Robotics Project

Benassi Alessandro, Calvo Daniele, Cristoforetti Niccolò, Gottardelli Matteo

February 3, 2025

Contents

1 Introduction

This paper describes the methodology and the results achieved by a ros2 based program that moves an UR5 robotic arm with a gripper, in order to grab and move some different randomly spawned blocks from a random start position to a desired position. The aim of this project is to provide a functional and correct implementation to be able to perform the operation without encountering singularities or errors.
Different instrumentation was used to develop the project, starting from the blocks to move, and the camera, which allows us to scan an image on which we can perform detection and determine the position and the orientation of the objects. The last fundamental component is the UR5 robotic arm, a lightweight arm designed for tasks that require flexibility; it has 6 degrees of freedom and is composed by revolute joints: a base, a shoulder, an elbow and a spherical wrist.
Regarding the development environment, different tools were used. The first is DockerDesktop which allows us to run the image provided by the professor. This last one is an Ubuntu virtual machine with ros2 on it, that allows us to use nodes and services for communication between the different parts. The Eigen and yolov5-pip libraries are used and the necessary Python modules are downloaded when starting the program. There is also the Roboflow application which is used to generate a custom dataset and then Google Colab to obtain trained weights for the recognition part.
The algorithm developed is divided into two main parts: vision and manipulation. The first is responsible of scanning the camera to extract an image, process it to localize all the blocks using Yolov5 and returning the coordinates of the bounding boxes to calculate the relative positions. The second controls the movement of the arm to reach the given position, grab the object and put it in the designed position.

2 Vision

Vision plays a critical role in robotics by allowing the robot to perceive and interpret its environment. The three main challenges for this project are: spawning the blocks, recognizing them, and calculating their position in the space.

2.1 Block spawn

As mentioned in 1 the absolute position is the one of the camera, given by the following parameters: x=-0.5 , y=0.5, z=1.2, R=0.0, P=0.4, Y=-0.06, the first three are the coordinates, the others are the rotation values. At this point we had the table with the arm attached, and we wanted to spawn the blocks in the right part, which is closer to the camera. The blocks are spawned directly in the sim.launch.py so they are spawned right after the creation of the environment with the table and the arm. The minimum number of blocks is 1 and it goes up to 4 and this parameter is inserted by the user in the command line. The spawn area of the blocks has a fixed height, which is based on the one of the table (z=0.88) and its width in the x and y axis is respectively (0.05, 0.405) and (0.2, 0.58). All the generated blocks are different in type and color. The type is decided randomly between a set of nine, taken from the ones in the folder published on moodle, which were eleven, but we decided to remove two of them that are larger than the single block to avoid issues with the gripper, that may not open enough to get around the block, since the script we are in posses only allows us to open and close and not to

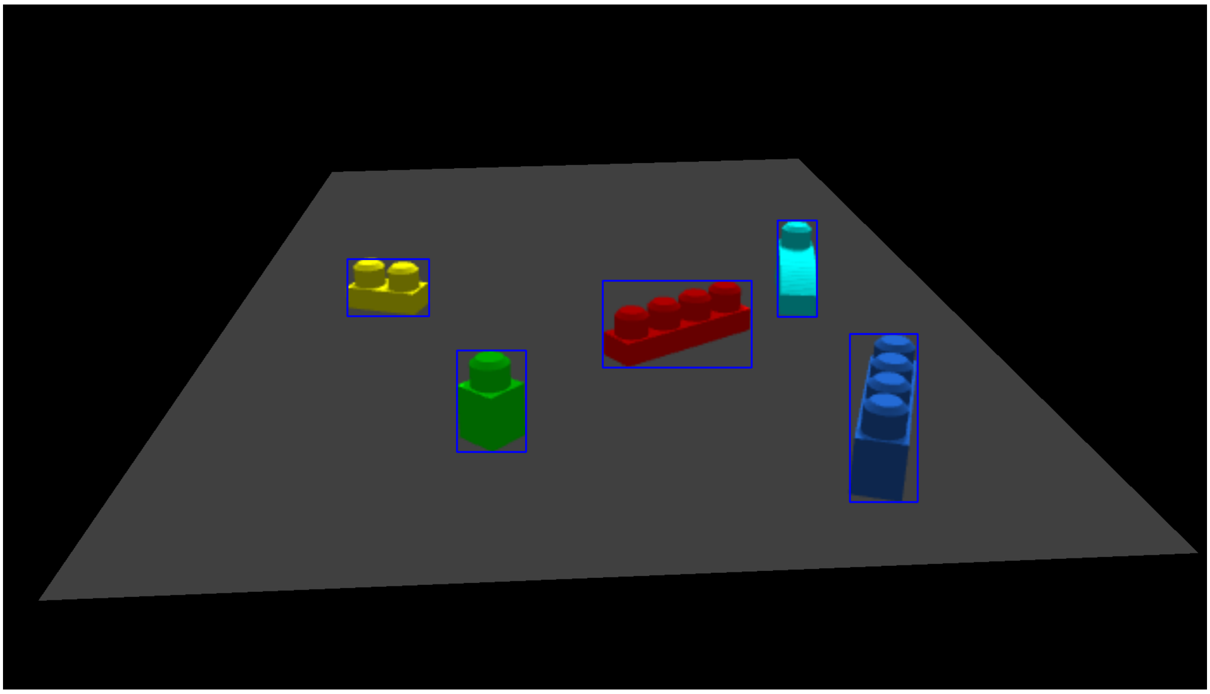

Figure 1: Blocks detection with bounding-boxes

decide a width. The color is also chosen in a random way between a selected set and, regarding the position, it is generated randomly and has to stay within the limits of the defined spawn area and be collision free, so there is a minimum distance between each object, otherwise we regenerate the overlapping one. The orientation is random and is only about the roll(z axis), because the project does not consider the blocks in a different position, like upside down, for a matter of complications in detection and grabbing. The objects are generated using a .urdf.xacro file, which is then passed to .urdf format and at the end in .sdf to be displayable on Gazebo.

2.2 Processing images

To perform object recognition, the first thing to do is scan an image from the camera. This is done by the node image_acquiring, which subscribes to the topic that publishes *sensor_msgs::msg::Image* data and, after gathering the content, it is processed with opencv2 library functions to print it on a generated image file in .png format. It is saved in a precise path, that will be used later in the program.
It is saved to a precise path that is then used by visual.py to retrieve the image and perform three different transformations on it. The first is the image resize, the second is the background removal, with respect to the resized image, to have only the table and the blocks. The last image is used for the detection process and it is generated with no background and with 25% of the table removed to avoid detecting the previously moved blocks, since recognition is performed at every complete movement of the arm.

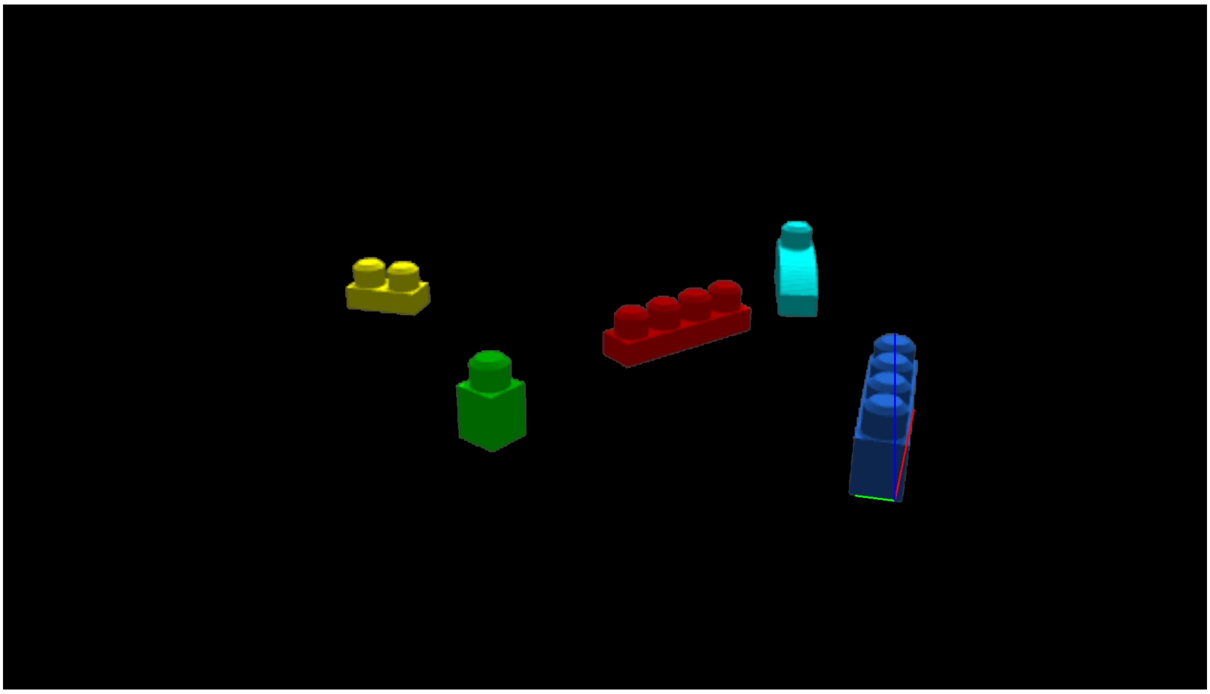

Figure 2: Axis for the block to determine the orientation

2.3 Detection

For this part we used Roboflow, which allowed us to create a dataset containing the images provided by the professor on moodle, labeled with the classess of the different blocks so we had a quite large number of labelled images per class. We ended up having eleven classes but we decided to use only 9 of them, see 2.1. A percentage of images was destined to training, and a smaller one to testing, this was done automatically by the program. The dataset was then downloaded to perform the training part on Google Colab and at the end of this process the result was blocktrain.pt, which is the file containing the trained weights to be used to perform the recognition with Yolov5. For the recognition the yolov5-pip repository is downloaded from github (link: `https://github.com/fcakyon/yolov5-pip.git`).

When passing an image we were interested in knowing the coordinates of the bounding box, the confidence, and the class. With the first one we are able to calculate the position of the block on the table and with the second one we can understand whether it is an erroneous identification. To obtain this information, we developed a Python script detection.py that creates the service: yolo_bounding_box_service, which takes in input the path of the image from the camera, taken as explained in 2.2, and returns an array of custom msg type called Boundstruct, that contains the class identifier, the confidence and the four values of the coordinates of the top left and bottom right points of the bounding box. The boxes are also printed on the image Yolo1.png in the vision_ws/generated folder, see 1.

2.4 Object elaboration

The values returned in 2.3 and 2.2, in particular the coordinates of the bounding box and the image path are essential to determine the position of the object with respect to the Gazebo world frame. The file process.py initializes the service convert_coordinates that takes in input the previously listed parameters. It finds the center of the bounding box and determines the position in the 3D Gazebo world frame, and it also computes the rotation and prints the rotated axis with respect to the lower angle on a .png image named Obj1.png in the vision_ws/generated folder, see 2 . At the end the returned values are: the x, y and z coordinates of the center, the rotation and a boolean to indicate the success of the operation. These parameters are then used by the manipulation part to perform the movement of the arm.
This whole process, except 2.1, is performed every time a block is moved, to avoid detection errors. The program will stop when all the blocks have been moved successfully and so the detection process gives no result.

3 Manipulation

The manipulation system allows the robot to interact with the surrounding environment and interact with objects through its end effector, which is a gripper in this case. The objective of this part is to develop an algorithm that, given a position, can reach the object, grab it and take it to its assigned storage location.

3.1 Direct and Inverse kinematics

Two fundamental instruments for performing the task of moving the arm are direct and inverse kinematics. The first one allows us to compute the position and the orientation of the end effector given a set of Denavit-Hartenberg(DH) parameters, which are: theta, alpha, d and a. For each joint of the arm the matrix with the position and the rotation is computed with the homogeneous transform, and then it is multiplied following the direct kinematics theory to obtain the T06 matrix. The function returns the position values as a vector and the rotation matrix.
On the other hand, inverse kinematics is the opposite process, by starting from a desired position and orientation of the end effector the aim is to compute the values of the DH parameters to reach it.

3.2 Trajectory

To compute the trajectory, the first thing to do is to check whether the destination is reachable. After that, there are 3 different ways to proceed:

- Operation space with the interpolating points: the trajectory is calculated specifying a series of waypoints, which are then lined up with an algorithm to obtain a linear path.

- differential kinematics: it is based on the relation between the joints velocity and the velocity of the end effector. The Jacobian matrix is used to map from the joint space to the operational space and then the pseudoinverse is used to compute the joint configurations over time. This is computationally expensive and can lead to problems when singularities are encountered, moreover we wanted to decide the velocity of the movement arbitrarily so this method does not fit with our needs because this last is controlled by the robot's security systems, which limit it.

- Operation space with positions: it is similar to the first one, but instead of interpolating the points, the desired positions are directly specified in the operational space and it is verified that the end effector reaches them. The direct and inverse kinematics are used to compute the point to reach and the configurations to adopt in order to do it.

We choose this last implementation and to guarantee a smoother trajectory a quintic spline is added. The choice to use this last one over the other implementations is because a tolerance related problem. Since this last was set by hand in the simulation, this gives limits in terms of speed, so this may cause problems if we use differential kinematics and we may end up in wrong positions. It is better to work in the joint space to handle this limitations better. The spline is added because it gives more smoothness to the movement even if it adds some inaccuracy.

3.3 Operational space using polynomials

The function p2pMotionPlan() computes the path, taking in input: the initial configuration of the robot, the position to reach with its orientation, the time of the movement and a matrix where to store the values. This computation is done using the inverse kinematics explained in 3.1; if no configuration is found, it means that there are no valid solutions for the requested configuration. Fifth grade polynomials are used to obtain a smooth trajectory and avoid sudden accelerations or oscillations. The matrix containing the move parameters(destination and rotation) is directly modified by reference and the function returns a boolean to tell if the operation is successful or not.

3.4 Constraints(reachable position, angles and space limits)

To avoid the arm to collide with itself, the table or the blocks, some constraints have been defined. The first one are for singularities and are about the asin and acos functions. For the asin function values are adjusted so that if $< -1, -\pi/2$ is returned, while if $> 1, \pi/2$ is returned, otherwise the values remain unchanged. For the acos function the adjustments are: if $< -1, \pi$ is returned, if $> 1, 0$ is returned. Also checkAngles() function is implemented; it controls that the shoulder and the elbow movements do not exceed the limits and generate singularities. Another function is implemented to adjust all the float values for the positions to 0 if they are close to this value, this is done to avoid strange behaviors. In addition, control over position is fundamental to avoid movements outside of the table area. This consists in limits for the upper part where the robot is attached, for the lower part with a security position a bit above the table height to avoid collision with the blocks or the table. For the lateral part the limit is the width of the table and then there is another constraint regarding the back part, because there is a panel attached to the table.

3.5 Logic motion

The process of the single block motion is the following: the arm starts in the home position, it opens the gripper and goes down over the block in a security position, 0.4

above the table, then it lowers down and closes the fingers to grab the object, after that it stops in the security position. Then it moves to the designed position, it goes down to the security position and then lowers on the table surface and releases the block. Finally, it goes up to the security position over the object and then it returns to the home configuration before restarting with the visualization part and iterating until all the objects are moved.

4 Challenges

- The first challenge was understanding the orientation of the block based on the image generated from the camera, which contains another factor to consider that is perspective. This was solved using Cartesian axes.
- A bug in the Gazebo management was encountered: if a goal is sent via message, the client successfully sends it, the server receives and accepts it, but it gets stuck indefinitely. This was solved with a bash line called from C++, but the line is quite large in size, and during various tests, we encountered errors not so much due to the algorithm of manipulation, but more because the computed trajectories are partially resolved, as if they are truncated. We encountered a malfunctioning of the complete algorithm that could also be attributed to this factor even if it has been solved. The problem is that the arm gets stuck because it cannot perform the path it wants due to the constraints we put as explained in the next point.
- Managing the robot's movement by avoiding singularities and obstacles such as the table or columns behind was done by defining the operational space. These limits lead to drawbacks in the inverse kinematics, because, due to these constraints, sometimes the system could not perform these trajectories.
- Managing the various communications between nodes and packages, including the creation of multiple blocks at the same time was done using messages and services by passing and taking the desired information.

5 Conclusion

In conclusion, we were not able to provide a working implementation of the project. Since everything is implemented separately, the vision part works fine and has a good expandability since a different number of blocks can be generated and detected and this can be modified in future implementations. For the manipulation part the issue with the movement of the arm that gets stuck described in 4, is not solved. This may be because of a problem of the server as explained before, but since we don't know for sure the idea was to try to do another implementation with differential kinematics but the time did not allow us to try it. This can be a challenge for the future.