Project Report

Martina Panini, Riccardo Randon, Carlotta Cazzolli

February 12, 2024

1 Structure of the project

This section presents the structure of the project, which aims to simulate a UR5 manipulator using ROS and Gazebo to recognize and pick up blocks on a scene.

There are two ROS nodes: vision and motion nodes. These nodes communicate through a service: the motion node requests the vision node for coordinates and the vision node provides them. Our original goal was to set up a vision node that could do block recognition and classification but instead we simulated a pick-and-place operation with hardcoded block positions, passed from the vision node to the motion node. The link between the two nodes is established through a service, "coordinate.srv", that specifies the type of message exchanged between motion and vision nodes.

The world folder contains the gazebo simulation world setup, hence the UR5, the blocks and the table on which they are placed.

The model folder contains the blocks that are going to appear in the world. We choose to use three pieces for our simulation.

The tests folder's purpose is to test the functionalities and functions independently.

1.1 Motion planning node

For the arm's motion, we relied on Inverse Differential Kinematics. The ur5Object.cpp file contains the implementation of the IDK as long with other functions that play a key role in the robot's motion, such as a direct kinematics function, a function to compute the Jacobian, trajectory planning with polynomial interpolation functions and the gripper management.

JointStatePublihser.cpp handles the functions necessary to publish the joint states information from the motion_planner.cpp to the ur5_generic.

1.2 Vision node

The vision node contains some hard coded blocks' coordinates in the world frame and uses a transformation matrix to translate them into the robot's frame. When the motion planner requests a block position, the vision planner node will send it a block position and an arbitrary final position where the block should be put.

We had to develop the project this way because we were not able to implement a fully functioning object detection and recognition node.

2 Motion planning

To move our robot we decided to use Inverse Differential Kinematics. In order to manage singularities, we used a damped least square inverse

The diagram above shows how the robot motion is implemented. We used the IDK_newVelocity to compute the joint velocities given the end effector's position. This function is called by a IDK_wFB function which performs the remaining computations. To find the desired position and orientation at each time frame we implemented a cubic polynomial trajectory. To move the gripper we also used a polynomial trajectory so that the finger opening would result smooth.

The motion planning receives the blocks coordinates from the vision node and, through the moveJoints and the moveGripper publishes said coordinates on the topics. ur5_generic reads the desired

```
├── model
│   └── X1-Y2-Z2
│       ├── mesh
│       │   └── X1-Y2-Z2.stl
│       ├── model.config
│       └── model.sdf
├── no-vision
│   ├── coco_dateset.ipynb
│   ├── config.json
│   ├── dataset_creation.ipynb
│   ├── KNN_classification.ipynb
│   └── model.ipynb
├── README.md
├── src
│   ├── CMakeLists.txt
│   └── package
│       ├── CMakeLists.txt
│       ├── include
│       │   └── package
│       │       ├── jointStatePublisher.h
│       │       └── ur5Object.h
│       ├── package.xml
│       ├── src
│       │   ├── motion_planning
│       │   │   ├── jointStatePublisher.cpp
│       │   │   ├── motion_planner.cpp
│       │   │   └── ur5Object.cpp
│       │   ├── tests
│       │   │   ├── testJointStatePublisher.cpp
│       │   │   ├── test_node.cpp
│       │   │   ├── testObject.cpp
│       │   │   └── testUr5Movement.cpp
│       │   └── vision
│       │       └── vision.py
│       └── srv
│           └── coordinates.srv
└── world
    └── blocks.world
```

Figure 1: Project's folder structure.

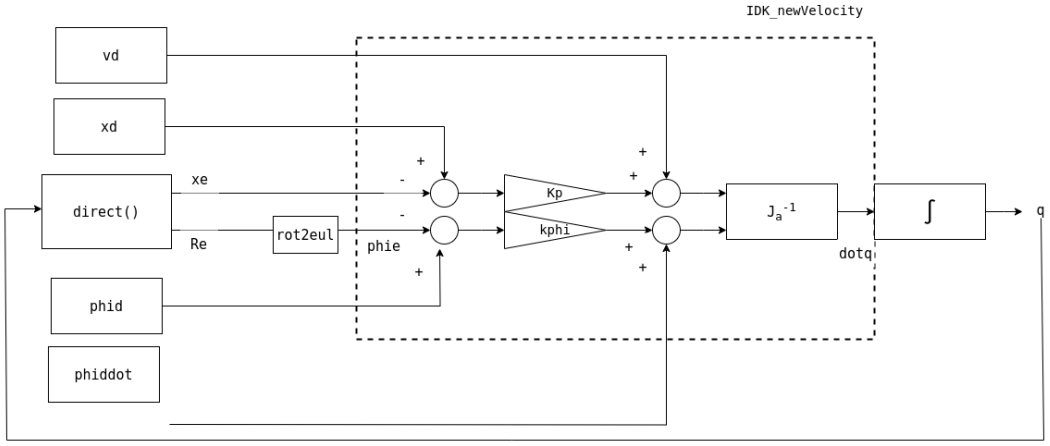

Figure 2: Inverse differential kinamatics with feedback scheme.

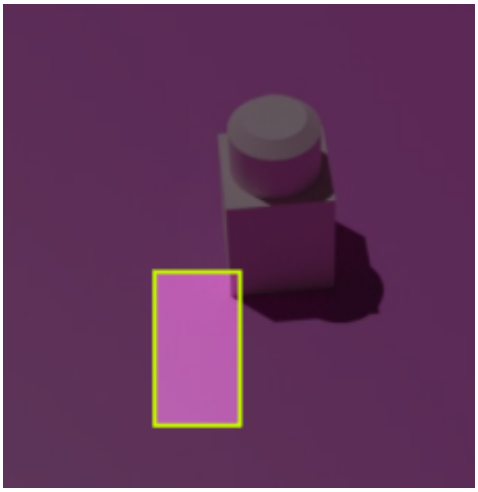

Figure 3: Wrong boxing.

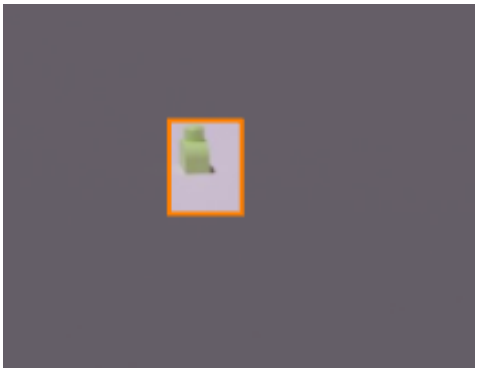

Figure 4: Right boxing.

position published on the topic and executes the movements: we chose to pass it 1000 positions per second.

3 Vision

Note: we could not finish the vision-based part of the project. Hereafter we breakdown what we've been able to accomplish.

Our idea was to use a pre-trained YOLO model to do the bounding boxes around the blocks. To do that, we needed to fine tune the pretrained model on our lego blocks dataset. We had problems converting the given dataset into a Coco dataset. The problem consisted in the fact that we were not able to understand which block coordinates the numbers in the .json file "bbox" tag referred to. Every combination we tried seemed to provide a non-systematic error (some solutions were better than others but there were some major boxing errors anyways). As you can see, Fig.4 shows a decently-boxed block while Fig.3 is a widely wrong one and the same processing rules were used.

Nevertheless, we proceeded into trying to fine tune the model, obviously expecting scarce results due to the poor dataset's boxes. As we predicted, the model was not able to clearly detect and box the blocks in the image. To complicate the problem even further, our computers were not able to satisfy the CPU requirements by the model fine tuning. For the object classification task of the project, we trained a KNN on the different lego pieces in order to be able to distinguish them later.

In order to be able to use what we have implemented in the motion node, we created a simplified vision node with some hard coded blocks positions. This is not ideal because part of the project was based on object recognition and classification, but we wanted to demonstrate that our robot could do a pick and place task.

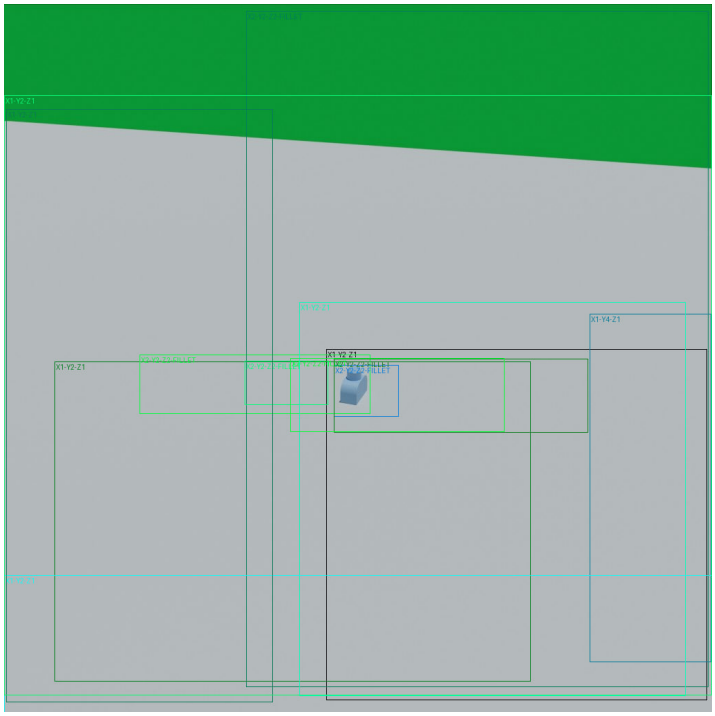

Figure 5: Model's prediction.

In the implemented vision node, there are 3 blocks' coordinates and 3 final positions. The motion planner sends a blank request to the vision node for block coordinates and the vision node selects the first block in the list, performs a coordinate conversion from the world frame to the robot's frame and sends back the block actual position, its final position and a boolean flag that informs the motion planner whether there are more blocks to pick up and move.

4 Grasping operation

To implement the grasping operation, we use the functions moveJoints and moveGripper. First, we move the robot to the homing position and we save that position as the "last position visited". Then the vision node will send the first block's coordinates: after receiving these data we perform eight subsequent operations. First of all, we move the end effector 20cm above the block, then we lower the end effector to the blocks height. The next movement consists in pinching the block, calling the moveGripper function. Once the block has been picked up, we move the manipulator to the block's final position using the same logic as before. We then save the last end effector's position so that the manipulator won't go back to the home position before moving to the next block.

5 Future developements

This project's future developments mainly focus on integrating the object recognition and classification. In particular, we will need to fix the bounding boxes problem and, subsequently, this should fix the model's prediction. With this improvements, we should be able to integrate the classification model to the recognition one and not only draw a reliable bounding box but also recognize which block the camera is seeing.

There are different reasons why we were not able to finish this project (lack of skills and time, lack of powerful computers to run smooth and quick simulations and train models, loss of a team member) but we tried to implement the more functionalities we could.