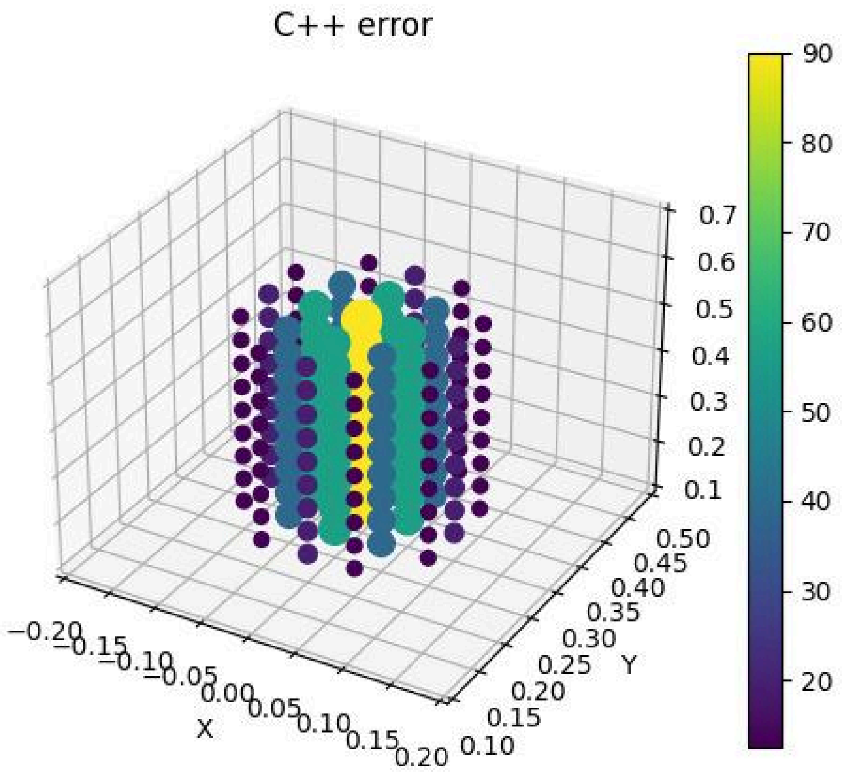
While side picking presented a promising solution, it introduced new challenges that required careful consideration during implementation:

- Determination of New Coordinates: Calculating the new coordinates for block grasping became a critical aspect of the implementation. Since each block might have a different orientation and distance from the robot. Based on the angle of the block, the new coordinates are calculated by adding the sine or cosine values of the angle to the corresponding axes, multiplied by a fixed dimension of 0.01.
- Rotation Matrix Adjustment: The introduction of side picking necessitated adjustments to the gripper's rotation matrix. This adjustment was founded on the angular orientation of the block. Achieving a proper alignment for successful grasping demanded a detailed analysis and calibration.

Furthermore, during the implementation of the side pick, we found it necessary to introduce additional control mechanisms. This adjustment was crucial to facilitate smooth robot movements, mitigating the risk of self-interference and ensuring overall operational fluidity.

Currently, the side pick functionality is not operational, as we have encountered certain issues that remain unresolved.

The graph below represents the inaccessible point without the implementation of side pick.



Computer Vision

The block recognition component was integrated into the project with the Computer Vision package. The package contains the script for block recognition, developed in Python, and the node for communication with the robot, developed in C++.

The script in python deals with analyzing the images received from the zed camera. For each image received, the blocks are searched using Yolov8 and the relevant bounding boxes are drawn. This information is then published and forwarded to the 'yoloNode' node via the topic /computer_vision/bounding_box, which will calculate the position and orientation of the recognised blocks. Subsequently, the node will communicate all the necessary information to the controller in order to reach and bring the various recognised blocks to their destination.

For block recognition, one of the models provided by Yolov8 was used, trained on a dataset generated by us. The dataset was generated with a python script integrated into Blender, where the robot's working environment was simulated and enriched with the blocks to be recognised. The blocks were generated with a random distribution of colour, position and rotation in order to guarantee a sufficient variety of images for training the model.
The entire dataset consists of 8,000 images following a proportion of 70 per cent for training and 20 per cent for testing and 10 per cent for validation.

The vision process is regulated by the controller, which communicates on the topic */computer_vision/permission* to give permission to start the block recognition phase.
During the initialisation phase of the robot, permission is given to recognise blocks to ensure that Yolov8 starts up correctly. Once the initialisation of the robot has been completed, the controller grants the block recognition permission again, in this case to actually receive information about the recognised blocks. The block recognition script, once it has gained permission, starts to recognise the blocks in the image received from the topic */ur5/zed_node/left_raw/image_raw_color*.

The bounding box is drawn around the recognised blocks, and a message containing this information is published on the topic /computer_vision/bounding_box, which is then received by the 'yoloNode', which in turn forwards it to the controller.
The drawn bounding box is narrowed down with respect to the recognised block, in order to ensure greater accuracy in the orientation evaluation phase.

The part developed in c++ is responsible for receiving information about the recognised bounding boxes and communicating with the controller.
Once the bounding boxes have been received, the 'yoloNode' decides whether or not to save the information received. The decision is based on whether the recognised block has already been recognised before or whether the block is too close to where it should be taken. With the bounding boxes saved, these will then be used to create the message *computer_vision::Instruction*, containing the position and orientation of the recognised block and the destination of the block. The destination of the block is setted before the execution of the program, and depends on the type of the block.

The position of the block is calculated using the intersection of the bounding box and the block itself. Using the */ur5/locosim/pointcloud* service, pointClouds are requested along the left side, from top to bottom, and the right side, from bottom to top, of the bounding box.
If the recognition is correct, the requested points will be the two opposite ends of the block and calculating the midpoint between the two points gives the central position of the block. However approximate, the calculated position is sufficient to ensure correct detection of the block.

The angle of the block is also calculated in a similar manner. PointClouds are requested along the bottom side, from left to right, and along the left and right sides of the bounding box. On the bottom side, it is checked that the point is chosen along the edge of the block. One can ensure that the point belongs to the edge of the block by checking its x-coordinate, as this will be the minimum with respect to the other points along the bottom side.
Having obtained these three points, one can calculate the distances between the x and y coordinates of the lateral points with respect to the one on the lower side, and with the use of the arcotangent, the angle of the block can be computed.
This calculation is performed for both the left and right sides, but only the one obtained using the larger distances will be returned as the angle of the block.
It is important to note that if the angle to be returned is the left angle, it will be necessary to subtract 180 degrees from this to obtain the correct angle for the robot.

Before returning the angle of the block, a final check is performed to assess whether the block is oriented vertically, i.e. 90 degrees. For this check, the distance between the points on the 2 lateral sides of the block is calculated and the distances to the x-coordinates of each point are checked. If these values are below a certain threshold, then the block is oriented vertically and the angle returned will be 90 degrees. Finally, a vector containing the positions and angles of the recognised blocks is published on the topic */computer_vision/Instructions*.

Results

During development, we encountered numerous problems due to the precision we wanted to maintain during movements. We tried to create a design that could make the best use of the space it had and this took a long time to test properly, so we only managed to complete 2 of the 4 required assignments. During the course of the assignments, the following KPIs were measured:

KPI 1-1	5.5 seconds
KPI 1-2	28.92 seconds
KPI 2-1	124.756 seconds (4 blocks)

The KPI 1-1 will always take at least 5 seconds, to ensure that the detection scripts initialize correctly.