UNIVERSITÀ DI TRENTO
Dipartimento di Ingegneria e Scienza dell'Informazione
Introduction to Fundamental of Robotics

PROJECT REPORT

UR5 Pick and Place

Prepared for:
Focchi Michele
Palopoli Luigi
Sebe Niculae

Prepared by:
Fumagalli Gabriel
Gore Stefan
Vigasio Simone

a. a. 2023/24

CONTENTS

INTRODUCTION

Structure of the project:

This project consists of two **ROS nodes**. One is specifically designed for **object detection** of the bricks on the table, and it calculates their **localization** and **orientation**. The other node is focused on **robot motion** and **task planning**.

```
├── CMakeLists.txt
├── include
│  └── robotic_project
│      └── kinematics.h
├── models
├── msg
│  └── BrickPose.msg
├── package.xml
├── src
│  └── motion
│      ├── Eigen
│      ├── kinematics.cpp
│      └── main.cpp <-- robot motion + task planning node
├── srv
│  └── ObtainBrickPose.srv
└── vision
     ├── best.pt
     ├── vision.py <-- vision node
     └── yolo 5
```

The link between the two nodes is established through a service. The "**vision.py**" file functions as the **server**, while the "**main.cpp**" file acts as the **client**, tasked with patiently waiting until the brick detection is complete. The "**ObtainBrickPose.srv**" file specifies the type of message that is exchanged between these two node:

In its core functionality, the service provides two arrays containing poses and names, with each set corresponding to a brick, and an integer indicating the structure's length. The "p" element comprises three double variables defining the location and another double specifying the smaller yaw angle.

```
geometry_msgs/Pose[] p
string[] name
int32 length
```

The "**kinematics.cpp**" file houses all the functions utilized for **robot motion** and **task planning**, and its associated header file is located within the "include/robotic_project/" directory.

Robot motion:

The **type of robot motion** we decide to implement requires the use of **quaternions**. The goal of this decision is to **avoid** the **representation singularities** and to have a more fluent motion, even if we still use linear interpolation.

Computer vision:

Computer vision is the set of functions which are able to understand the **location** and the **orientation** of the various **bricks** on the table. The only needs **required** are a **point cloud** and a **model trained on object detection**, generating their boundary boxes.
Once it gets all these information, they are sent to the high-level planning code.

High-level planning:

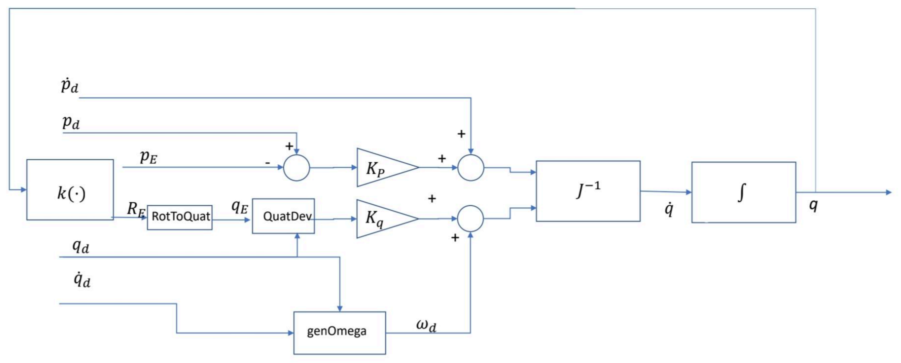
The **high-level planning** is the last task of the process. Thanks to the robot motion functions and the information relating the bricks on the table, received by computer vision, the high-level planning **gives** us the complete **path the UR5 has to follow**. The path is in the **joint space** and it also contains the **grasping** and **detaching** of the various bricks.

ROBOT MOTION

Choice of implementation:

For **robot motion**, we opted to **employ quaternions** to **circumvent** the challenges associated with **representation singularities** arising from the manipulation of row-pitch-yaw angles. While this approach addresses many issues, it **doesn't** entirely **eliminate** the problem of **effective singularities**. To overcome this limitation, we implemented a precise task-planning algorithm.

Focusing on robot motion, our work concerned about the implementation of this schema:



Utilizing Professor Palopoli's Matlab code, we developed the necessary functionality for planning the robot's linear motion trajectory. In detail, our decisions included employing the Eigen library for data structure manipulation, specifically for quaternions and matrices.

The core functions we implemented are based on the use of **Differential Kinematics**. For this reason we needed a function of **linear interpolation** between 3D points, **SLERP** function between quaternions and obviously **Direct Kinematics**. In all these implementations we use a **discrete time** with $\Delta time = 0.1 sec$.

For **correction factors**, we set **Kp** and **Kq** to 10 for linear and 1 for quaternion motion, respectively.

Pseudoinverse Jacobian:

As **Jacobian** we chose to implement the **geometric** one.
Once we have the Jacobian, to avoid representation singularities we **compute** the **pseudoinverse Jacobian** choosing the **dump correction**:

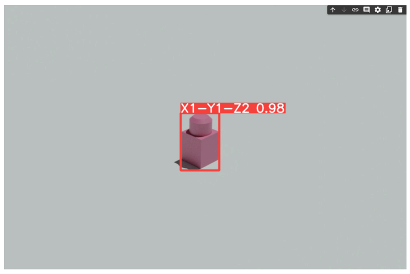
$$J^{\dagger} = J^{T}\left(J^{T}J + dump \cdot I\right)^{-1}$$

This decision was motivated by its **ability to prevent the robot from crashing** when the planned motion is **close to a singular configuration**, even if it deviates from the original trajectory.

COMPUTER VISION

Perception:

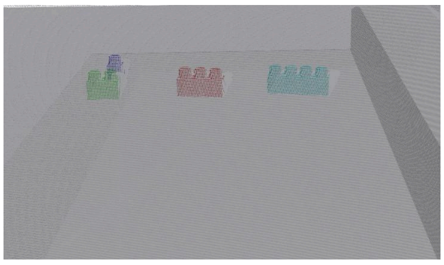
To identify objects, we leverage both **Image** and **PointCloud** data structures **provided** by the **Zed Cam**.



Our approach involves instantiating a Machine Learning **model** capable of **providing bounding box** coordinates (x, y, w, h) and the **class name** for a given RGB image. Once we obtain the bounding box coordinates, we proceed to **crop the RGB image**. However, this cropping includes both the 2D points of the object and the background.

To **isolate only the object**, we employ a **masking technique** based on color. Since the object exhibits distinct colors compared to the background (which is predominantly gray), this approach effectively separates the two.

With the masked image containing only the object's points, we utilize the ROS function read_points to convert 2D points into their corresponding 3D points relative to the camera frame.

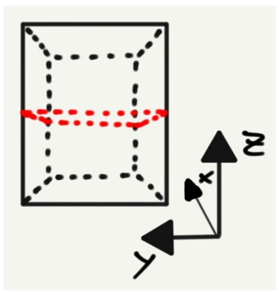


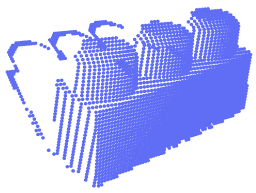
Subsequently, we **transform** the obtained **3D points into the world frame**. This involves acquiring the Zed camera's position and orientation, allowing for a straightforward transformation.

It's worth noting a crucial **observation**: while the AI model demonstrates high accuracy, **it faces challenges when dealing with objects positioned far away and in front of the camera**. Specifically, it **struggles** to **identify** the **short side** of the blocks. To address this, we implement an additional **color detection mechanism**. Blocks with a larger width are designated as yellow, while others are labeled blue. The color detection is based on the HSV scale. By converting an RGB image to HSV and analyzing hue, saturation, and value, we effectively discern the colors of the objects.

Three vertices:

The **idea** used to get the pose and the localization of a brick on the table **requires** knowing **three points**, taken in the following process.

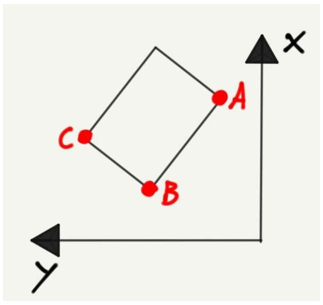


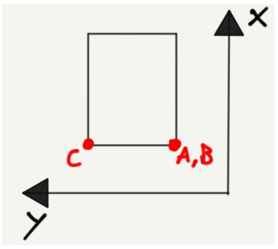


From the **point cloud**, containing only the points of the brick, we **select** all the **points at a specific height** (half the height of the brick) in order to work with less and more precise points.

Then we take the **three points**:

- **A**, **lowest y** coordinate;
- **B**, **lowest x** coordinate;
- **C**, **highest y** coordinate.

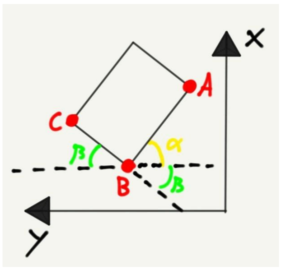




For **A** and **C** we actually do not take the exact lowest and highest values, we take the **ones** with the **lowest x** coordinate **around** those y coordinates. This is due to the fact that if the block is not rotated we want that B coincides with either A or C.

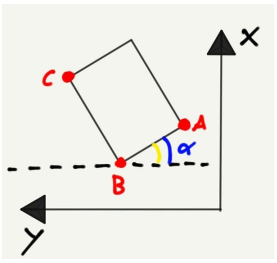
Pose detection:

Having the three vertices (A, B, C, read the related paragraph), we calculate the **distances AB** and **BC**, controlling that their values are over a certain threshold, otherwise we will not consider the wrong one in the following process.

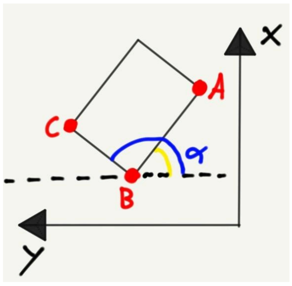


At this stage we get **two possible angles** (it will be only one if a distance is wrong): **alpha**, **beta**. Using $atan2$ with the coordinates of A, B, C.
Between the two angles **we choose the smaller one**, in absolute value, because it should have a **better accuracy**.

Alpha is now the right rotation for the brick, from the camera point of view.



The last step is to understand **how to grasp the shortest side** of the brick. It is necessary to know the values of the brick's sides, we get them from the training model that understands what type of brick it is.

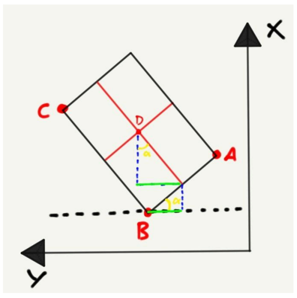


So, we **control in the direction of which side we rotate**, there are two possibilities:

- the rotation is in the direction of the **shortest side** and we have **nothing** else **to do**;
- the rotation is in the direction of the **longest side** and we have to **add** or **subtract** $\frac{\Pi}{2}$ at angle.

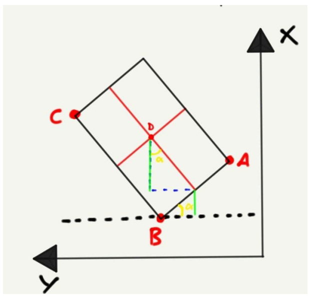
As we said, **the angle** we get is now right but it **is seen from the camera point of view**. The **end effector** is **perpendicular to our alpha** equal zero, not parallel. In order **to correct** this, it is needed to **add** $\frac{\Pi}{2}$.

Brick localization:



Having the three vertices (A, B, C, read the related paragraph) we need to know the pose and the sides' value of the corresponding brick, in order to **calculate the center point of the brick**.

At this stage, it is just a matter of **mathematics**, we want to **find D**:

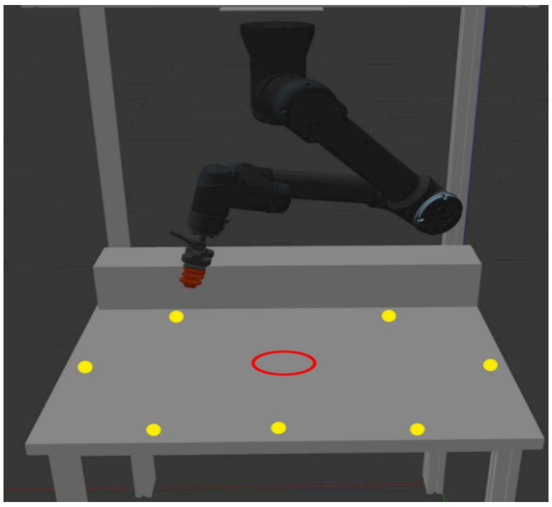


- **z** is actually a **standard value**
- $x = B_x + \frac{AB}{2} sin(\alpha) + \frac{BC}{2} cos(\alpha)$
- $y = B_y + \frac{BC}{2} sin(\alpha) - \frac{AB}{2} cos(\alpha)$

HIGH-LEVEL PLANNING

Avoiding singularities:

The **first main downfall** to tackle is the presence of **singularities** into the trajectory. To do that, we decide to **identify the location** where the **singular issue occurs**. As we can see, from the following image the **red circle** represents the **dangerous zone**, that is the place within the robot goes in singularity configuration.



To make sure our robot can go anywhere on the table, except the spots in the red circle, we created a **smart path plan**. We **use the yellow points** you see in the picture above. Starting from where the robot is, we figure out the best sequence of points to get to where we want. Here's the **idea**: we find the **nearest yellow points** for both the ***robot hand*** and the ***final place*** we want to reach. After that, we **plan a series of points** that let the robot move **in** a **circular** way to **stay away** from the **risky zone**.

Grasping operation:

To **pick up** and **move** the **brick**, we've planned a series of **specific steps**. First, we make sure the **robot** is **in** a **secure starting position**, no matter where it begins, so we implemented a **function** that moves the robot in a **safe configuration**. This helps with safety and sets the stage for the following planned movements. Then, we calculate the **path** the robot will follow **using** a method that considers **both movement and rotation**. This ensures the robot goes where we want it to go. When the **robot** is **above the brick**, we **move down the gripper** in order to use it to **grab the brick** by opening and closing it. Later, we **repeat** these **actions in reverse** to put the brick back down.
All points employed in this algorithm undergo validation through a function that ensures their placement within the designated table.

Moving more bricks:

Upon obtaining the **boundary boxes**, **yaw angles**, and **names** for **all the bricks** present on the table, we **assign a specific point** for each **type of brick**. Importantly, these predefined points are not assigned to any initial position, providing the flexibility to assign each point as needed. With these premises, we can perform a graspin operation for each brick on the table.