University of Trento

Project Report

— Fundaments of Robotics —

Giorgia Lorengo, Nicolò Marconi and Nizar Nadif

Abstract

The project consists of the implementation of a system to recognise the position of blocks placed on a table and move them using an anthropomorphic arm manipulator.

In order to achieve that goal we splitted the problem in three main modules:

1. *Vision Module* : which detects blocks using a **YoloV8 model**, uses a **ZED Camera** to have a representation of the table and where the blocks are placed, and combines these information to compute the positions and orientations of each block [made by Giorgia Lorengo, Nizar Nadif];
2. *Movement and High-Level Planning Module* : which computes the movement path that the **UR5 manipulator** has to follow in order to grab the blocks and leave them in a predefined position on the table [made by Nicolò Marconi];
3. *Client Module* : whose task is to **manage communication** between the vision module, the motion module and the robot using **ROS**. This is done by means of a cascade communication: the client module waits until the vision has returned information about the blocks that have been recognised, sends the information of one block at a time to the movement module, and waits for the return of the path of the movement to be followed by the robot, and finally sends this data to the robot [made by Nicolò Marconi, Nizar Nadif].

We use two `.srv` file to enable **request/response communication** between the `vision-client` and `movement-client` nodes:

- `GetBlocks.srv`:
  - `res`:
    - `geometry_msgs/Pose[] poses` : pose of the blocks detected
    - `string[] blocks_id` : all ids of the blocks detected
    - `int8 n_blocks` : number of blocks detected
    - `bool finished` : check if vision has finished to detect all blocks
- `MovementHandler.srv`:
  - `req`:
    - `geometry_msgs/Pose pose` : pose of the block to be moved
    - `string block_id` : id of the block to be moved
  - `res`:
    - `Path path` : path matrix of the joints configurations that the robot has to follow

Contents

1 Perception

The *vision module* of the project consists in **detecting the blocks** on the working table and **estimating their position**. This was possible using the data obtained by a ZED Camera, placed above the working area, which allowed us to have both a 2D picture of the area as well as a 3D point cloud representation.

For each block on the working table, the vision module **returns to the manipulator**:

- the *type* of the block (e.g. X1-Y1-Z1);
- the *(x,y,z) coordinates* of its barycenter;
- its *yaw rotation*, where `yaw=0` means that the longest side of the block is parallel to the x axis.

At a high level, our approach involves using a **neural network** to detect blocks in a 2D image while leveraging **point cloud** data to determine each block's orientation and center position. We observed that both the model and the point cloud can exhibit uncertainty in certain situations. For instance, the model might predict an object with low confidence, or the point cloud might miss parts of a block if another object is obstructing the view. To address these issues, we developed an **algorithm to identify blocks that are most likely "correctly detected and located"**. These confirmed blocks are then sent to the *Movement module*.

As we will see, the vision process can be **executed multiple times** until the working area is completely cleared.

Below, we provide a more detailed description of the various components that constitute the vision module, along with the rationale behind certain design choices. It is important to note that we threaded the object detection component to optimize the program's execution. This was necessary due to the fact that the point cloud processing requires, on average, twice the amount of time as object detection.

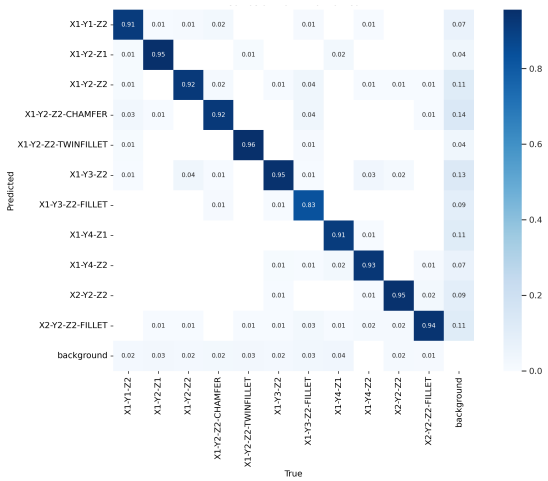
1.1 Object Detection with YoloV8

We needed to find a way to develop a neural network to detect and classify the blocks; a viable option we found was **fine-tuning a YoloV8 model**, which allowed us to avoid creating a CNN from scratch.

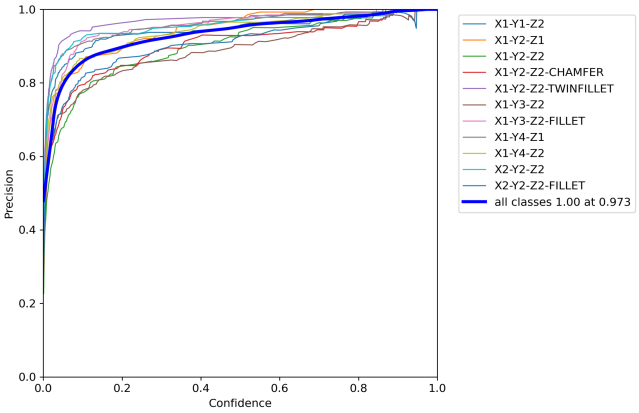
The dataset was created using **Roboflow** by extracting the images and relevant information from the provided file. To do so, we developed a simple Python script to generate the annotation files required by Roboflow. The final dataset consisted of **1,500 images**, divided into a training set (70% - 1,050 images), a validation set (20% - 300 images), and a testing set (10% - 150 images).

The model was fine-tuned in **Google Colab** for approximately **1,000 epochs** (976 to be exact), after which we achieved good results. We stopped the training at this point to avoid the risk of overfitting.

[1]Since the blocks are small, we approximated the barycenter to the coordinates of the center of the block, which is easier to compute.

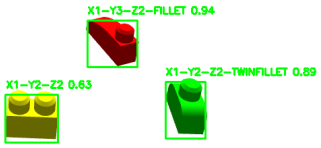


[Figure] Confusion Matrix Normalized



[Figure] Precision Confidence Curve

The neural network, saved locally as a .pt file, is used to make predictions on the 2D images captured by the camera. The predictions are provided as a list of objects, sorted by confidence in descending order. For each object, we receive detailed information, including the coordinates (in pixels) of the **bounding box vertices**, the **predicted class**, and the **confidence score** of the prediction.



From this point forward, we will refer to the blocks detected by the model as ***yolo blocks***.

1.2 Pose Detection with Point Cloud Data

We use the point cloud generated by the **ZED Camera** to determine the object's location in the world frame. Given the high density of the point cloud, we applied several **filters** to reduce the number of points we needed to process. Specifically, we began by extracting only the points corresponding to the working zone. This was done by projecting into 3D only the pixels within a rectangular region (whose coordinates were determined empirically) that represented the table zone in the 2D image.

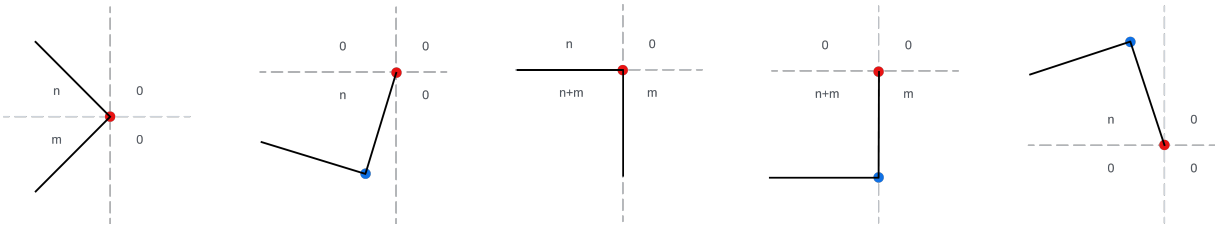
Since the camera that generates the point cloud is not aligned with the world frame, we applied a **transformation matrix** to convert all the 3D points into the world frame.

To process the point cloud, we horizontally "sliced" it at a specific z-value (`z=0.875`) to obtain a **cross-section containing all the blocks**; to do this efficiently, we created a **dictionary** with the z-values as keys. If the dictionary does not include the key value 0.875, it indicates that the **working table is empty**, and the vision system returns a boolean `finished=True` to the manipulator to **signal the end of the task**.

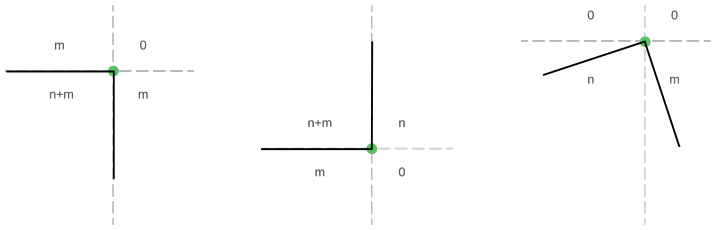
We then applied the **DBSCAN clustering algorithm** to group the points into distinct objects. For each block, we extracted **three points corresponding to the vertices** visible to the Zed Camera.

In order to do so, we firstly set the rightmost point in the cluster as the vertex of the block. The **cluster gets divided in quadrants around the vertex**, and based on the cardinality of points in each quadrant we set the **middle vertex**. The middle vertex may be considered the final vertex, as it should be in the middle between the two touching sides. By computing the quadrants of the new vertex its possible to deduce, after comparing the values obtained,

where are the sides and how to find them.

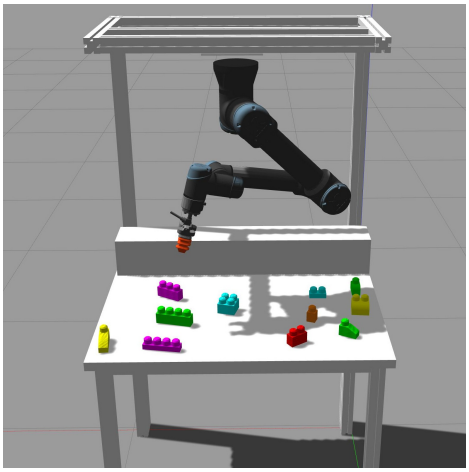


[Figure] Different cases given the rightmost vertex

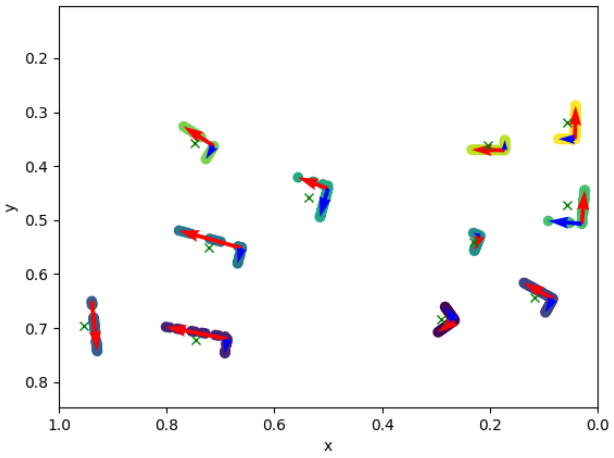


[Figure] Quadrants after the vertex is computed

We computed the **(x,y) coordinates of the center** simply as the middle point of the bisector of the rectangle base of the block; having only two sides - and being point A and point B the points, seen from the point cloud, at the end of each side - the bisector is simply the line connecting A to B. The **orientation**, instead, has been determined by computing the arctangent of the longest side w.r.t. the x axis. Moreover, each block has an **"accuracy" parameter** which reflects the quality of its point cloud measurement; this is computed as *how close the angle between the two vertices is to 90 degrees.*



[Figure] Blocks from Gazebo



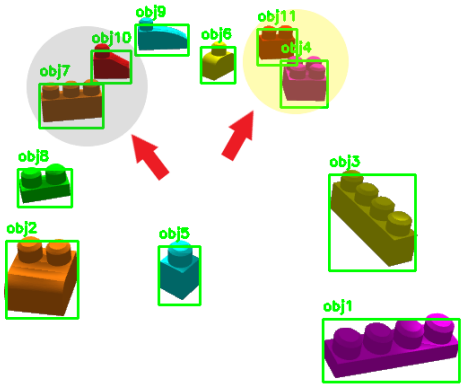
[Figure] Blocks scanned by Point Cloud

From this point forward, we will refer to the blocks detected by the point cloud as ***zed blocks***.

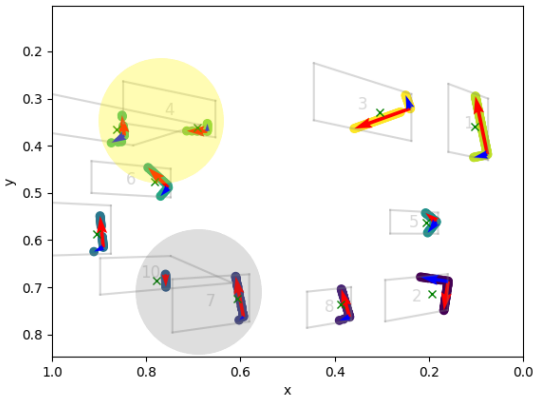
1.3 Algorithm to Evaluate the Accuracy of Block Classification

Firstly, we needed to **correlate in the right way the information obtained** from the model with the ones obtained from the point cloud. To achieve this, we project each detected

bounding box into 3D by taking the four vertices and mapping them onto the point cloud. An important consideration to make is about the fact that the model predicts the objects and gives us information about which object has been detected via a rectangle bounding box on a 2D image. What can happen is that a bbox covers an area which contains two or more blocks. Therefore, when we link a zed block to a yolo block, we need to be sure to correlate the right information.



[Figure] BBoxes in 2D



[Figure] BBoxes in 3D

What we do is **link a zed block to a yolo block** only if the center of the zed block is contained within the bounding box of the yolo block; if the center of the zed block is located within the bounding boxes of multiple yolo blocks, we do not associate the zed block with any yolo block. The idea is that if we are uncertain about which bounding box to assign to a zed block, we simply **discard it** from the "well detected" list and **wait for the next round of vision processing**. In one of the subsequent rounds, the blocks causing uncertainty may be repositioned, or the model might provide a more accurate detection.

The **list of "well detected" blocks**, which will be returned to the manipulator, consists of blocks that 1) have an associated prediction (i.e., a bounding box) with a confidence greater than a threshold (set to 0.65), and 2) have sides measured via the point cloud that match the dimensions of the predicted object.

If the list of blocks taken is empty, what we do is manage the special cases, which are 1) the object is perfectly parallel to the camera, and therefore only one side is seen or 2) the object has a confidence too low to be considered "correctly detected". In these situations, we choose an object to return to the manipulator by examining the blocks closest to the ZED camera-those with smaller x values for their vertex point. This approach prevents interference from blocks in front of the point cloud, which could obstruct or partially obscure the object being measured. What we do is check whether a prediction is associated to the block and, if true, we proceed as follow:

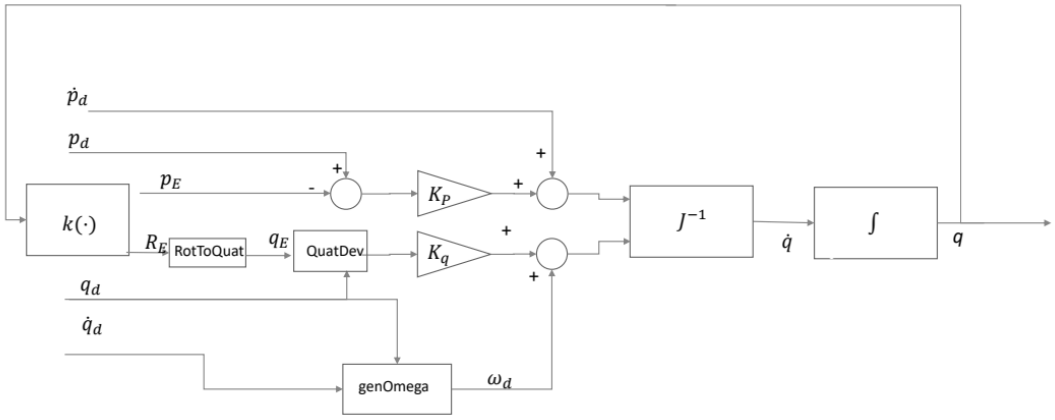
1. if the object is perfectly parallel to the camera, we check if the detected side length matches one of the predicted side lengths. If it does, we calculate the center point and yaw angle based on the measured side and on the predicted one, and return the block to the manipulator;
2. if the object has a low confidence but the lengths of both sides match the predicted values, we return the block to the manipulator.

For each block returned to the manipulator, we **compute its z-coordinate** by extracting the z value from the prediction name and scaling it to correspond to the block's actual dimensions.

2 Robot Motion

The *robot motion* is implemented using Professor Palopoli's Matlab scripts adapted for C++. To manipulate vectors, matrices and quaternions we used the **Eigen library**.

The main difficulty when implementing a robot motion algorithm is the **singularity** problem. In order to **avoid** this problem, we decided to implement a **differential kinematics** algorithm that uses **quaternions** instead of roll-pitch-yaw angles. This approach does not completely eliminate the problem of singularities, in fact, to overcome this limitation, we implemented a precise schema for the motion:



[Figure] Manipulator Control Scheme

In particular, we implemented a **precise** differential kinematics control algorithm that uses **linear interpolation** for position of joints, **spherical linear interpolation** for orientation of joints and **direct kinematics**.

We decided to use **geometric jacobian** and the pseudo inverse jacobian method with **damping factor** to avoid singularities. This approach uses the "*Nakamura*" method, which consists in computing the damping factor λ using the manipulability measure w, a threshold value w_t and the maximum damped factor λ_0. The approach is given by this formula:

$$w = \sqrt{\det(JJ^T)} \quad , \quad \lambda = \begin{cases} \lambda_0 \left(1 - \frac{w}{w_t}\right)^2 & \text{if } w < w_t \\ 0 & \text{if } w \geq w_t \end{cases}$$

This approach is **better** than using a fixed damped factor because λ changes its value depending on the arm configuration: when the arm is *near a singularity* configuration, the manipulability measure w has a value near zero, so, given the formula, the damped factor λ will be closer to λ_0 and the robot can easily **exit the singularity** configuration, and vice versa, if the robot is *far from the singularity*, the manipulability measure w has a value near w_t or bigger, so the the damped factor λ will be closer to zero and the robot will be **more accurate**.

In addition, we used also a **secondary task** for staying away from the joint limits according to this formula:

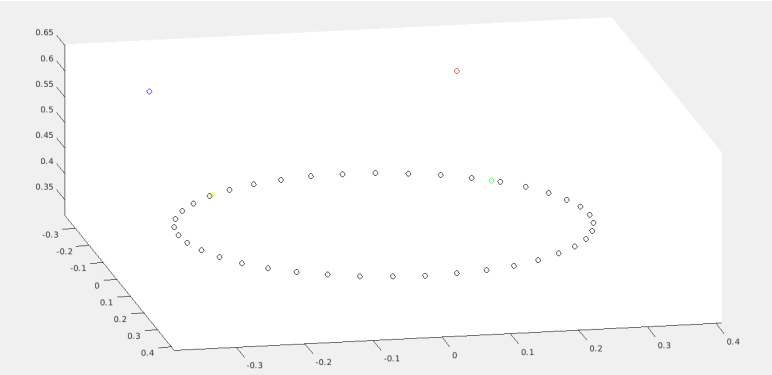
$$\dot{\mathbf{q}}_0 = -k_0 \left(\frac{\partial w(\mathbf{q})}{\partial \mathbf{q}}\right) \quad \Rightarrow \quad \frac{\partial w(\mathbf{q})}{\partial \mathbf{q}} = \frac{1}{n}$$

$$\Rightarrow \quad \frac{\partial w(\mathbf{q})}{\partial \mathbf{q}} = \frac{1}{n} \begin{bmatrix} \frac{1}{(q_{1M}-q_{1m})^2} & \frac{q_1-\bar{q}_1}{(q_{1M}-q_{1m})^2} \\ \frac{1}{(q_{2M}-q_{2m})^2} & \frac{q_2-\bar{q}_2}{(q_{2M}-q_{2m})^2} \\ \cdots & \cdots \\ \frac{1}{(q_{nM}-q_{nm})^2} & \frac{q_n-\bar{q}_n}{(q_{nM}-q_{nm})^2} \end{bmatrix} = \frac{1}{n} \begin{bmatrix} \frac{q_1-\bar{q}_1}{(q_{1M}-q_{1m})^2} \\ \frac{q_2-\bar{q}_2}{(q_{2M}-q_{2m})^2} \\ \cdots \\ \frac{q_n-\bar{q}_n}{(q_{nM}-q_{nm})^2} \end{bmatrix}$$

with $k_0 = 20$, $n = 6$, $\bar{q}_i = 0$ and $q_{iM} - q_{im} = 2\pi$.

3 High-Level Planning

As written before, the **main problem** of robot motion is the presence of **singularities** which occur in the area *under the robot's base joint*. So we implemented a precise motion planning that forces the robot to **stay away** from the area under the base joint. This is possible using a **circular trajectory** that the end effector has to follow during the movement.



[Figure] Circular trajectory of the end effector

This circular trajectory is made by **40 points equally spaced** at a specific height that we called `STD_HEIGHT=0.4` (from the base frame).

Whenever the robot has to move, the end effector follows a **path** which is always at the `STD_HEIGHT`. The end effector changes its z coordinates only when the robot has to reach the brick to be grabbed or it has to leave the block.

This decision allows the robot to **reach every point** of the table without hitting any obstacle or blocks.

The bricks are spawned in **random positions** on the table, and their positions are computed by the vision module and sent to the movement module by the client module. While the block's spawn position is random, its **final position is fixed** on the elevated part of the table and it depends on the block id.

The **plan** of the robot motion is the following:

1. get its initial position by asking the joint configuration to the robot using the topic `/ur5/joint_states`;
2. calculates the `finalPosition` (where the block has to be placed);
3. calculates the `initialPositionStdHeight` and the `brickPositionStdHeight`;

4. move the robot to the `initialPositionStdHeight`;
5. calculates the nearest point on the circumference with respect to the `initialPositionStdHeight` and the `brickPositionStdHeight`;
6. calculates the trajectory on the circumference between the points just computed and the direction of movement;
7. move the robot to the `brickPositionStdHeight` following the circular trajectory;
8. open the end effector tool;
9. go down to the block position;
10. close the end effector tool;
11. go up to the `brickPositionStdHeight`;
12. calculates the `finalPositionStdHeight`;
13. calculates the nearest point on the circumference with respect to the `brickPositionStdHeight` and the `finalPositionStdHeight`;
14. calculates the trajectory on the circumference between the points just computed and the direction of movement;
15. move the robot to the `finalPositionStdHeight` following the circular trajectory;
16. go down to the final position;
17. open the end effector tool;
18. go up to the `finalPositionStdHeight`.