UNIVERSITÀ
DI TRENTO

UR5 Robotics Project Report

Oral presentation exam date: 14/02/2024

by

Leonardo Pasquato
Alessia Rinaldi

Course teachers:

Luigi Palopoli
Michele Focchi
Niculae Sebe

Information, Electronics and Communication Engineering
University of Trento

Contents

1 Introduction

The main purpose of the project is to simulating a grab-and-release operation by using an UR5 and ZedCamera. All the project is simulated on Gazebo by using ROS and LOCOSIM frameworks.

The simulation begins by showing the UR5, the table on which it is mounted and the ZedCamera. Then, 10 different Lego-like blocks should be placed on the table surface, and the ZedCamera should recognize and localize them. After getting the classification and localization of every object, the UR5 should grasp each block one by one and place them at predefined positions.

This project is created using the Catkin build system. The *ur5_project* folder is a Catkin package that can be compiled and runned indipendently from other packages. However, this project can only be executed when another ROS master is active, in particular the one launched by the *ur5_generic.py* of the LOCOSIM didattic framework.

The project is structured with three main ROS nodes, each placed in its respective file:

- ur5_project/src/objects/objects.cpp: the node in this file is used to spawn all the objects in known positions using the ROS Service gazebo/spawn_sdf_model;
- ur5_project/src/main.cpp: this is the "control" node that coordinates all motion and acquisition. It is the responsible for moving the UR5 and receiving messages from the vision node;
- ur5_project/scripts/vision.py: this node is responsible for detecting, recognizing and locating all objects placed on the table. It acquires data from the ZedCamera and it sends the recognized objects to the main node using the custom service VisionResults.srv.

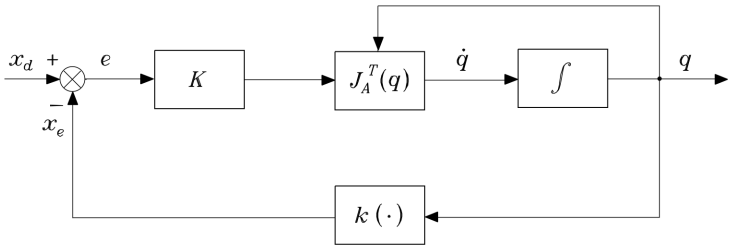
In order to have structure the project more effectively, all main motion functions are divided across various files, and the node passes through this files by being requested as a reference parameter.
Motion functions are all stored in UR5.cpp, and most of them require functions from other folders, like trajectory (Move_trajectory.cpp and Move_linear.cpp) and Stay_away_from. This last class is utilized to maintain a safe distance between the UR5 and critical points, such as table borders and singularities.

Among these files there is vision_receiver.cpp which is the most indipendent one, because it is responsible only for acting as a client to the VisionResults service. When it makes a request to the service, vision.py node respondes by sending the ObjectPose elements, and the vision_receiver.cpp must receive them. After receiving it returns all the objects to the main node.

The responsible for the high-level planning is the main code in main.cpp, which extracts the starting position of the UR5 and its target position. By using certain computations, it interpolates a cubic trajectory between checkpoints that are found in order to avoid critical zones, such as table borders or singularities.

2 Motion control

Most of the effort in Motion control was spent in order to achieve a movement of the UR5 between blocks as smooth as possible. The first challenge of the motion control consists in finding an algorithm for inverse kinematics: in this project, a simple one is used:



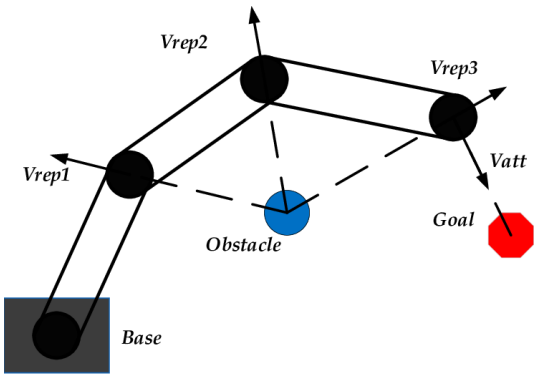
This algorithm uses the Analytical Jacobian and its inverse, in order to obtain joint velocities. After obtaining joint velocities, they are integrated using the Euler integration method:

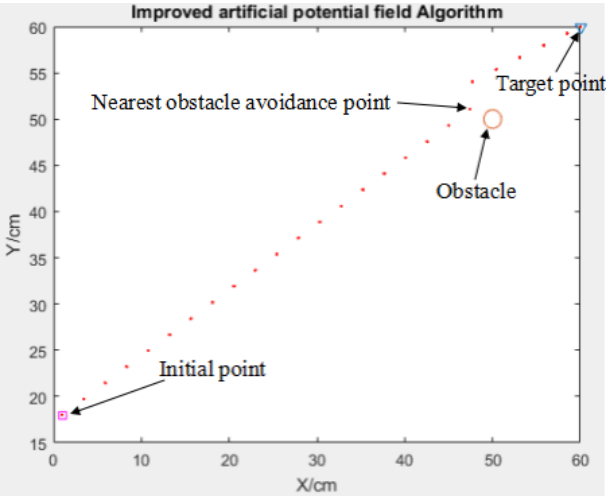
$$q(t) = \int_0^t q'(\tau)d\tau + q(0) \Longrightarrow q(t_{k+1}) = q(t_k) + q'(t_k)\Delta t \tag{1}$$

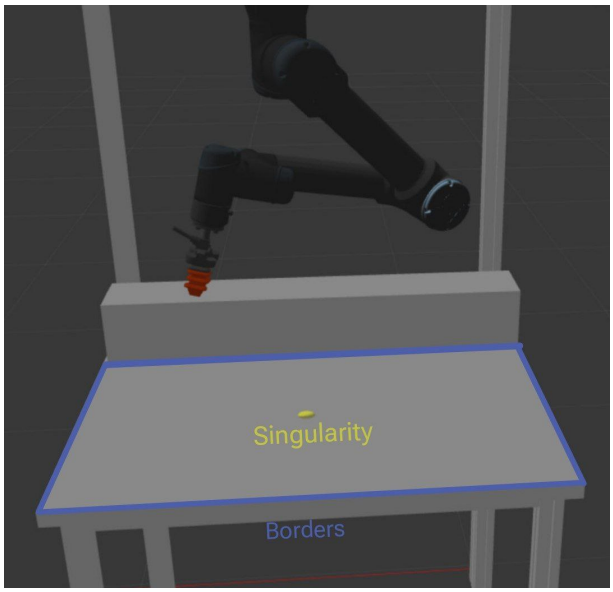
On one hand, this implementation guarantees simple and efficient computation; however, it still admits **operarational errors**. This error is not a significant issue, but due to this there are many "sperimental" offsets along with the kinematics computation.

All movements of the UR5 are divided in vertical and horizontal motion in order to avoid collisions with other blocks: from a starting position (which could be any position in the workspace), the UR5 moves upon a desired block by first executing a vertical motion and then a horizontal motion. The first motion is necessary to position the end-effector at the correct height. Then, the horizontal motion moves the end-effector towards the desired block (or a generic position).

The horizontal motion is the critical one because it needs to be computed while avoiding table borders and singularities.
In 'Singularities' and 'Borders', an algorithm named "Gradient Descent" is used to compute the gradient of a scalar field with respect to the position of a point within the simulated environment. The algorithm updates the position of the point to move in the opposite direction of the gradient. By doing so, the value of the scalar field gradually decreases until reaching a local (or global) minimum. The position of the point iteratively updates until a convergence condition is met. In simpler terms, this algorithm finds non-passing checkpoints for critical areas such as borders and singularities.







After obtaining some checkpoints that describe the trajectory, a *point-to-point* movement must be implemented: between each conseguent couple of checkpoints, a cubic polynomial interpolation is computed in order to achieve a smooth movement.

$$p(t) = a_0 + a_1 \cdot t + a_2 \cdot t^2 + a_3 \cdot t^3 \tag{2}$$

By knowing the starting position p_{start}, starting velocity v_{start}, target position p_{target}, target velocity v_{target}, starting and ending times t_{start} and t_{target}, the resulting values for each coefficient a_n are the following:

$$a_0 = p_{start} \tag{3}$$

$$a_1 = v_{start} \tag{4}$$

$$a_3 = \frac{p_{target} - \frac{t_{target} v_{target}}{2} + \frac{v_{start} t_{target}}{2} - v_{start} t_{target} - p_{start}}{t_{target}^3 - \frac{3}{2} t_{target}^3} \tag{5}$$

$$a_2 = \frac{v_{target} - 3a_3 t_{target}^2 - v_{start}}{2 t_{target}} \tag{6}$$

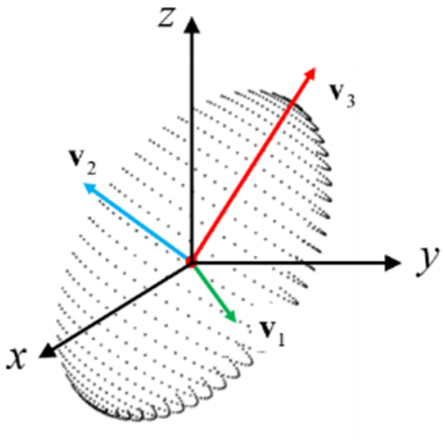
3 Robot vision

For this part of the project, training a neural network is the starting step. It is possible to train YOLO v8 on Colab, download the neural network weights, which will then be used to create a pre-trained YOLO instance in vision.py. To train a neural network, a custom dataset has been created using 'Roboflow'.

The code dedicated to Robot Vision is divided into two parts:
vision_server.py written in Python code and vision_receiver written in C++ code.

3.1 vision_server.py

This code is designed to run on a ROS node that receives RGB images and point clouds from the ZedCamera. When new RGB images and point clouds are received, callback procedures are triggered.
Thanks to rgb_callback, the RGB image is converted into a format usable by OpenCV. Then, the YOLO model is used to detect objects present in the image. Instead, in pc_callback, the point cloud is examined to recognize objects within the three-dimensional scene. The rotation of objects is determined using Principal Component Analysis (commonly called PCA) and eigenvectors.
PCA can be employed to derive the size of dispersion for each dimension as well as its orientation for datasets spanning multiple dimensions. Within machine learning and various other domains, PCA finds utility in dimensionality reduction, where only the dimensions that effectively capture the data's features are retained.
Eigenvectors are used in order to obtain the inertial ellipsoid of the object: the inertial ellipsoid is described by the eigenvectors, which represent its principal axes. Eigenvectors estimate the potential rotation that the object can achieve. This photo can explain better this notion:



The case of the project differs from this photo, but the principle is the same: each block is placed on the table and it's possible to compute the eigenvalues and eigenvectors of each of them. The smaller the eigenvalue is, the smaller the corrisponding eigenvector is. The eigenvector corresponding to the smallest eigenvalue is the normal vector of the point cloud data and it coincides with the yaw angle.

After extracting information about the objects from both the RGB images and the point cloud, this data is utilized to create a ROS service named "vision". This service provides the positions and rotations of detected objects within the scene.

3.2 vision_receiver

This C++ code introduces a vision_client function that serves as a client for the ROS service named "vision" (defined within the Python code). Upon invocation, the vision_client function initializes a client for the vision service using a 'ros::ServiceClient object'. The client awaits the availability of the vision service by using the 'waitForExistence()' function. Upon readiness, the client triggers the vision service

by utilizing the 'call()' function. If the service call is successful, the function processes the obtained response, converting object names extracted from the response and saving the positions and rotations of detected objects in a vector.
Finally, the function returns the vector containing the positions and rotations of detected objects.

4 High level planning

4.1 Generating objects in Gazebo

The first thing to do after launching the UR5_generic simulation, is to place all the objects on the table. The objects.cpp code has this responsability. It knows the path for each model and it spawns them in the simulation by sending messages to *gazebo/spawn_sdf_model*.
Although, in the code there are random positions for each objects, but there is not a true randomizer for them. A good improvement would be to implement a randomizer, but it could be critical for the vision node.

4.2 Obtaining vision data

After initializing the main ROS node, the highest priority task to achieve is the data acquisition from the Camera. The camera should be able to capture images and point clouds of the table where blocks are placed. To achieve this, it's important that the robot is not in the field of view of Camera. Therefore, first, the UR5 moves to a position far away from the Camera and then the Camera begins its acquisition.

After obtaining and processing the data, the main node receives from the vision node all the charateristics of each object: label, position, yaw angle and name which nominates the block in the Gazebo simulation.

4.3 Object grasping and repetition

After the acquisition of these information, the UR5 is allowed to start its job. It moves to the first recognized object, it rotates the gripper to the object orientation and then it catches the block and brings it to a predefined position on the other side of the table.

After the completation of the first operation, it lifts up and restarts all the motion. The UR5 is allowed to move vertically only when the end-effector is placed over the desired block or its final position. Between these two positions a smooth interpolation is created: first, the generation of checkpoints in the horizontal trajectory, then cubic interpolation between each checkpoint. Checkpoints are created in order to mantain a safety distance from obstacles, as explained in the Motion Control chapter.

References

- **Robotics: Modelling, Planning and Control** by Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, Giuseppe Oriolo
- https://medium.com/@hirok4/analysis-of-3d-point-cloud-orientation-using-principal-component-analy
- https://www.politesi.polimi.it/retrieve/a81cb05a-3dfa-616b-e053-1605fe0a889a/2012_12_Ciolino.pdf
- https://www.researchgate.net