UNIVERSITÀ
DI TRENTO

Project Report

Fondamenti di robotica

Gruppo B

Luca Boschiero, Mauro Meneghello, Usama Aarif

GitHub repository

Luca Boschiero	217460	luca.boschiero@studenti.unitn.it	GitHub
Mauro Meneghello	217564	mauro.meneghello@studenti.unitn.it	GitHub
Usama Aarif	217696	usama.aarif@studenti.unitn.it	GitHub

Indice

Parte I

Introduction

1 Goal

The main goal of this project is to develop a software entity that is able to move a UR5 robotic manipulator and, by controlling it, pick up and move some lego blocks placed near the robotic arm. These blocks are different from each other in terms of shape and position, and they need to be detected and classified in order to allow the robot to do its job. The camera that was used in the detection part was a 3D depth Zed camera, while the classification part was done with vision tools such as Yolo and Open3D.

2 Tasks

The project is composed of 4 tasks with increasing difficulty, but all of them follow the same path: first detect the position and typology (or class) of the lego block/s, and then pick them up using the robotic arm and moving them into the desired position.

2.0.1 Task 1

There is only one lego block on the table, standing in its natural position, that has to be moved to a final position according to its class.

2.0.2 Task 2

Many lego blocks are placed on the table in their natural position. They have to be moved to a final position according to their class.

2.0.3 Task 3

Many lego blocks are placed on the table, lying on their lateral side or on their top. They have to be rotated and moved to a final position according to their class, and objects with the same class have to be placed on top of each other.

2.0.4 Task 4

Many lego blocks are placed randomly on the table. They have to be moved in order to create a sort of building based on a known design.

Parte II

Kinematics and Motion

Kinematics is about the geometrically possible motion of a robotic structure without considering the forces involved. In particular, it establishes a relationship between the robot's joint configuration and the end-effector position and orientation.

During the realization of our project, we developed several types of kinematics starting from the Matlab Scripts available on the website of the course.

The motion part deals with the use of kinematic techniques to move the arm to the right position and to grasp the lego blocks.

1 Direct Kinematics

The goal of direct kinematics is to find the pose of the end-effector given the joint variables.

In order to do this, we computed the homogeneous transformation matrix for each joint variable ($T^i_{i+1}(q)$)and by multiplying them together we found the final matrix ($T^0_6(q)$).

Finally, exploiting the homogeneous transformation composition, we extracted the position and orientation (rotation matrix) associated to the joint configuration of the robotic arm.

$$T^0_6(q) = T^0_1(q_1) \cdot T^1_2(q_2) \cdot \ldots \cdot T^6_6(6) = \begin{bmatrix} R^0_6(q) & p^0_6(q) \\ 0^T & 1 \end{bmatrix}$$

2 Inverse Kinematics

It is the contrary of the direct kinematics: given the position and orientation of the end-effector it finds the configuration(s) of the joint variables. However, the problem is non-linear and it cannot be solved in a closed-form expression, and that means that it may have no solution as well as multiple or infinite solutions.

In our scenario, the inverse kinematics problems returned a 8x6 matrix, where each line is a possible configuration for the joint vector.

$$Th = \begin{bmatrix} Th_{0,0} & Th_{0,1} & Th_{0,2} & Th_{0,3} & Th_{0,4} & Th_{0,5} \\ Th_{1,0} & Th_{1,1} & Th_{1,2} & Th_{1,3} & Th_{1,4} & Th_{1,5} \\ Th_{2,0} & Th_{2,1} & Th_{2,2} & Th_{2,3} & Th_{2,4} & Th_{2,5} \\ Th_{3,0} & Th_{3,1} & Th_{3,2} & Th_{3,3} & Th_{3,4} & Th_{3,5} \\ Th_{4,0} & Th_{4,1} & Th_{4,2} & Th_{4,3} & Th_{4,4} & Th_{4,5} \\ Th_{5,0} & Th_{5,1} & Th_{5,2} & Th_{5,3} & Th_{5,4} & Th_{5,5} \\ Th_{6,0} & Th_{6,1} & Th_{6,2} & Th_{6,3} & Th_{6,4} & Th_{6,5} \\ Th_{7,0} & Th_{7,1} & Th_{7,2} & Th_{7,3} & Th_{7,4} & Th_{7,5} \end{bmatrix}$$

3 Inverse differential Kinematics

We also used a differential inverse kinematics algorithm in order to find the joint velocities and positions that follow the trajectory, exploiting their relations represented by the Jacobian. The Jacobian is a matrix that describes the linear relation between the velocities of the end-effector and the velocities of the joints.

We applied differential inverse kinematics using two functions:

- invDiffKinematicControlSimComplete simulates the inverse differential kinematics control and, given the initial position and orientation of the end-effector and the initial joint configuration, it finds the desired velocity and angular velocity and calculates the joint vector in the following period of time using numerical integration, that can be approximated as follows.

  $$q(t_{k+1}) = q(t_k) + q^{'}(t_k) \cdot \Delta t$$

  During the process, some control operations are performed to handle the singularity and ensure that the robot moves safely and correctly.

- invDiffKinematicControlComplete calculates the derived joint vector using the inverse of the Jacobian (this is possible because the matrix is squared) and the desired velocity and angular velocity.

  $$q^{'}(t_k) = J^{-1}(q) \cdot v_e$$

  It also applies a check on the error between desired orientation and current orientation and limitates it to avoid oscillations.

3.1 Control

The problem with Euler approximation is that errors accumulate and could generate a mismatch between the desired position and the actual one. To avoid this problem, we applied a control and hence made a correction by trying to reduce to zero:

1. the error ($e(t)$) between desired($x_d(t)$) and actual position($x_e(t)$) of the end-effector.
2. the error ($error\_o$) between desired(w_R_d) and actual orientation(w_R_e) of the end-effector.

$$e(t) = x_d(t) - x_e(t)$$
$$e^{'}(t) = x_d^{'}(t) - x_e^{'}(t) = x_d^{'}(t) - J(q) \cdot q^{'}$$

Since the Jacobian is invertible, we chose $q^{'}$ so that:

$$q^{'} = J^{-1}(q) \cdot (x_d^{'} + K_d \cdot e(t)) \cdot (K_{phi} \cdot error\_o)$$

4 Motion

To develop a motion plan for our robotic arm, we created a MoveTo function that uses differential inverse kinematics to generate a trajectory to follow, from an initial position and orientation to a final one, and moves the robot along this path.

Using this function, the motion plan is composed of a series of fixed steps as follows:

1. Moves the robot from the initial homing position towards the first detected block (the position and orientation of each block is available in a matrix thanks to the vision part).
2. Lowers the arm to the block to grasp it, with the gripper opened.
3. Closes the gripper and grasps the block.
4. Raises the arm with the grasped block.
5. Moves the robot to the final position, that depends on the class of the block.
6. Puts the block down to the final stand and leaves the block by opening the gripper.
7. Moves back to the homing position and, if other blocks are detected on the table repeat the motion plan, otherwise stops its execution.

Parte III

Vision

1 Yolo

YOLO (You Only Look Once) is a popular object detection algorithm developed by Ultralytics that was introduced in 2015. It revolutionized the field of computer vision by significantly improving the speed and accuracy of real-time object detection.

Its primary objective is to identify and locate objects within an image or video stream and provide bounding box coordinates along with class probabilities for each detected object. This information is crucial for robotics tasks such as autonomous navigation, object tracking, and human-robot interaction.

Integrating YOLOv5 with ROS (Robot Operating System) allows to leverage its capabilities within a robotics framework. ROS provides a flexible and modular environment for developing robotics applications, making it an ideal platform for implementing YOLOv5-based object detection on live image topics.

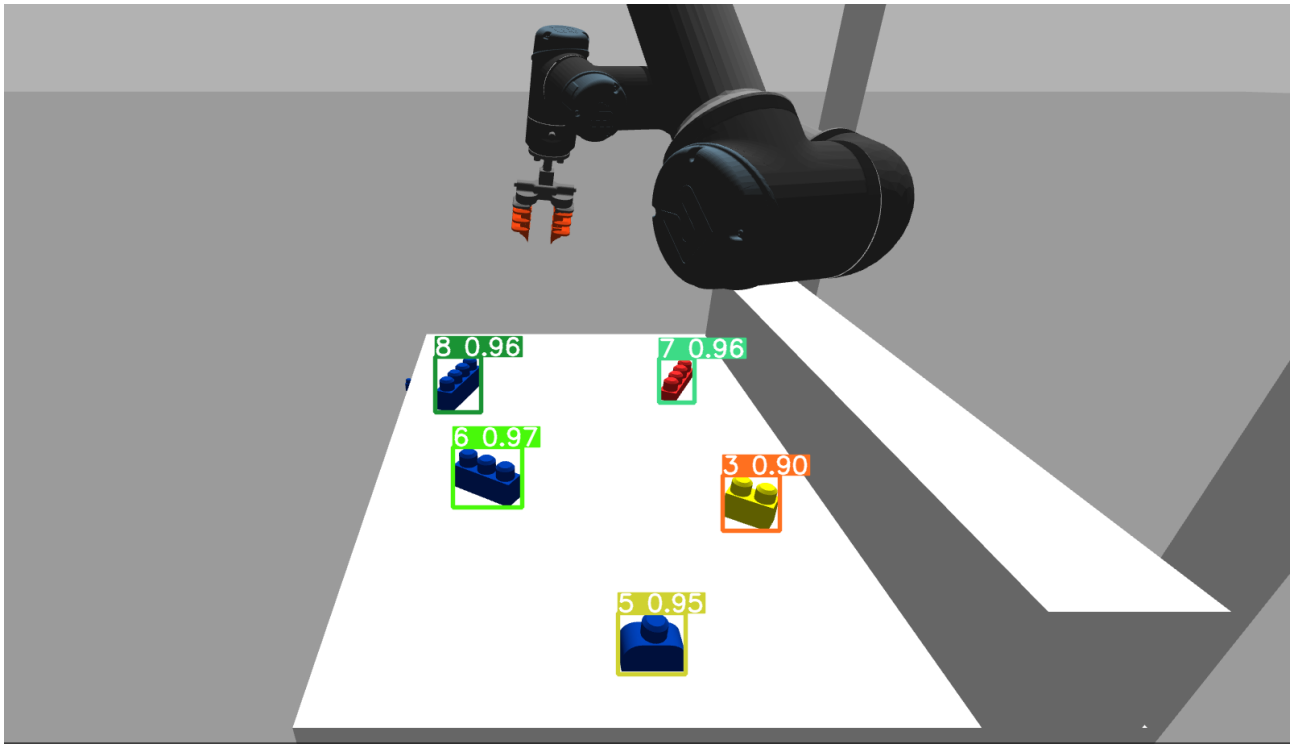
To integrate YOLOv5 with the Robot Operating System (ROS) for real-time object detection, we used an existing GitHub repository named "yolov5-ros". This approach simplifies the setup process by providing a pre-configured ROS package specifically designed for YOLOv5 integration. It allows for seamless integration of YOLOv5's speed and accuracy into the modular ROS framework, making it ideal for robotics applications.

To begin, we cloned the "yolov-5ros" repository from GitHub, and run catkin-make to build the "yolov5-ros" package and its dependencies. This command compiles the code and generates the necessary executables for running YOLOv5 with ROS.

Within the "yolov5-ros" package, we modified the provided configuration files to suit our requirements. These files include network architecture, weights, and detection parameters. We also trained our custom weights, on a database of almost 2000 pictures in local and renamed every block for better debugging,

Launching the "yolov5-ros" package we can subscribe to the desired image topic and perform object detection on incoming images. The package automatically processes the images using the YOLOv5 model and publishes the detected objects as ROS messages.

To visualize the detected objects, the "yolov5-ros" package offers integration with image-view.



2 Object orientation

After detecting the objects using yolov5-ros, the next step is to get the orientation of the blocks to rotate the end effector to correctly pick up the blocks. The pose estimation system takes as input the point cloud data from the depth camera and compare it with the STL model of the blocks, with a method called global registration of the Open3d library.

First of all we take the output from the topic /yolov5/detections to obtain the information about the detected objects: the bounding boxes and the class of the blocks. The coordinates of the bounding boxes are first of all transformed in the camera frame and then to the world frame. After these transormations the coordinates are published in the topic /coordinates, which is used by the *custom_joint_publisher* to move the robot. We created a custom message, whose name is Coord, to publish the position and the orientation of the blocks in this topic.

Then we transorm the STL file in an Open3d pointcloud and on both pointclouds we performe voxel downsampling to reduce the number of points, estimating surface normals for each point, and computing Fast Point Feature Histograms (FPFH). These FPFH features represent the local geometric information of the point cloud.

The open3d registration method utilizes the RANSAC (Random Sample Consensus) algorithm to perform point cloud registration. It takes as input two downsampled point clouds, along with their associated FPFH features, and finds correspondences between the points using feature matching.

RANSAC then iteratively estimates a transformation (rotation and translation) that aligns the input point cloud with the point cloud from STL. This registration process helps align the detected object's point cloud with a reference point cloud obtained from an STL file.

In addition to RANSAC, the code also employs the open3d ICP (Iterative Closest Point) algorithm. After the initial registration using RANSAC, ICP is applied to further optimize the alignment between the two point clouds. ICP minimizes the distance between corresponding points from the input point cloud and the point cloud from the STL, iteratively refining the transformation until convergence.

Finally we obtain the orientation matrix of the blocks and we publish it with their position.

Parte IV

Conclusion

1 Performance Indicator

Task 1 :

- KP1: time to detect the position of the object
- KP2: time to move the object between its initial and its final positions, counting from the instant in which both of them have been identified

Task 2 :

- KP1: total time to move all the objects from their initial to their final positions

Task 3 :

- KP1: total time to move all the objects from their initial to their final positions

	KP 1	KP 2
Task 1	1 - 2 s	20 - 25 s
Task2	60 s (3 blocks)	/
Task 3	Only detection	/

2 Outcome

Task 1 and Task 2 have been successfully completed. We achieved the desired outcomes with a discrete precision. The only problem is that sometimes the block remains attached to the gripper and, instead of being realesed, it does not come off.

For Task 3, the detection part is functioning correctly. However, due to time constraints, we couldn't implement the code for block rotation and movement.

Task 4 was not completed.