Fundamentals of Robotics Project Report

Alessandro Nardin, Kuba Di Quattro, Francesco Martella
University of Trento, 2025

Contents

1 Vision and Localization

1.1 Introduction

The vision and localization system plays a crucial role in detecting Lego blocks within the workspace, estimating their positions and orientations, and relaying this information to the motion planning system. The implementation leverages a combination of YOLO-based detection and point cloud processing to achieve robust block recognition and precise pose estimation.

1.2 Architecture

To integrate the localization service with the rest of the project, we implemented it as a service. This design choice allows on-demand access to the results when necessary, while also preventing the node from running continuously, thereby improving overall performance.

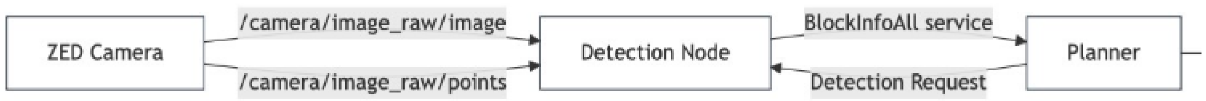


Figure 1: Localization Service Architecture

The node obtains the required inputs—an RGB image and a point cloud—by subscribing to two distinct topics. The RGB image is provided via the `/camera/image_raw/image` topic, while the point cloud data is supplied via the `/camera/image_raw/points` topic.

The node returns a list of all detected blocks, ordered by their distance from the camera (with the closest blocks first). To facilitate this, we define a custom interface, `BlockInfoAll`, with the following structure:

```
# Request
---
# Response
string[] block_names
float64[] positions_x
float64[] positions_y
float64[] positions_z
float64[] orientations_x
float64[] orientations_y
float64[] orientations_z
float64[] orientations_w
```

The request is empty, as no input data is required. The response, includes the block's type, position, and orientation.

1.3 Detection Pipeline

1.3.1 Step 1: YOLO Inference

The first phase employs YOLO for the initial block detection. When a detection request is received, the system follows these steps:

- Converts the ROS image message to OpenCV format.
- Performs inference using the YOLO model to classify and generate the bounding boxes for the blocks.

Details regarding the training process for the YOLO model are provided at the end of this report.

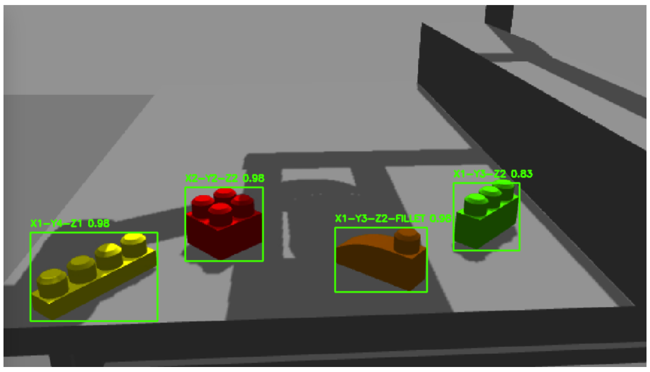


Figure 2: YOLO Inference Result

1.3.2 Step 2: Point Cloud Processing

After object detection, the system processes the corresponding point cloud data. The system extracts the relevant point cloud sections based on the bounding boxes identified during YOLO inference. Preprocessing techniques are then applied to filter out noise and remove outliers, ensuring cleaner and more accurate data for subsequent analysis.

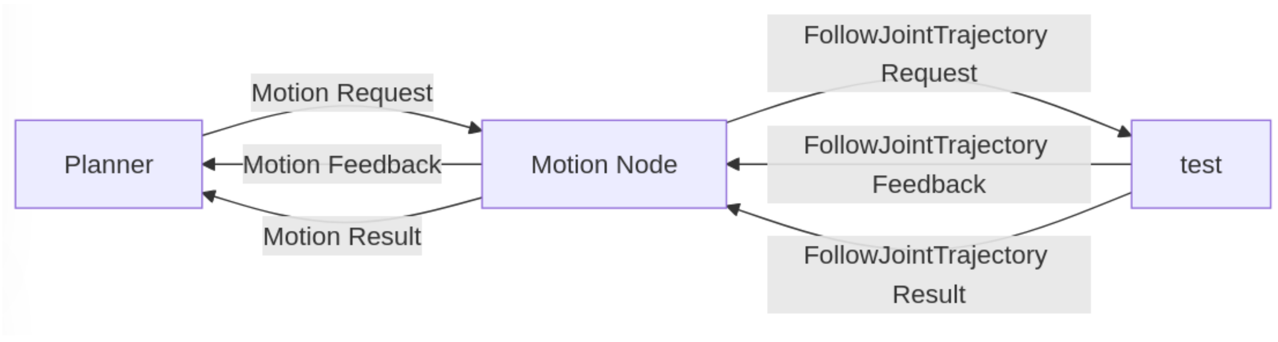
1.3.3 Step 3: Pose Estimation

For each detected object, the system loads the corresponding STL model for the detected class and generates a point cloud from this STL model, which serves as the source point cloud. The filtered camera point cloud is then used as the target. The Iterative Closest Point (ICP) algorithm is employed to perform registration, aligning the source point cloud to the target for precise pose estimation. More information about the ICP implementation are provided at the bottom of this report.

2 Trajectory Planning

2.1 Introduction

The trajectory planning node is responsible for computing optimal motion paths for the robot, ensuring smooth and efficient movement from the starting position to the target destination. It considers the robot's kinematic constraints, obstacle information, and environmental dynamics



FollowJointTrajectory Request
Motion Request
Planner
Motion Feedback
Motion Node
FollowJointTrajectory Feedback
test
Motion Result
FollowJointTrajectory Result

to generate feasible trajectories. By enabling precise motion planning, this node plays a crucial role in ensuring safe and effective robot navigation.

2.2 Architecture

To integrate the localization service with the rest of the project, we implemented it as an action. This approach allows the system to execute movements on demand while providing real-time feedback during execution. The node connects to the action server for the `Follow Joint Trajectory` action to perform the motion.

2.2.1 Motion Action Interface

The interface for calling the motion action is structured as follows:

```
# Request
float64 x
float64 y
float64 z
float64 r_x
float64 r_y
float64 r_z
float64 w
---
# Response
int32 error_code
int32 SUCCESSFUL = 0
int32 INVALID_GOAL = -1
int32 INVALID_JOINTS = -2
int32 OLD_HEADER_TIMESTAMP = -3
int32 PATH_TOLERANCE_VIOLATED = -4
int32 GOAL_TOLERANCE_VIOLATED = -5
int32 INVERSE_KINEMATICS_ERROR = -6
---
# Feedback
float64 x
float64 y
float64 z
float64 r_x
```

```
float64 r_y
float64 r_z
float64 w
```

Request: Specifies the desired position in XYZ coordinates along with a quaternion representing orientation.

Response: Provides an error code indicating whether the action was successfully completed or failed due to specific constraints.

Feedback: Returns real-time updates on the robot's current position during movement.

2.3 Trajectory Planning

The trajectory planning approach in this project follows a structured and efficient strategy based on the workspace's constraints. Given that the environment consists solely of blocks on a table, collision avoidance primarily involves these objects. Since the blocks have a relatively uniform height, the robot can execute motions in a predictable manner:

1. **Lift the end effector:** The robot first moves vertically to a predefined safe height above the blocks.
2. **Perform lateral movement:** While maintaining this height, the robot moves in a straight line to the target `x` and `y` coordinates while simultaneously adjusting its orientation.
3. **Descend to the target position:** Once aligned, the robot moves vertically down to the final `z` position.

This trajectory is computed using linear polynomials blended with parabolic curves to ensure smooth transitions in speed and position.

The initial positions are determined in Cartesian space using the table frame as a reference. After computing each intermediate waypoint, the coordinates are transformed into the robot's base frame, and the corresponding joint positions are calculated using inverse kinematics.

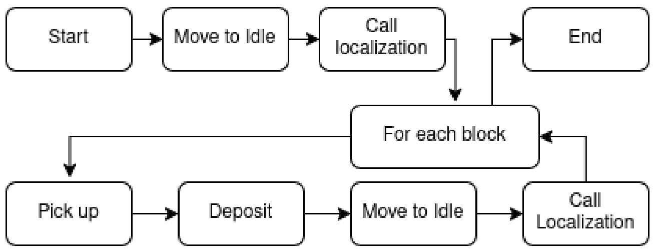
3 Planning

3.1 Introduction

The action planning module orchestrates the sequence of high-level operations required to execute the task efficiently. It ensures smooth coordination between perception, motion planning, and execution while adapting to dynamic conditions in the workspace. The process follows a structured, iterative approach to systematically detect, pick, and place blocks.

3.2 Execution Pipeline

The execution follows a looped sequence designed to handle multiple blocks while ensuring clear visibility and efficient movement.



1. **Move to Idle:** The robotic arm moves to a predefined idle position to clear the workspace and avoid obstructing the view.
2. **Localization:** The localization service is called to detect and retrieve a list of blocks, sorted by distance.
3. **Iteration for Each Block:**
   - **Move to Pick Position:** The robotic arm moves to the closest detected block's location.
   - **Pick Block:** The gripper securely grasps the block.
   - **Move to Deposit Position:** The arm transports the block to its designated placement location.
   - **Deposit Block:** The block is released onto the table.
   - **Move to Idle:** The arm returns to the idle position to ensure an unobstructed view for the next detection cycle.
   - **Localization Update:** The localization service is called again to get an updated view of the remaining blocks.
4. **End of Iteration:** The loop continues until all detected blocks have been processed.

Note: Each time the localization service is called, the updated list of detected blocks may differ from the previous one due to the movement of a block. However, this does not cause any issues. Since we always pick blocks in order of distance and move them closer to the camera, a block can only move up in the list. Because we always select the next entry in each iteration, we are guaranteed to pick the closest remaining block every time.

4 ICP Algorithm Implementation

The Iterative Closest Point (ICP) algorithm is implemented through several key stages to achieve precise alignment between the source (STL model) and target (camera-captured) point clouds.

4.1 1. Initial Pose Estimation

The algorithm begins by estimating an initial transformation between the source and target point clouds:

- Computes initial alignment using *Principal Component Analysis* (PCA).

- Calculates centroids of both point clouds.
- Estimates the initial rotation matrix using eigenvector alignment.
- Computes the initial translation vector between centroids.

4.2 2. Iterative Refinement Process

Once an initial alignment is established, the algorithm enters an iterative phase consisting of the following steps:

4.2.1 a) Point Correspondence

- Identifies corresponding points between the source and target clouds using **nearest neighbor search** (KD-tree).
- Applies 90th percentile filtering to remove outlier matches.

4.2.2 b) Transformation Computation

- Centers both point sets by subtracting their respective centroids.
- Applies **Singular Value Decomposition** (SVD) to compute the optimal rotation.
- Calculates the translation vector based on centroids and rotation.

4.2.3 c) Convergence Check

- Applies the current transformation and computes the alignment error using **Root Mean Square Error** (RMSE).
- Checks for convergence based on error improvement.
- Terminates if error reduction falls below a predefined tolerance.

4.3 3. Final Pose Extraction

Once convergence is achieved, the final transformation is extracted:

- Converts the final rotation matrix into quaternion representation.
- Outputs the final position vector and orientation.
- Provides the final alignment error metric.

5 YOLO Training

5.1 Introduction

Accurate object recognition is a fundamental component of our robotic system, as block identification is essential for precise relocation. To achieve this, we implemented a machine-learning

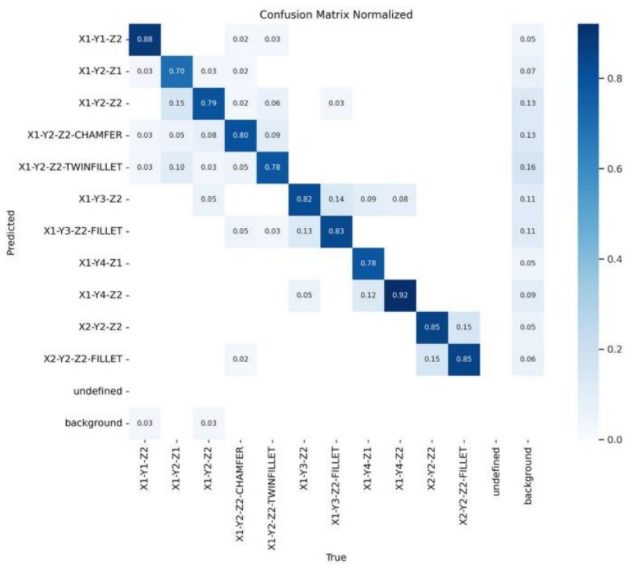


Figure 3: Confusion Matrix

approach utilizing YOLOv8, an advanced object detection framework developed by Ultralytics. YOLOv8 offers high accuracy, efficiency, and real-time detection capabilities, making it particularly suited for our application.

For training, we initially sourced a dataset from Roboflow. However, due to numerous undefined labels reducing its effectiveness, we supplemented it with additional images provided by Professor Sebe. This brought the dataset size to 1,500 images before augmentation.

5.2 Dataset Organization

To optimize training efficiency, the dataset was systematically divided into three subsets:

- **TRAIN (80%)** – Used for model training.
- **VALID (10%)** – Utilized for validation during training.
- **TEST (10%)** – Reserved for final performance evaluation.

Each image is accompanied by a label file in YOLO format (X_n, Y_n, Z_n), where annotations consist of normalized coordinates and class identifiers. This structured partitioning ensures balanced data distribution and improves generalization across various object appearances.

5.3 Data Augmentation Strategy

Given the dataset's limited size and missing labels, data augmentation was crucial for enhancing model performance. By artificially expanding the dataset to 5,500 images, augmentation improved model robustness and reduced overfitting.

5.3.1 Implementation of Augmentation Techniques

A dedicated script was developed to apply augmentation techniques while preserving label integrity. The primary techniques include:

- **Grayscale Conversion** – Reduces sensitivity to color variations while maintaining structural features.
- **Gaussian Blur** – Simulates out-of-focus conditions, improving adaptability to varying sharpness.
- **Brightness and Contrast Adjustment** – Enhances resilience to different lighting conditions.
- **Randomized Augmentation** – Randomly applies one to three transformations per image to increase dataset diversity.

All transformations preserved object positions, ensuring alignment between images and their respective labels.

5.4 Training and Fine-Tuning Methodology

The YOLOv8 model was trained using Google Colab, leveraging Nvidia GPUs for accelerated processing. Despite Colab's computational limitations, training was successfully completed over 150 epochs, spanning approximately three hours. To configure the dataset for training, a `data.yaml` file was created, specifying dataset paths and essential parameters.

5.5 Impact of Data Augmentation on Model Performance

The integration of data augmentation significantly improved model accuracy and robustness. Key benefits include:

- **Increased Dataset Volume** – Provided a broader set of training examples.
- **Reduced Overfitting** – Encouraged generalization by exposing the model to diverse variations.
- **Improved Object Recognition** – Enhanced detection capabilities across different lighting and sharpness conditions.
- **Enhanced Real-World Performance** – Ensured consistent recognition accuracy in dynamic environments.

By implementing these techniques, we successfully optimized the YOLOv8 model for accurate and reliable block detection.