Fundamentals of Robotics, University of Trento
Teachers: Focchi Michele, Palopoli Luigi, Sebe Niculae

UNIVERSITY OF TRENTO - Italy
Department of Information Engineering and Computer Science

UR5

A manipulator to reorder Lego Blocks

A project done by Alex Pegoraro, Alberto Dal Bosco, Federica Lorenzini, Jacopo Veronese

---

INTRODUCTION

Problem description: Several mega-blocks, like big pieces of LEGO, are stored without any specific order on a stand located within the workspace of a robotic manipulator.

The manipulator is an anthropomorphic arm, with a spherical wrist and a two-fingered gripper as end-effector. The objects can belong to different classes but have a known geometry (coded in the STL files).

The project's goal is to use the manipulator to pick the objects in sequence and position them on a different stand according to a specified order (final stand).

A calibrated 3D sensor is used to locate the different objects and to detect their position in the initial stand.

Our group decided to divide this complex and articulated project into four subparts, each assigned to a member of the group:

- Detection & Classification Part with Computer Vision *(Alberto Dal Bosco)*
- 3D Positioning & Orientation Part *(Jacopo Veronese)*
- Motion Planning Part *(Federica Lorenzini)*
- High-Level Planning & Task Planner *(Alex Pegoraro)*

The division of the project into these 4 subparts allowed us to optimize the work effectively, with each person working independently on their part, referring to the initially agreed-upon interface. This ensured that the final merging process was as simple and quick as possible. In our opinion, this was the key to achieving the best performance.

Detection & Classification Part with Computer Vision

In this first part of the project, the goal is to create a precise AI model capable of detecting mega-blocks and classifying them throughout the environment.
For the fine-tuning process, we decided to utilize Ultralytics YOLOv8, a cutting-edge, state-of-the-art (SOTA) model that builds upon the success of previous YOLO versions and introduces new features and improvements to further enhance performance and flexibility. YOLOv8 is designed to be fast, accurate, and user-friendly, making it an excellent choice for a wide range of tasks including object detection and tracking, instance segmentation, image classification, and pose estimation. YOLO follows its conventions, so initially, we modified the dataset provided by Professor Sebe and adapted it to YOLO standards.

DATASET

The Sebe dataset is organized into three subfolders, each of them containing a series of scenes with different compositions of situations. For each situation, the dataset gives: image.jpeg, bbox_image.jpg, vertices_image.jpg, depth_image.jpg, depth_plane_image.jpg and annotation.json.
I apply a series of scripts in cascade:

1) JSONtoYOLO.py → This script converts the Sebe dataset, taking only the image and the .json annotation file, into a YOLO v8 compatible dataset. The images are renamed (prefix + incremental number) and saved in a folder. The same process occurs for the labels, but only after processing the .json file to extract realistic bounding boxes through filtering of the 3D bounding boxes (the 2D ones are wrong).
2) split_dataset.py → This script reorganizes the images and their corresponding labels into three subfolders: TRAIN, VAL, and TEST, splitting them randomly by 70%, 20%, and 10%, respectively.

At this point, we manually added to the dataset images taken with a phone after annotating them with an online tool. This allows us to have a mixed dataset, not only with simulated images (Sebe dataset) but also with realistic images taken in a real environment. This step has proven crucial: the network is able to recognize real mega-blocks very well. These images and their respective annotations were also separated into the same subfolders as before.

3) Augmentation.py —> It takes each image from the TRAIN and VAL folders, resizes it to 640 * 640 pixels, and applies a Black/White filter and a random brightness filter. It also keeps the original version, while the others are renamed with the addition of a prefix. Note: the same process occurs in parallel for the labels so that each image is associated with its unique annotation file.

This step is fundamental for increasing the dataset size (in this case, tripled), allowing for more flexible training of the network, for example with darker, larger, or inverted images.
The dataset is now ready to be used, consisting of approximately 11,000 images and 11,000 labels (300 of ours)

TRAIN MODEL

For the fine-tuning process of the network, We've tried different approaches:

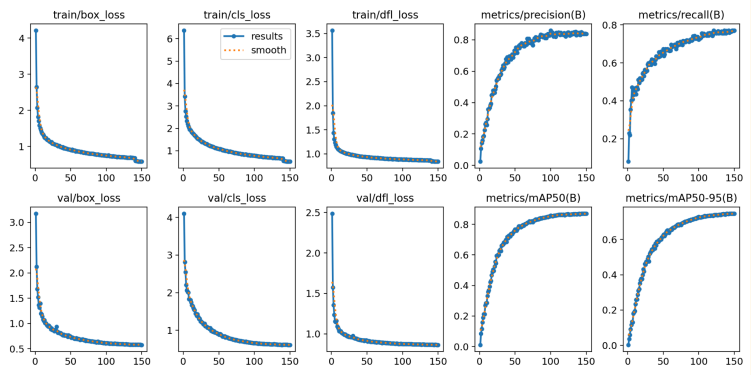
1) fine_tuning.py → This script is used for offline fine-tuning on the host machine. However, this process is very time and resource-intensive, making it unsustainable for our model (more than 30 hours).
2) Google Colab → online services that provide highly performant GPUs. The major drawback of these services is their time and usage limitations; thus, we have experienced multiple instances of losing all data due to sudden interruptions. This was partially resolved by splitting the training into multiple Google accounts thanks to the resume_training.py script. By doing this, we have prevented system crashes due to timeout.

After various trials and comparisons, we obtained the best model (trained for 200 epochs, taking 5 hours).
For this purpose, it is also necessary to create a config.yaml file to define the path of the dataset, information about bb, and other configuration aspects.

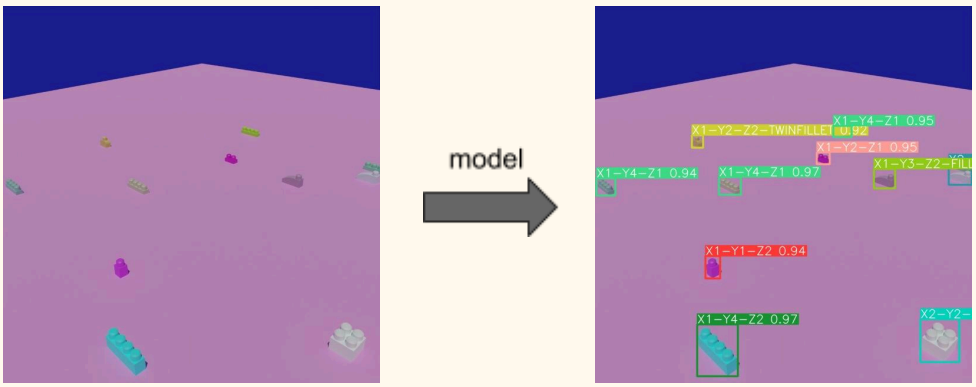
The last but most important file is custom_model.py. With this script, it is possible to make inference on the real image of the zed camera to obtain bounding box information. This is the only script used by ROS.

Here are some statistics highlighting the performance of the model and some incredible results of prediction.

It's evident from the graph that the accuracy increases as the number of epochs increases, while the error, both in training and validation, decreases significantly. After 150-200 epochs, the values stabilize, confirming that this range is ideal for our model to avoid overfitting or underfitting.



Here I also report images before and after passing through the model:

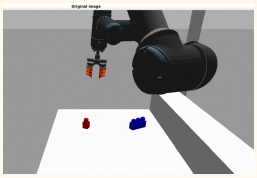


As we can see, the image on the right (taken from the TEST folder) has, in addition to the mega-blocks in the background, bb with their corresponding IDs and confidence values.

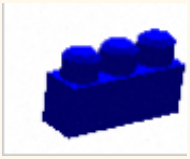
Appears very high, demonstrating a well-trained network.

3D Positioning & Orientation Part

Once the objects had been successfully recognized, the images were exploited alongside the bounding boxes previously identified on the zed's image. The goal for this problem was to isolate the object itself and find all the points that were part of it.

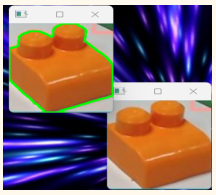


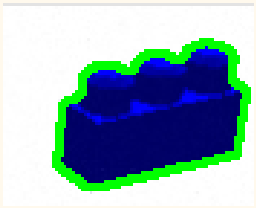
A solution was found by first cutting the zed's image using the bounding boxes' measure (we sliced the image) and adding a little offset on both right, left, upper and lower sides of the sliced image to make sure not to lose any relevant section of the brick. The resulting cropped image was basically the brick surrounded by a bit of background. At this point one could simply interrogate the pointcloud asking for the corresponding points in 3D, however, this would result in a big image containing all the points including the background ones creating a few problems when dealing with the following reasonings regarding the position detection of the brick. One example is that, with this method, the resulting centroid was absolutely not right at all.

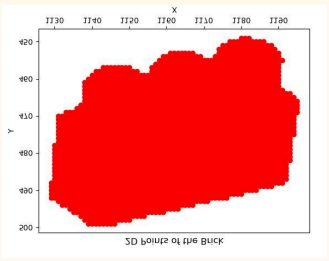


To avoid this case, we needed to extract only the points that were part of the object itself and we used contours to do so. The cropped image was in fact passed to a function (**find_object**) that had the role of finding the largest contour contained in an image. Then all the contours are iterated and the biggest one (which, considering our approach, should be the one containing the brick) is selected by discriminating it based on its area.

Now, **discriminate_brick_points** finalizes the job and finds all the points that are part of the contour and that are part of the internal part of the contour. The points are added with another offset that re-scales the image into the original size of the zed's point of view.







We now have found all the tuples of the object and we can try representing them. This result is, we think, quite satisfactory and is consistent with the positions of the object in the original image. UV Points obtained from the pointcloud can finally be interrogated to find the corresponding 3D points. These represent the real objects in the simulation. Then we had to convert all the points from the zed camera frame (which is the one that told us where the 3D points were) to the world frame where all the motion planning techniques take place. This is done with the tf2 library that allowed us to listen to the ROS master for the needed frames. We just tell our function "Take this coordinate and convert it to the world frame knowing that it's currently expressed in the zed camera frame" and it does everything for us. The only downside of this is that the process is quite slow because we need to convert every single point. An alternative solution (not used but tested by us) consisted of simply using rotation matrices along with a linear adjustment.

Once the points had been converted into the world frame, we were able to find the centroid by passing the whole set of points to **find_centroid.** This function takes as said the 3D points and finds the centroid (center of mass) of the object. This is done by np.mean(points_array, axis=0) that calculates the mean along each axis, resulting in the centroid's (X, Y, Z) coordinates.

Regarding the orientation, we chose to exploit the models given by Professor Sebe and we used the open3d library to try aligning the pointcloud coming from the STL file of the model to the pointcloud extracted from the actual object in the scene. Basically, we convert in open3d's pointclouds both the source pointcloud (the one coming from the Zed camera) and the target pointcloud (the one provided by the STL file). Then we estimate a first rotation matrix that should theoretically allow us to obtain the target pointcloud starting from the source one, with the function called **execute_fast_global_registration.**

The next step we followed was giving this first approximated matrix to the function called icp_registration. This function iterates over the transformation passed to it and tries to refine it in order to find a more accurate transformation matrix.

Once this process is done, the rotation matrix is converted into an euler angle by an ad-hoc function. The yaw angle is the one that interests us hence we extract it. Position and orientation are then passed on to the motion planning module.

Motion Planning Part

To implement this part we based our work on Professor Luigi Palopoli's Matlab scripts, however we adjusted them to adapt them to C++ code and to our UR5 manipulator. To manipulate matrices and quaternions we used the **Eigen library**.

As far as concern the motion planning part we decided to operate mainly in the operational workspace and we considered the joint workspace to perform a smoother path, avoiding abrupt movements of the manipulator.

First of all, we implemented a function to compute the transformation matrix from frame *i* to frame *j*. Thanks to this we were able to perform direct kinematics and also the Jacobian computation. The latter two allow us to perform inverse differential kinematics to compute the path that the robot has to follow to achieve its tasks.

The **direct kinematics** is performed by first calculating all the transition matrices from a frame to its successor, and then they are multiplied to obtain the final transition matrix. Hence the final matrix is the following: $TransformationMatrix = (T10) * (T21) * (T32) * (T43) * (T54) * (T65)$.

We'd like to underline that we also perform a rototraslation about the x-axis of $\frac{\pi}{2}$ and the vector (0.5, 0.35, 1.75) to align the base frame to the world one (fixed frame transformation), and a π rotation around the z-axis of the gripper frame to align it with the world (current frame transformation).

The **Geometric Jacobian matrix** is computed using the cross-products rule. To obtain the right result firstly we compute the intermediate matrices as in the direct kinematics, then we compute all the transformations from world to frame *i*, in this way we can obtain all the z_i and p_i that we need to compute geometric Jacobian.

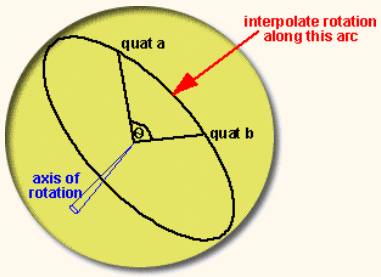
$$J = \begin{bmatrix} J_{p1} & J_{p2} & J_{p3} & J_{p4} & J_{p5} & J_{p6} \\ J_{O1} & J_{O2} & J_{O3} & J_{O4} & J_{O5} & J_{O6} \end{bmatrix}$$

Since all the joints are revolute, J_{Pi} and J_{Oi} are computed by:

$$J_{Pi} = z_{i-1} \times (P - p_{i-1})$$
$$J_{Oi} = z_{i-1}$$

The **path** the robot has to follow is created using **inverse differential kinematics control with quaternions**. We chose this solution to have a smoother trajectory since we can control each joint variable. To avoid the problem of inversion in a neighbourhood of a singularity we decided to implement the DLS inverse (Damped Least Square), this solution uses a K damping factor (equal to 0.0001), which is a positive definite matrix, that renders the inversion better conditioned from a numerical viewpoint. This factor is used to establish the relative weight between the two objectives. We also use the following correction factors: Kp=0.001 and Kq=0.001. Joint velocities are computed using DLS, and we use them to update the joint variables with the following formula: $js\_k = js\_k + (js\_dot\_k * dt)$.

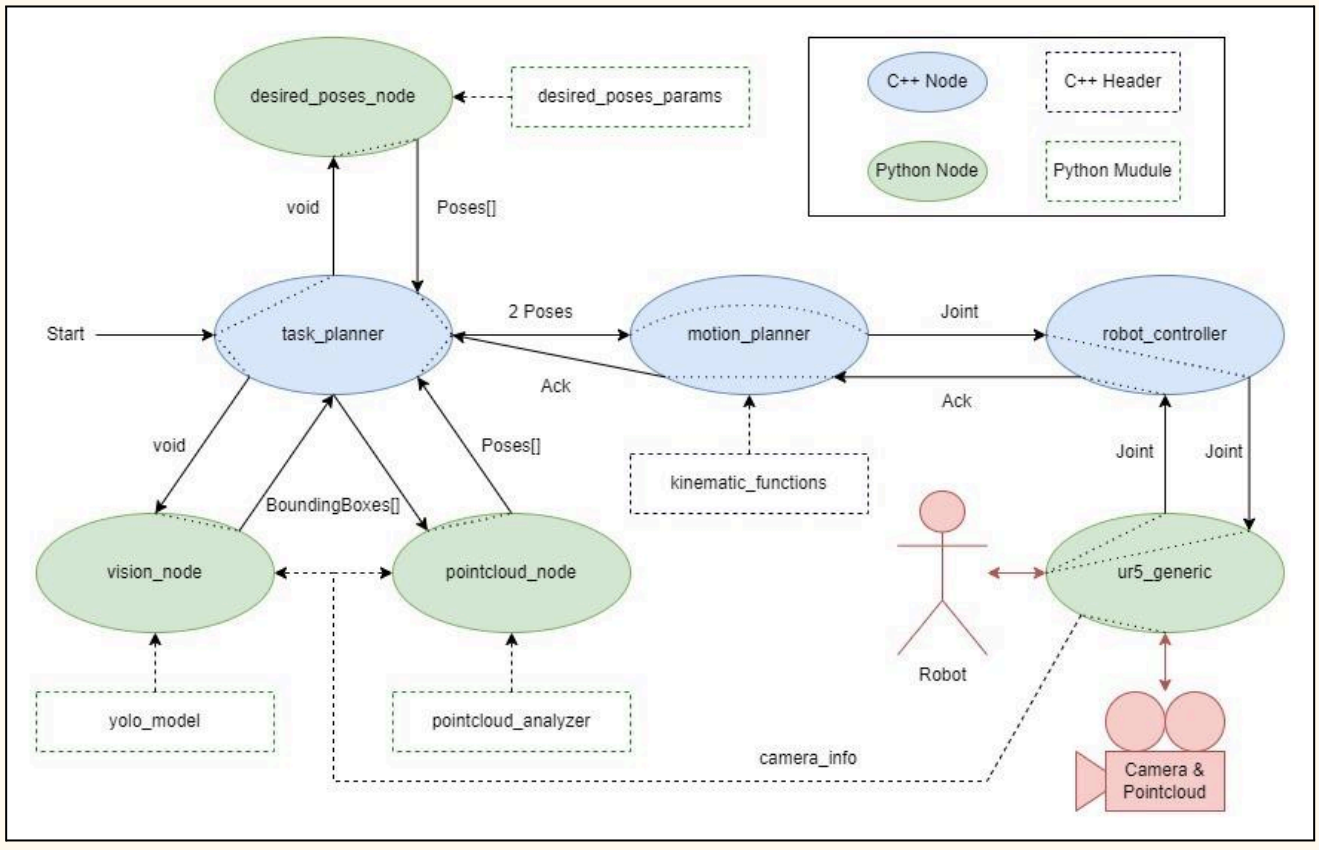
The interpolation between the actual quaternion and the desired one and between the actual position and the desired one is performed using **Slerp** and **Lerp** respectively. The effect is a roto-translation with a uniform angular and linear velocity around a fixed rotation axis. In fact, SLERP allows the quaternion to follow a *geodesic curve* on the spherical surface with constant angular speed, as we can see in the image.



ROS Configuration and Simulation Part

ROS, which stands for Robot Operating System, is an open-source middleware framework widely used in robotics research and development. Despite its name, ROS is not an operating system but rather a collection of software libraries and tools that help developers build robot applications. The main parts of ROS are:

- **Nodes**: ROS applications are composed of nodes. Each node performs a specific task.
- **Communication**: Nodes in ROS communicate with each other using a **publish-subscribe** messaging model. They can publish messages to topics and subscribe to topics to receive data.
- **Messages**: Messages are data structures used to exchange information between nodes.
- **Services**: ROS supports remote procedure calls through services. Nodes can offer services to perform specific tasks upon request from other nodes.
- **Master**: It's the coordination system that facilitates node discovery and communication.
- **Packages**: Packages encapsulate related functionality, such as drivers for specific hardware or algorithms for perception and navigation.



This part of the project interconnects everything explained in the previous pages. Deepening into the details we can analyze the chart above, which describes the messages' flow. First of all, the task planner requires the list of detected bounding boxes (BB) from the vision_node, which relies on the YOLO_model, which returns them. Then the task planner sends them to the pointcloud_node which queries pointcloud for each BB in order to return the pose of the mega-block. At this point, we are conscious about all the information of the mega-block detected except its final pose, obtained from desired_poses_node, which imports a python module defined by us containing the final destination of each block. Now, a search couples the desired and the actual

poses for each block, and sends them to the motion_planner, which uses kinematic_functions to compute the path the manipulator has to follow to achieve the goal, also considering some via point to perform an approach. The theta variables of each joint, just computed, are passed to the ur5_generic which has the final task of making the robot move. Such a move resulted a little bit sharp, so we decided to perform another interpolation in the joint space to smooth it. Sucha behavior is implemented thanks to the robot_controller node, which also performs the role of an interface to mask the access ur5_generic as a service, avoiding a direct access to it with messages.

The project contains a launch file to simultaneously run all the student defined nodes, but not ur5_generic. So, a bash script ur5_lego_run.sh has been created to execute them all and divide their outputs into different terminals opened in the background.