ROBOTICS PROJECT REPORT

Filippo Adami, Mattia Barborini, Matteo Grisenti, Angelo Nutu

February 2024

1 Introduction

The project consists of using an anthropomorphic Ur5 robotic arm, equipped with a spherical wrist and a two-finger gripper as an end effector, to pick up a series of objects of various classes from an initial stand and place them on a final stand in an order specified. The objects, one per class, are stored without a specific order on the initial stand, but are distinguished by their respective known geometries, encoded in the STL files. The goal is to use a calibrated 3D sensor to locate objects, recognize their class and detect their position on the initial stand, then the robotic arm will have to pick up each object and place it in the final position reserved for the class of the object itself.

2 Project Structure and Workflow

2.1 ROS structure

Before proceeding with writing the code it is necessary to define the **Ros structure** of the project, which will have to manage a first section of **Robotic Vision**, capable of identifying the blocks on the table and obtaining their position and orientation, a second section of **Kinematics**, which instead will take care of calculating the various paths that the arm will have to follow to complete the delivery, and finally a third section capable of managing the communication, and therefore the sharing of data, between the two previous sections. Therefore the project was divided as follows:

- **vision_planner**: it is the Ros package containing all the code for the Robotic Vision section, in particular it contains the **main.py** node, whose workflow is described in the third section, thanks to which the detection will be performed;
- **motion_planner**: it is the Ros package containing all the code for the Kinematics section, in particular it contains the **inverse_kinematics_node** node, whose workflow is described in the fourth section of this document, necessary for computing the trajectories of the robotic arm.
- **task_planner**: it is the Ros package containing the code to manage communication and data sharing between vision_planner and motion_planner, in particular the **state_machine_task_node** node manages the workflow, and therefore the functioning, of the entire project.

2.2 Comunication between Nodes

The state_machine_task_node communicates with the main.py and inverse_kinematic_node node, therefore with the vision_planner and motion_planner sections, thanks to two services:

ObjectDetection.srv:

- **Request**: it is an empty request, without any attribute.
- **Response**: it is composed by three float64 array: one for the poses, one for the orientations and one for the names of the detected blocks.

InverseKinematic.srv:

- **Request**: it is composed by three float64 arrays: one for the actual joint configuration, the other two are for the desired end-effector position and rotation. Furthermore it has other two attributes for the menage of the output log file where are printed some debugging information, them are a string title for the title of the file and a boolean flag that indicate if it is the first request of the task planner. In the end it has a boolean attribute that indicate if the request is for a grasping operation.
- **Response**: it is composed by a float64 array that contain all the set of the joint configuration derived by the Motion part

The request will always be sent by the task_planner, while the vision_planner and motion_planner sections will transmit the responses.

2.3 Workflow

After initializing the Ros node, the **go_to_start_position()** function is called, so that the arm can position itself in its standard configuration. Then, invoking the **ask_object_detection()** function, the detection process could begin. This function calls the **objectDetection.srv** service to locate all the blocks on the work surface and organizes them into a vector **"blocchi"**. Finally, for each block present in the vector **"blocchi"**, the **ordering_block()** function is invoked, which will calculate the trajectories that the arm will have to follow to pick up the block on the table and deposit it in its defined position. For calculating trajectories, the **ordering_block()** function will invoke the **ask_inverse_kinematic()** function, which through the **InverseKinematic.srv** service will trace the trajectories needed for the arm movement. All the operations mentioned, such as detection and trajectory calculation, will be discussed later in the document.

3 Perception

3.1 2D - Object Recognition

The starting point of the project was solving the task of detecting blocks positioned on a table to facilitate subsequent estimation of their pose, enabling the UR5 robotic arm to grasp and relocate them to specified destinations.

The dataset employed for training comprised over **10.000 images generated using Blender**, employing a script to randomize block positions and orientations within the environment. This dataset was partitioned into 70% for training and 30% for validation, with a subset of images intentionally devoid of blocks to prevent the model from associating background elements with objects. The chosen computer vision model is **YOLOv8**, and training was conducted locally. Initial training sessions revealed a sluggish convergence rate with the default Stochastic Gradient Descent (SGD) optimizer, occasionally resulting in suboptimal outcomes characterized by local minima, thereby impeding overall training efficacy. Consequently, the decision was made to adopt the **Adam optimizer** due to its reputation for faster convergence, particularly suited for scenarios involving extensive data and parameters.

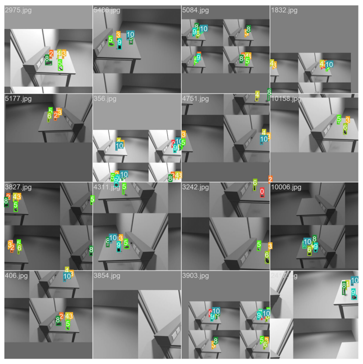


Figure 1: Example of augmented dataset batch

To mitigate risks of overfitting or underfitting associated with Adam's high convergence rate, a strategy was devised to decouple the weight decay term from the gradient update, accomplished through utilization of the **AdamW** variant of the Adam optimizer.

Two versions of the model have been developed. The **Grayscale version** yielded optimal results in Gazebo simulation, while the **RGB version** demonstrated proficiency in real-life scenarios. Following model development, a method for inference was required. Although the Ultralytics library, used for training, was a viable option, we chose a more hands-on approach to enhance performance and ensure model portability across diverse platforms, independent of the architecture it would run on.

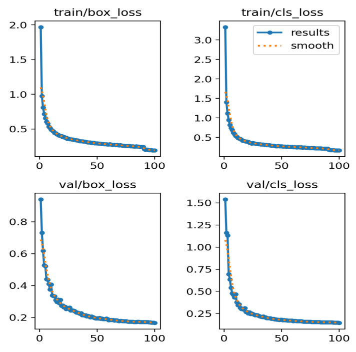
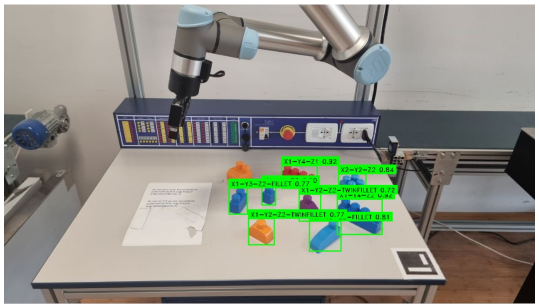
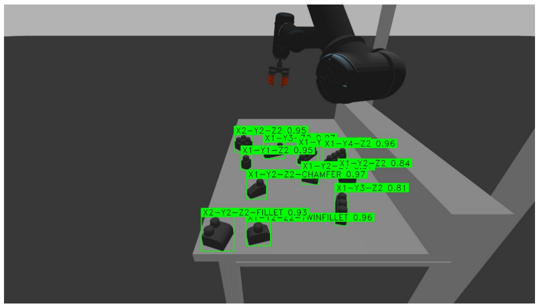


Figure 2: Train and validation loss curves throughout training epochs

Thus, the model was exported to the **ONNX format**, offering advantages such as the aforementioned interoperability across various frameworks and hardware platforms, but also hardware specific acceleration. Subsequently, the **OpenCV DNN library** was employed for inference. Our methodology involves initial image pre-processing to conform to the input tensor requirements of the ONNX model. Subsequently, inference is performed to obtain results from the **unconnected layers** within the model, which serve as the output layers.

The outputs from the inference phase are then analyzed to extract pertinent details such as **object IDs, confidence scores, bounding box coordinates, and 2D centroids**. These extracted details undergo filtering based on a predetermined confidence threshold, ensuring that only significant detections are retained. **Non-Maximum Suppression (NMS)** is then


(a) Example of real-life detection

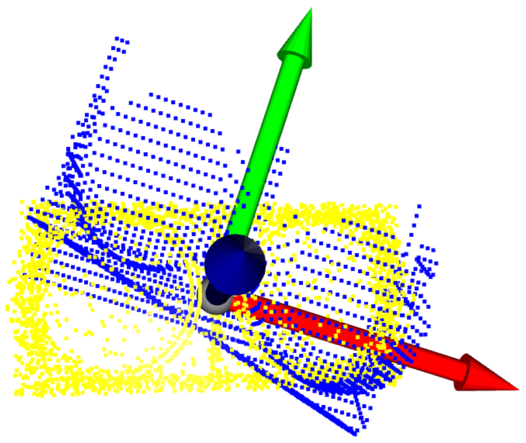

(b) Example of simulated detection

employed to remove redundant detections and enhance the accuracy of the final detected object set.

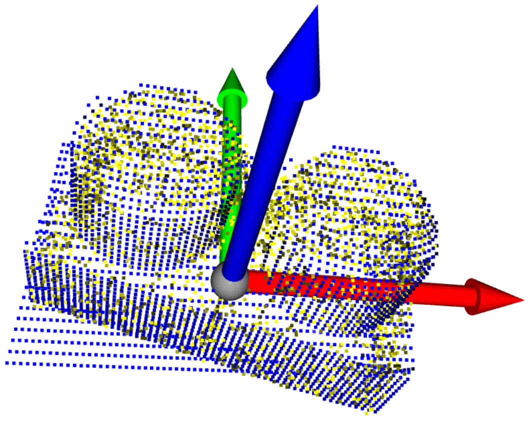
3.2 3D - Pose Detection

Following the successful 2D object recognition achieved through the YOLOv8 model, the subsequent endeavor entailed acquiring the precise position and orientation of the identified objects within the three-dimensional (3D) world frame.
To accomplish this objective, a **point cloud** output obtained from the **Zed2 stereo camera** was utilized to establish a comprehensive 3D representation of the environment. However, a significant challenge arose due to the inherent *disparity in coordinate systems* between the point cloud and the image captured by the left camera of the Zed2 stereo camera, which served as the input for the YOLO detection process.
To align the point cloud with the image captured by the left camera, a series of transformations were executed, and, finally, the point cloud was converted into a **depth frame**, leveraging the **intrinsic parameters** of the camera. Subsequently, given that the newly obtained depth frame was indeed aligned with the initial image, it was cropped in accordance with the *2D bounding boxes* returned by the YOLO model.
Noted that the center of each bounding box corresponded to the projection of the *block's center* onto its surface within the image plane, a reverse transformation using the inverse of the intrinsics matrix facilitated the derivation of the precise three-dimensional coordinates (x, y, z) of the block centers within the 3D-camera frame.
Regarding the determination of block orientations, the process involved extracting depth image crops corresponding to the detected objects, which were subsequently transformed into point clouds; this time without relying on camera intrinsics. These point clouds underwent rigorous filtering to eliminate extraneous elements, including portions of neighboring blocks and background clutter. This purification process was executed through robust methodologies such as **RANSAC clustering** and background subtraction.
Once the pristine point clouds of the blocks were obtained, a *three-start* **Iterative Closest Point (ICP)** algorithm was employed for precise alignment with reference models. Notably, due to the perspective-dependent perceived dimensions of the blocks, a **dynamic scaling** factor was applied to the reference models. The three-start ICP approach was adopted to mitigate potential issues stemming from significant rotational disparities between the target and reference models. By initializing ICP predictions from three distinct rotation angles and selecting the prediction yielding the lowest error among them, robust alignment results were achieved.

To enhance model validation and refine pose estimation accuracy, we devised a system that iteratively conducts Iterative Closest Point (ICP) calculations using features derived from the blocks. This system addresses **potential mismatches** in classification among blocks of



(a) cropped pointcloud (blue) and reference mesh (yellow) before ICP



(b) cropped pointcloud (blue) and reference mesh (yellow) after ICP

Figure 4: ICP alignment process example

identical dimensions but differing types by analyzing **Root Mean Square Error (RMSE) outcomes** from various ICP procedures. It then determines the most probable classification based on these results. Through this approach, misclassifications are effectively mitigated, ensuring the identification of a **singular instance** for each distinct class within the workspace, consistent with anticipated outcomes.

4 Robot Motion

4.1 Quaternions

The second part of the project aims to control the movement of the Ur5 arm, making it able to move freely within the workspace, in order to achieve the goal of reaching the various blocks on the table and transport them to their final position. Controlling arm movement requires precise knowledge of its kinematics, which determines the position and orientation of its joints in response to given inputs. Among the various methodologies available to calculate the kinematics, we opted for the use of **quaternions**, as they offer a series of advantages, including a better representation of the orientation in space and the management of singularities. In particular, quaternions provide a more compact and efficient representation of the orientation compared to other techniques, such as rotation matrices: while a rotation matrix requires 9 parameters to be defined, a quaternion requires only 4, reducing computational complexity and storage load. This not only makes integration with algorithms and functions for controlling the movement of the arm itself easier, but allows rotation operations to be performed more accurately, making them less subject to approximation errors. However, the main advantage is given by the geometric nature of the quaternions, which allows **singularities** to be avoided. For example, solutions using representations based on Euler angles are subject to the phenomenon called **"gimbal lock"**, a particular case in which two rotation axes become aligned, causing the arm to lose one or more degrees of freedom, effectively compromising the functioning of the system.

4.2 Trajectory Computation

The idea for controlling the movement of the arm consists in initially calculating, via the **DirectKinematics()** function, the forward kinematics, passing as parameters the current configuration of the arm (appropriately provided by the task planner), in order to obtain the current position and orientation of the end-effector. Subsequently, a second function **invDiffKinematicControlSimCompleteQuaternion()** will be invoked capable of computing the trajectory for the arm movement. In particular, a certain period of time will be chosen, corresponding to the total duration of the movement, and it will be divided into a specific number of "steps"; for each step the function will calculate the position, orientation and linear and angular velocity of the end-effector, and will compute, by invoking a specific function **invDiffKinematicControlCompleteQuaternion()**, the inverse differential kinematics to obtain the velocity of the joint. The function will return a matrix in which each row will represent the configuration of the arm at a given instant of time (or step), effectively tracing the trajectory that the arm will have to follow.

4.3 Collisions

To guarantee the correct functioning of the arm, it is necessary to take into consideration some factors that can negatively influence the behavior of the system. The first major challenge is to avoid possible physical obstacles that may hinder the movement of the arm. In particular, it is necessary to check that, during movement, no part of the arm collides with the structure on which the robot is mounted. To do this, the contact areas were initially identified in the surface of the table, the "step" (structure containing the electrical controls) also present on the table and the support columns of the structure. To carry out the check, for each configuration over time the function **posizioneGiunti()** will be called, which will return a matrix whose each of the 6 rows represents the position, expressed in coordinates (x,y,z), of the corresponding joint. This matrix will then be passed as a parameter to the **checkCollisionSingularity()** function which will check that no configuration collides with the structure. Once this problem has been resolved, it is necessary to consider that, during the movement, the arm will not have to hit the blocks on the work surface. The solution is simple as well as effective: first the trajectory that will lead the arm from its current position to the coordinates of the selected block will be calculated, however the block will not be reached directly, but an intermediate point located exactly 10 cm above the center of the block itself, from which to then descend vertically and proceed with the grasping operation. During the first movement, the **checkCollisionSingularity()** function will consider the surface of the table elevated by 5 cm along the z axis, so that the arm does not have the possibility of hitting any blocks. Once the block has been grabbed, the same procedure will be repeated, so the arm will move vertically 10 cm and then lead the block to the desired position.

4.4 Singularities

Although it was previously said that quaternions allow us to solve the problem of singularities, in reality this is not strictly true. By singularity we mean a point in the workspace where the robotic arm loses degrees of freedom and the speed of the joints grows exponentially, well beyond the physical limits of the joints, making the system's behavior unstable. Now through quaternions we can be sure that the arm will never pass through a singularity point, however it cannot be prevented from passing near one of these points. A point is considered to be a singularity if the determinant of the Jacobian matrix of the system is 0, however, after numerous attempts and experiments, it was found that the behavior of the Ur5 arm becomes unstable for configurations in which the determinant of the analytical Jacobian matrix is lower than the absolute value of **0.5**. To ensure correct behavior of the system, the **checkCollisionSingularity()** function, in addition to checking for possible collisions as explained in the previous point, will also check that the arm's trajectory does not pass close to singularity points.

4.5 Solutions

In summary, before we can guarantee that the trajectory calculated by the **invDiffKinematicControlSimCompleteQuaternion()** function is correct we must invoke the **checkCollisionSingularity()** function, which for each configuration over time will check that the arm does not collide with the structure and does not pass close to a singularity point . In the event that this function recognizes a collision or a singularity it will be necessary to calculate an alternative trajectory to reach the destination point. In this case the **alternativeTrajectory()** function will be called, which will generate a point in a random position in the workspace and calculate the trajectory from the starting point to that point and subsequently from the intermediate point to the final point. If in these two movements the **checkCollisionSingularity()** function detects singularities or collisions, a new random point will be generated and the procedure will be repeated. In order to avoid infinite cycles, the **alternativeTrajectory()** function will generate a maximum of 10 random points. If in no case a valid trajectory is found, a further step will be used which consists in calculating a trajectory that passes through two intermediate points called "safe points". If the block cannot be reached in this case too, the system considers the point unreachable and will move on to the next block.