UNIVERSITY OF TRENTO

Department of Information Engineering and Computer Science

Dipartimento di Ingegneria e Scienza dell'Informazione:

Università degli studi di Trento

Anno formativo 2022/2023

Corso di laurea in Scienze e Tecnologie Informatiche
Fondamenti di Robotica

Titolo

"Project-1: Robotic Manipulator"

Gruppo

W

Degli allievi

Luca Cazzola, Dennis Cattoni, Sergio Brodesco

UNIVERSITÀ DI TRENTO

INDICE

1. Technique used for high-level planning

The design of the project follows the modularity paradigm: In fact each element in the architecture performs a specific task.
The main idea is to manage every communication between the vision and the motion plan with an intermediate master element.
Another key point is implementing most of our nodes in C++ language in order to have better performance at runtime.

1.1 *ROS Architecture*

The system architecture has 3 main ROS nodes: ***image_processor***, ***task_planner*** and ***motion_processor***.

The ***Image processor*** node is in charge of object detection, classification and pose estimation, it then sends the collected data to the task planner node.

After receiving the detection data, the ***task planner*** node decides which points to pass through to reach the blocks and move them in their final positions.
Those points are then sent to the motion processor node.

Finally, the ***motion processor*** node has the purpose of choosing the best configuration for the joints and the trajectory to reach it, while handling velocities and checking for collisions, joint limits and singularities.
After deciding the trajectory to follow, it then publishes the joint position for the real/simulated robot.

1.2 *ROS Topics and Messages*

To make the communication between the nodes possible, we defined two custom ROS message types:

- ***ImageData***, is a message published from the Image processor that contains the computer vision information.
  It is composed of:

  - An integer representing the class type of the detected object.
  - An array containing the position of the block with regards to the world frame.
  - A rotation matrix representing the object rotations.

- ***RobotInstructions***, is a message published from the task planner that contains the next point to reach and the diameter for closing/opening the gripper.
  It is composed of:

    - An integer representing the class type of the detected object.
    - An array containing the position where we want the end effector to move (not necessarily the block position).
    - A rotation matrix representing the rotation we want for the end effector.
    - A boolean to decide if we have to open/close the gripper.
    - And a float representing the gripper diameter (in case we have to move it).

Each message is published on a custom topic:

- "***/imageProcessor/processed_data***" that connects the Image processor (publisher) with the task planner (subscriber).

- "***/task_planner/robot_instructions***" that connects the task planner (publisher) to the motion processor (subscriber).

1.3 *Services*

To move the gripper on the real robot we decided to use a rosservice call to the "*move_gripper*" service defined in locosim, while in simulation we need to treat each gripper finger as an additional joint, transforming the diameter and appending it to the message we publish on the "***/ur5/joint_group_pos_controller/command***" topic.

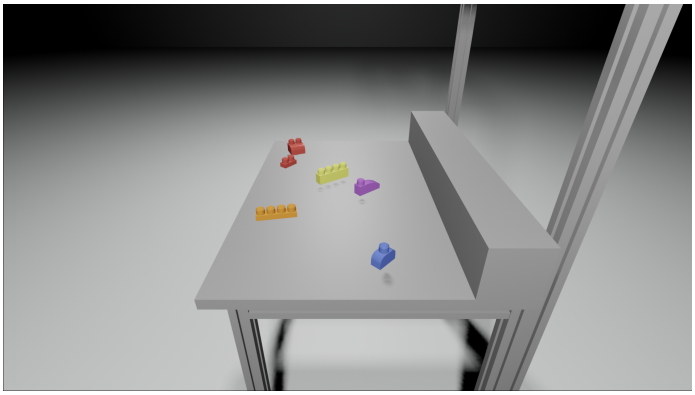
2. Technique used for perception

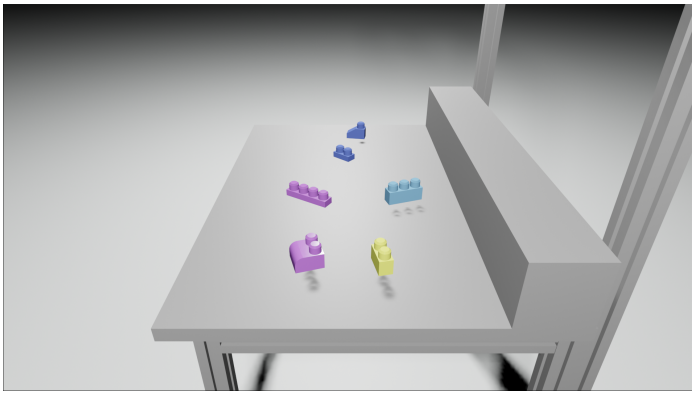
2.1 *Dataset generation*

Our goal is to detect and classify a total of 11 distinct classes, each representing a MegaBlock.

Thanks to the given .stl mesh files of the 11 blocks and the table mesh we managed to build a synthetic dataset through the **Blender** virtual environment.

Each object in a Blender scene can be edited via python scripting, so we've made a script that automatically displays the blocks in random quantity, position and rotation, making sure they don't intersect with themselves. A random plastic-like material (between 7 possible materials that have different colors) is also assigned to each object.





Thanks to 2 artificial lights we can also change the exposure of the scene. Last but not least we randomly change in a defined range the focal length of the camera to be more robust to camera changes in position and make a render of the scene.

To accomplish object detection and classification we also need labels for our images, those are fairly easy to obtain since we're in a controlled environment like Blender. What we need to establish is the labels' format, since it varies depending on the object detection model.

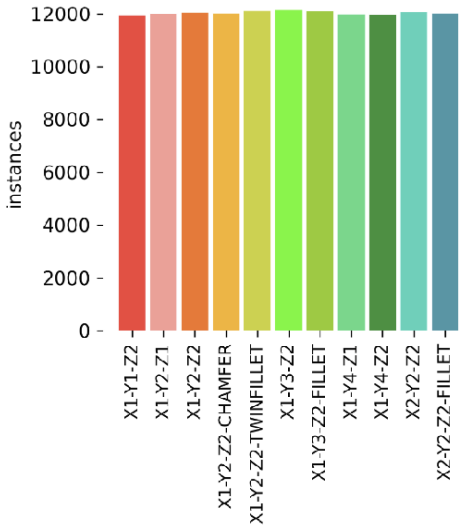
2.2 Model training

We've looked for a model from the ***Yolo*** family, since they're designed for real time detection, providing fast and accurate results.
We've chosen the ***Yolov5 model by Ultralytics*** https://github.com/ultralytics/yolov5 that is pretty recent and hasn't any significant compatibility issues with ROS 1.

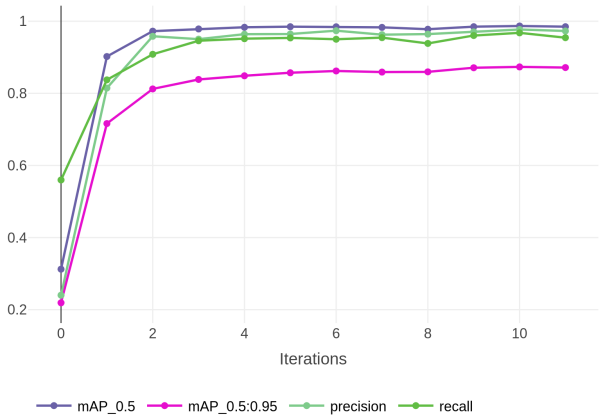
Since our dataset is totally synthetic we don't have many problems with limited data size, so following the guidelines of Yolov5 we've opted for a total of around 12'000 instances for each class, our actual dataset size is of around 20'000 images.

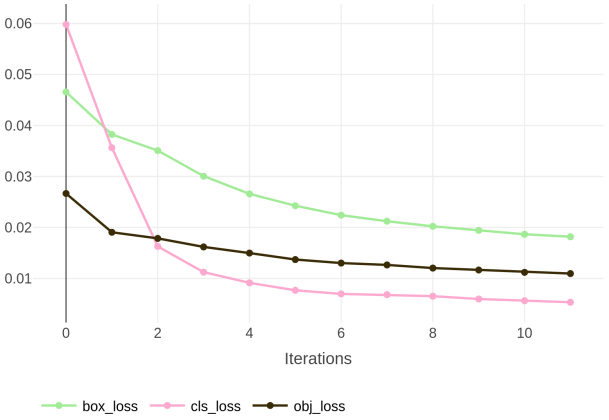
To validate our model we've made a validation set composed of **500 synthetic images** (generated as described above) and around **220 images obtained from the Zed camera** that we've manually labeled.
To monitor our training we've used the cloud environment of **ClearML**, that is fully integrated with Yolov5.

Class distribution :



Training statistics :





We've trained for a total of 11 epochs since our dataset is fairly large and chosen as best weights the ones obtained on epoch 10:

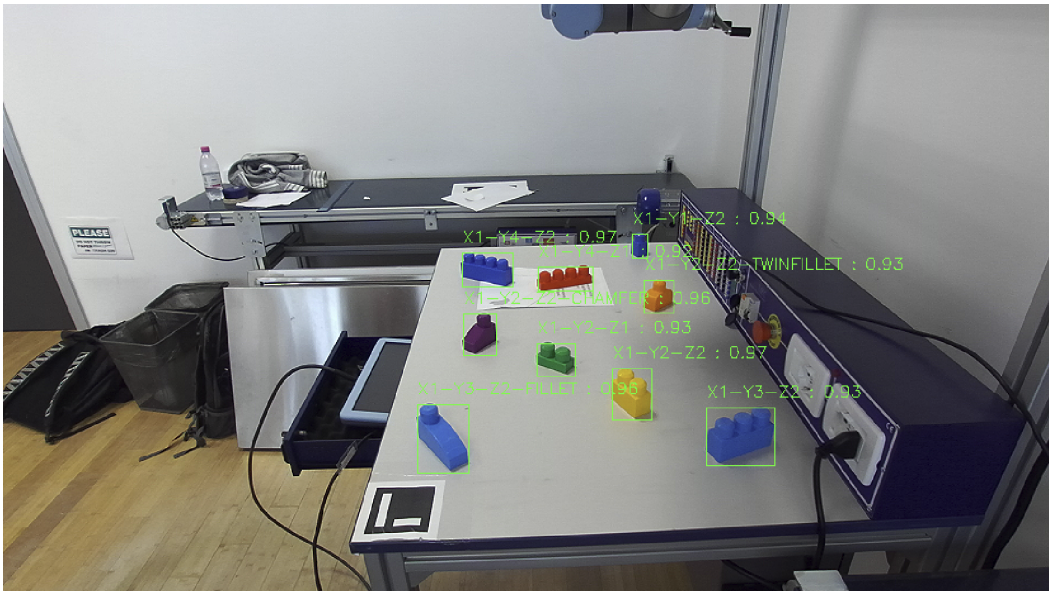
Precision	**0.977**
Recall	**0.968**
mAP_0.5	**0.987**
mAP_0.5:0.95	**0.873**

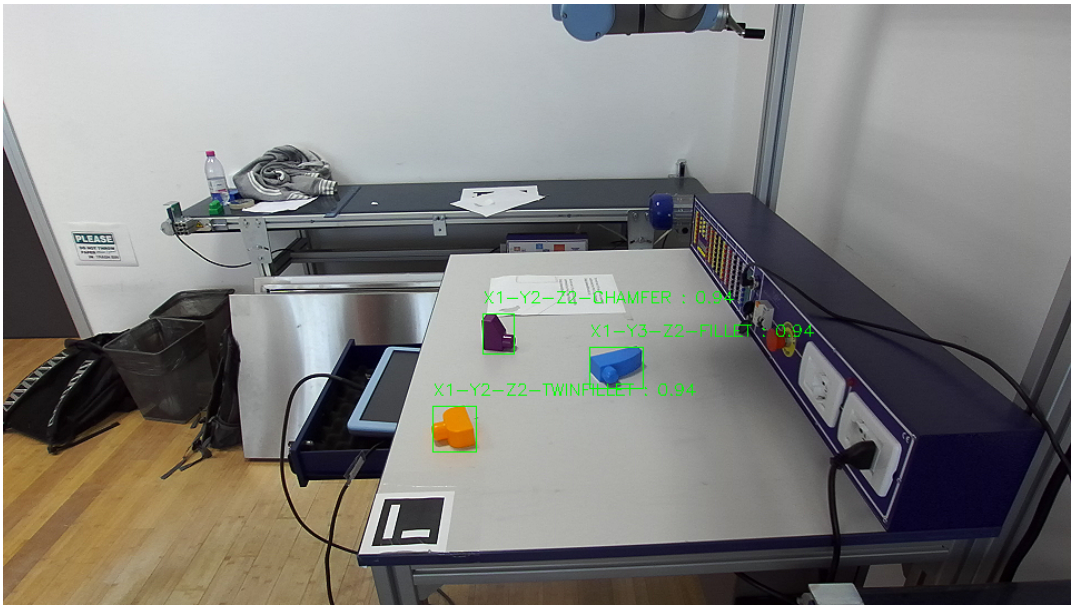
2.3. *Detection*

Now that we have our trained model we can simply run the detection on the image acquired from the virtual or the ZED camera.

To know where our object is located we can simply extract our point cloud and filter the points related to the object detected area in the image, while being aware that Nan values or points with an height < of 0.867 need to be ignored, since they're not valid or belong to the table (or something else).

Then we can get the location of our object in space by averaging all the points we've choosen.





2.4 *Pose estimation*

To estimate the pose of our object we used **point cloud registration** techniques which is pretty uncommon since our camera is stationary and we can't compare point clouds from different angles of the same scene…

The basic idea is that we can compare the point cloud obtained from the ZED camera with points sampled directly from the objects .stl files, that are used as a "second point cloud" to make the registration with.
Since the orientation of the object mesh is well known as it's always aligned with the world frame we can compute the transformation that best fits the 2 point clouds and use it as the object's rotation.

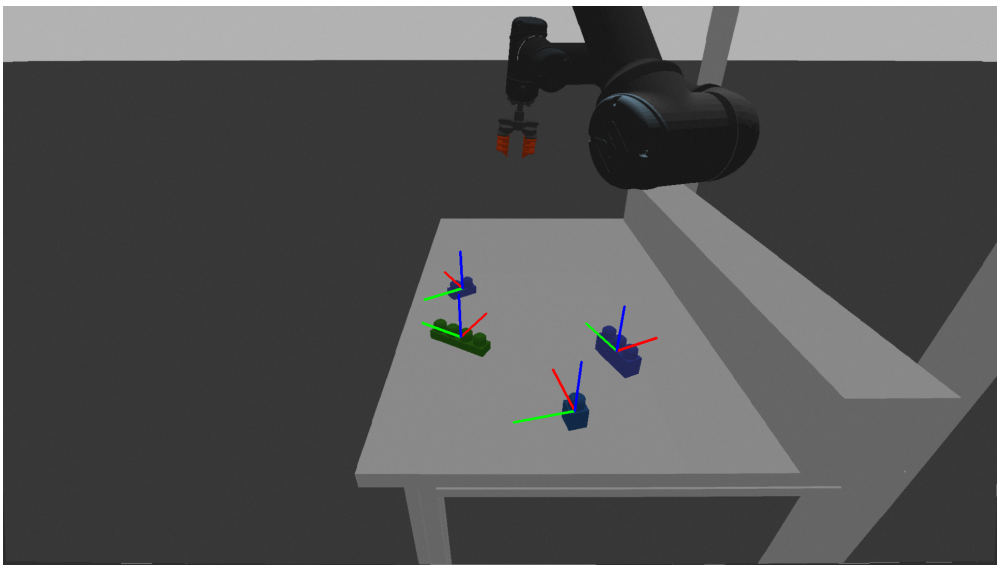
The point cloud registration is done in 3 major steps:

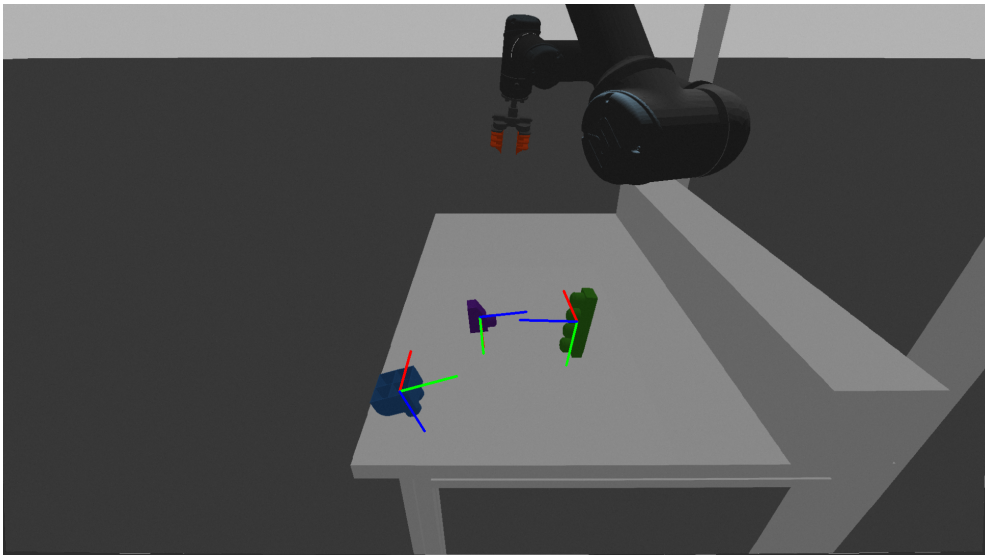
1. The 2 point clouds get downsampled, we estimate the normals for each point and calculate the **FPFH features**.
   FPFH features are basically descriptors of the surrounding area of each point. Those features compute an histogram which describes surface curvatures and shape variation.

2. We run **RANSAC** as global registration technique. At first we suppose the 2 point clouds are perfectly aligned, as convergence is reached we'll obtain an estimation about the transformation which best interpolates the 2 point clouds.

3. We then run **ICP** as local registration technique to refine the result obtained by RANSAC and make it even a better guess.

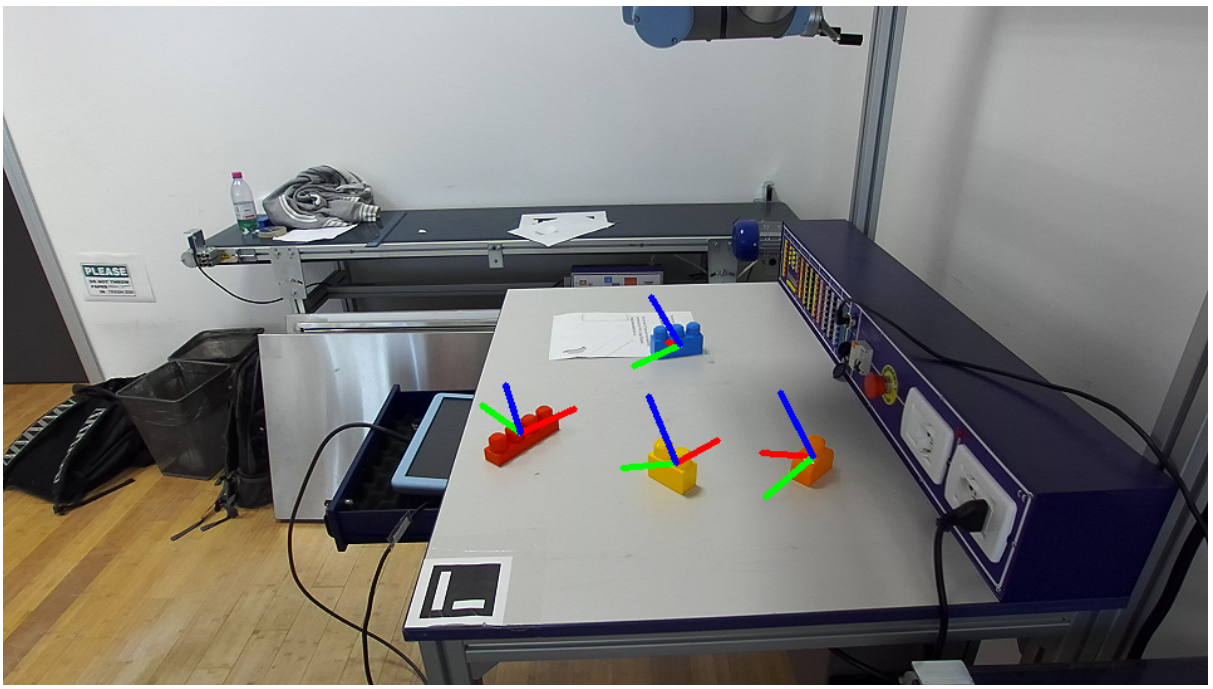
This process allows us to obtain precise information about the orientation of the blocks in space.





In the simulated environment results are nearly perfect even though it’s worth noticing that it’s not the fastest approach, as it takes at least **~2 seconds per object**.

During lab testing we couldn't obtain optimal results… Anyway we could only test for one day and surely there would have been room for improvement



We noticed as expected that the real world point cloud is much more noisy than the simulated one, and surely this adds difficulty to the problem.

Now that we have the object's position, orientation and class we can send an "ImageData" message to the task planner node.

3. Technique used for robot motion

3.1 *Task Planning*

After receiving the position of the block to pick up given by the detection, the task planner node decides a set of intermediate points to follow in order to avoid getting too close to the table and moving the other blocks.
The first position sent is just above the block, with the second one the robot moves down to grab it and with the third one it comes back up; the same is done for the destination of the block. //spiegare target point
Overall 6 positions are sent for each detected block, with the gripper moving when the end effector is close to the table to either grab or drop the object.

We decided to implement this on a different node than the motion processor so that we can change the set of points the robot has to pass through more easily.

3.2 *Best Configuration*

To select which joint configuration to use (obtained from the inverse kinematics) in order to reach a target point, we decided to start by sorting all the configs from the one with the minimum difference in overall joint angles, to the one that shifts them the most.

We also check if any configuration has acceptable joint angles (for example nan values) and if the target point is actually on the table (in case the detection goes wrong);
In the case one of those exceptions verifies, we proceed to set it as not valid so we don’t use it to generate a trajectory.

Once we have our valid configurations sorted, we see if the trajectory to reach the first configuration has any collisions with the table or the robot itself; if one exception is detected, we proceed to see if the next one is acceptable.

After a valid trajectory is found, we can start publishing the joint angles without checking the ones for the other final configurations and we start publishing the joint angles in order to reach the final point.

3.3 *Trajectory planning*

To obtain the robot trajectory, we decided to use a cubic polynomial function to describe the path the end effector should follow, knowing the joint positions and velocities at the start and at the end of the trajectory.

The trajectory time is obtained by considering the joint that needs to move the most among the 6 and using its velocity to estimate the minimum time needed to reach the goal point. Using a cubic polynomial allows us to have a smooth movement of the robot while avoiding spikes in velocity and acceleration.

We also added an additional control to correct the generated joint angles in order to not have values outside of their limits.

3.4 *Differential kinematics*

We decided to implement differential kinematics to make adjustments to our trajectory. This is done by computing the error in position between the values obtained from direct kinematics and the desired end effector position. The same process is done for the end effector rotations.

These errors are used together with the desired end effector velocity to get the velocity of the joints, using the inverse of the Jacobian matrix.
By integrating this velocity we can get the adjusted joint angles that consider the current error.

Since we use the inverse of the Jacobian, this process only works when the matrix is invertible, which is true outside of singularities; to handle this problem we add a damping factor to the elements of the Jacobian to maintain stability when we are near these critical points.

Another problem may occur when using Euler Angles: when applying a rotation of $\pi/2$ on the Y axis (using ZYX representation) we encounter a Gimbal Lock, and we can't properly translate from rotation matrix to Euler angles.

To deal with this issue, we computed the rotation error for the end effector using Angle Axis representation instead.

3.5 *Collision detection*

Each object we're interested to detect the collisions on is represented as a so-called *CollisionBox*, that is an element composed by: a translation vector, a rotation matrix, and a vector containing the size of the box.
When we need to check if a point in the robot frame goes inside any box the point is transformed into the CollisionBox frame (that is put at the center of the box) and we simply check if all its coordinates are within the box limits, if that's the case the point is for sure colliding with the box.

The points we're interested to evaluate are of course the ones related to the robot, to compute them we simply apply the transformation matrices used for the direct kinematic to gather the current position of each joint, then we displace a 14x14 cm square (set of four points) around each joint, that gets transposed by a fixed distance along the length of the joints, this gives us a set of squares that cover most of the joints.

We don't run collision detection for all the configuration of the trajectory, because it would slow the process too much, but still we run it every 10 configurations, which is more than enough to avoid collisions.

3.6 *Gripper*

The ur5 robot we worked with has 2 types of gripper: the 2 finger ***soft gripper*** which is used in simulation and the 3 finger ***rigid gripper*** that is used for the real robot.

To move the soft gripper in simulation we can treat the 2 fingers as additional joints for which we publish positions along with the other 6 of the ur5 robot.
In the case of the rigid gripper on the real robot we can only publish values for the six robot's joints, so in order to open or close the fingers we have to use the "*move_gripper*" service defined inside Locosim.

To handle the publishing of the joints and the gripper fingers we made a C++ class that initializes the needed publishers, differentiating between real robot and simulation and between the 2 or 3 finger gripper, and offers methods to send the desired joint values to the robot.
An object of this class is defined inside the motion processor node.

4. KPI's Table with measures from Gazebo

Assignment 1:

- KPI 1-1 time to detect the position of the object
- KPI 1-2 time to move the object between its initial and its final positions, counting from the instant in which both of them have been identified.

Assignment 2:

- KPI 2-1: Total time to move all the objects from their initial to their final positions.

Assignment 3:

- KPI 3-1: Total time to move all the objects from their initial to their final positions.

Assignment 4:

- KPI 4

KPI's Table	*1.1*	*1.2*	*2.1*	*3.1*	*4*
Assignment 1	~0.165sec	~22sec			
Assignment 2			~1.32min		
Assignment 3				/	
Assignment 4					/

5. Alternative Design

5.1 *Configuration sort criteria*

During testing, we tried another way to sort the configurations obtained from the inverse kinematics.
The idea of this approach is to give more importance to the movement of the links starting from the base to the end effector, because a small movement in the base link causes a movement in all the next joints in sequence.

In the end, we decided to only check the total joint movement because in most of our tests both methods chose the same configurations.

5.2 *Quintic polynomial Trajectory*

We tried implementing a quintic polynomial function to also control accelerations when computing the trajectories for the arm, but in the end we didn't find a good way to use these additional parameters effectively for our task's purpose.

Our final decision was to use a cubic polynomial in order to have less computational complexity.

5.3 *Trajectory with via points*

We also tried to compute the trajectories by adding a via point to pass through, this was to have more control over the end effector rotations needed for the third task.
The idea was to use those intermediate points to create more customized trajectories, but we had issues when deciding an appropriate velocity to have during the movement, and this often resulted in inappropriate joint angles.