Robotic Project

Luca Hardonk, Keith Leoni and Niccolò Lechthaler

Project Overview

Our project aims at picking up blocks from their starting positions and placing them in predefined final positions. It begins with *planning.py*, which integrates object detection from our custom-trained YOLO model. The model detects objects and extracts keypoints, which are then stored in an array of poses that the robot will follow.

Vision Module

The *pt_inference_node* utilizes our custom YOLO model to detect objects. It publishes an array containing the following information for each object: *[class, confidence, center_x_normalized, center_y_normalized, width_normalized, height_normalized]*. Using point cloud data the grasping frame is computed based on the object's center.

Motion Planning

Once the target object is identified the planning begins and an array of frames is computed. These frames are passed in pairs to the *move_a_to_b* server. The server sets the number of interpolation points between two poses and calls the *compute_path* server. The *compute_path* server performs SLERP (Spherical Linear Interpolation) between the two poses, generating smooth waypoints. To convert these waypoints into joint space, the *trajectory_service* is called. This service takes the SLERP-interpolated waypoints and applies cubic interpolation to connect the poses.

Inverse Kinematics and Execution

Inverse kinematics (IK) is then applied to determine the joint positions for each waypoint, ensuring precise movement. The *select_closest_one* function compares the eight possible IK solutions with previous joints positions, selecting the most suitable one to ensure smooth transitions while avoiding singular and invalid (containing NaN values) configurations. The IK solutions are retrieved via a dedicated service that prioritizes mid-range joints values for enhanced stability.
Finally, a *trajectory_msgs* message is generated and sent to the trajectory controller action provided by the UR5 official integration, which executes the motion in a simulated environment.

ROS2 SETUP

We created a ROS2 workspace and structured the code into several packages:

- **arm_bringup**: contains all the necessary launch files to start and coordinate the system, following ROS2 conventions.
- **collision_detection_pkg**: monitors potential collisions between the robotic arm and the table, rising warnings when a risk is detected.
- **custom_message_interfaces**: defines handy custom ROS2 message types that can be used across all packages in the workspace.
- **motion_pkg**: implements motion-related services for the robotic arm. These services are set up as servers, allowing asynchronous calls that improve execution flow and responsiveness.
- **planning_pkg**: handles motion planning by integrating services from other packages and incorporating object detection for precise movements.
- **pose_estimator_pkg**: estimates the position and orientation of objects detected by the trained model, forwarding the information to motion planning.
- **ros2_ur5_interface**: serves as the foundation of the project, containing the setup and essential configurations required to interface with the UR5 robotic arm, provided by Placido Flaqueto.

MOTION - motion_pkg

compute_path_service.cpp

This file contains the part responsible for the path planning utilizing the SLERP technique for robot motion in a ROS2 environment.

The code creates a server that computes an interpolated path between two poses and a publisher that visualizes the resulting trajectory in RViz. The primary goal is to compute smooth transitions from a start pose to an end pose by linearly interpolating the positions and using SLERP for the orientations.

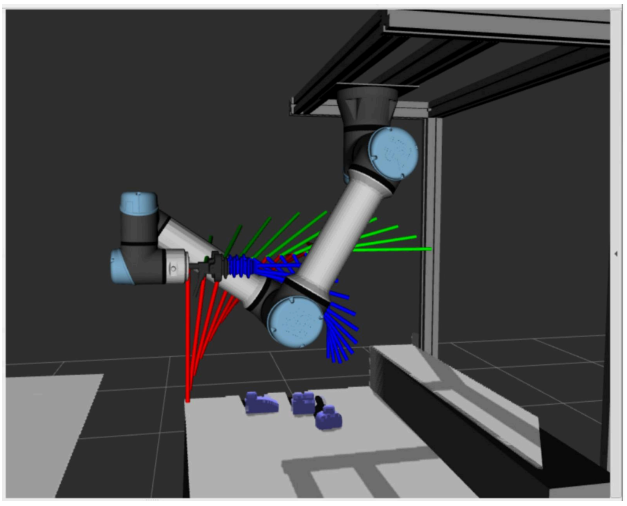
compute_trajectory_service.cpp

This code defines a ROS2 server that computes a joint trajectory for a robotic arm. It uses inverse kinematics (IK) to calculate joint angles for a series of target poses and generates a smooth trajectory through cubic interpolation. The process involves calculating waypoints from target poses, utilizing an IK service to determine joint configurations, and incorporating a function to detect and avoid singularities using the Jacobian matrix. The node provides a ROS2 service that computes and returns the trajectory based on the incoming pose data. Both position and velocity control are present in the robotic trajectory planning system, but the approach ultimately revolves around position control.

We decided on this approach since the task necessitates high positional accuracy and repeatability as it allows the system to precisely regulate the robot's end-effector position within a defined coordinate frame. In scenarios where the robot must dynamically adjust to external disturbances, variable resistances, or interact with unpredictable environments, velocity control enables real-time adjustment of the robot's movement based on feedback from the system's velocity and external forces.

direct_kin_server.cpp

This ROS2 service computes the direct kinematics of the UR5 robot, providing the end effector's position and orientation in response to incoming service requests. The kinematics are calculated using the Denavit-Hartenberg convention, with the results returned as a response containing the pose of the robot's end effector.

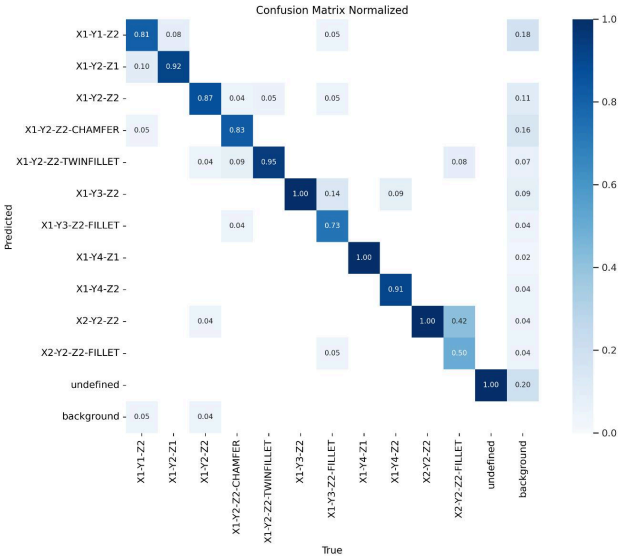


inverse_kin_server.cpp

This ROS2 service computes the IK. The service employs Eigen for efficient mathematical operations, including matrix transformations, and utilizes quaternions for orientation representation. Since the quaternion representation is used, it aids in avoiding issues such as gimbal lock, which is common when using Euler angles to describe rotations.

VISION - pose_estimator_package

In our project we used YOLO (You Only Look Once), a real-time and single-stage object detector that identifies and classifies objects in a single pass of the image. We used YOLOv11, the latest version, which provides better accuracy, speed, fewer parameters, improved feature extraction and more.
We used a labeled dataset created using Roboflow to train our YOLO model. For our training we used 400 epochs. From the results our model seems to be a little overtrained, however it performs well for our application.



pt_inference_node.py
This code defines a ROS2 node called *YoloDetectorNode* that uses a YOLOv11 model to perform object detection. The node subscribes to the */camera/image_raw/image* topic, where it receives the image from the camera. It then analyzes the images using the YOLO model. Since we are processing a single image at a time, *results[0]* contains all the detection results for that image, which includes bounding boxes, confidence scores, and class IDs for the detected objects.
The detection result with the highest confidence score is published to the */detection_result* topic as an array *[class, confidence, center_x_normalized, center_y_normalized, width_normalized, height_normalized]*. The values are normalized to be scaled easily between different image sizes.

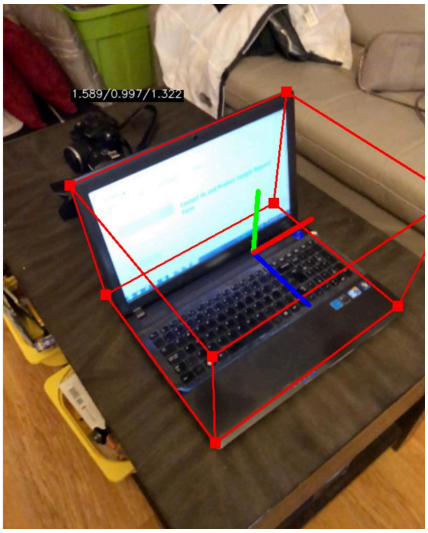
detection_result_republisher.py
This code defines a *DetectionResultRepublisher* node that subscribes to */detection_result* topic. It adds to the received detection a timestamp and frame ID called "*camera_rgb_frame*", and publishes the result on the */detection_result_stamped* topic. The timestamp is necessary for ICP to ensure synchronization between the detection and the point cloud.

pose_estimator.py
This code defines a node called *PoseEstimationNode* that subscribes to two topics: */detection_result_stamped* and */camera/image_raw/points*, which returns the pointcloud. The node takes the point cloud and detection result with the timestamp and it extracts the bounding box and scales the *x* and *y* center coordinates to the point cloud width and height. The scaled center coordinates are then passed to the point cloud to extract the distance from the camera. After this the node gets the transform (tf) from "*camera_rgb_frame*" to "*object_frame*" and from "*object_frame*" to "*grasping_frame*". It broadcasts the "*grasping_frame*" tf complete of its position and orientation expressed using a quaternion. This frame is the one the robot has to reach to grasp the object.

visualization_node.py
This code defines a node called *YoloOverlayNode* that subscribes to the two topics */camera/image_raw/image* and */detection_result*. It draws the box of the current best detection on the image and also adds a label specifying its

class and confidence score. It publishes the overlaid image on the */detection_overlay* topic. This is used to ensure that the robot is identifying an object.

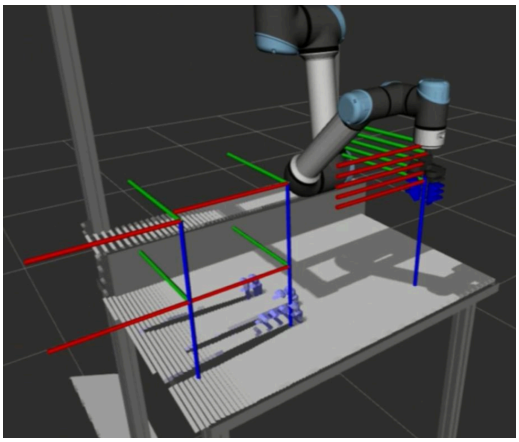
CenterPose - ISAAC Ros



CenterPose is a keypoint-based, single-stage approach for estimating the pose of objects at the category level. Using just a single RGB image, it processes unknown object instances within a recognized category. The pretrained model identifies 3D keypoint projections, estimates a 6-DoF pose, and predicts the relative dimensions of the 3D bounding cuboid.
The model uses a single-stage network for predictions and was trained on the Objectron dataset, which includes 15,000 video clips with each object annotated with a 3D bounding cuboid.

We were unable to proceed with this solution due to the difficulties encountered in training it with previously unknown objects and because it required a cumbersome setup process to integrate with the existing framework, along with the need for a separate Docker environment.

HIGH-LEVEL PLANNING - planning_pkg



We programmed the robotic arm to follow a precise path:

0. Start from initial position
1. Move to the safe plane
2. Move to the safe plane position above the object + open the gripper
3. Move to the grasping frame + close the gripper
4. Move to the safe plane position above the object
5. Move to the safe plane position above the goal position
6. Move to the goal position + open the gripper
7. Move to the safe plane position above the goal position

These steps are designed to ensure controlled movement and avoid collisions. By moving to and from a predefined safe plane the manipulator can approach objects and goal positions safely, minimizing the risk of collisions with the environment or with other objects laying on the desk.

planning.py
The *planning.py* handles a detection result, computes the sequence of poses for the robot to follow, and executes the planned poses while managing transformations and gripper operations. Here we also define the safe plane height.

This code defines a node *SafePlaneFramePublisher*, which publishes a pose array to the *planned_poses* topic and subscribes to the */detection_result* topic to receive detection results. It also subscribes to the *trajectory_done* topic to receive notifications when a trajectory is completed.
The node unsubscribes from */detection_result* after the first valid detection is received to avoid repeated triggers. It defines a pose array (that follows the steps described above) based on the detection result and publishes them on *planned_poses* for visualization. After this the node calls *move_to_next_pose* to plan the motion from one pose to the next in the planned sequence, it handles gripper operations (open/close) at specific transitions and calls the *MoveAB* service to execute the motion. It then waits for the *trajectory_done* notification to continue with the next planned poses.

move_a_to_b_server.py
This code defines a node called *MoveAToBServer* and it is designed to manage motion planning, from path and trajectory computation to execution, ensuring smooth, non-blocking operations by leveraging multi-threaded execution. It provides a *move_a_to_b* service to move a robotic arm from one pose (position and orientation) to another. The *move_a_to_b* service integrates several custom services and actions to compute paths and trajectories, which are then executed on the robot.
The node also sets up clients for the *ComputePath* and *ComputeTrajectory* services, as well as an action client for executing joint trajectories using the *FollowJointTrajectory* action.
When a request is received, the node calls the *compute_path* service to compute a path between the start and end poses. Once the path is computed, it calls the *compute_trajectory* service to generate a trajectory based on the computed path. The node then sends the computed trajectory to the action server for execution.
The node also includes a publisher to notify when the trajectory execution is done. It handles the result of the action execution and publishes a message indicating whether the execution succeeded or failed. The node uses a *MultiThreadedExecutor* to avoid deadlocks when blocking on futures and ensures smooth operation.
In summary, this node handles requests to move a robot from one point to another by computing a path, generating a trajectory, and executing the trajectory while managing the communication and coordination between different services and the action server.

block_frame_publisher.py
This code defines a *BlockFramePublisher* node. Its primary function is to transmit the final position and orientation of each block. These positions are determined based on the type of block. This node periodically broadcasts transformation data for three distinct blocks, with respect to the "base" frame.