UR5 MOTION AND LEGO BLOCK'S RECOGNITION

developed by Bonetto Stefano, Roman Simone and Rigon Mattia

2022/23

1 Introduction

This project has been developped as the exam of "Fondamenti di Robotica" (UniTN ICE),and it is divided in 3 main parts:

- UR5 Motion
- Perception
- The high level organization

2 UR5 MOTION

2.1 Motion Plan

To move our UR5 we have decided to generate a trajectory looking the joint space. Our goal was to move the end effector of the robot from an initial position P_i to a final position P_F.

We can transform this problem into another one : starting from a q_0 configuration of our six joints , we want to achieve q_F final configuration (the q_0 configuration correspond to the P_i position of the end effector and the q_F configurations correspond to the P_F position of the end effector). For switch between the position and orientation space to the joints space we have used the inverse kinematics, which takes as inputs the coordinates x, y, z (in the robot frame), and returns a matrix made by 8 rows, in each row there is a possible configurations that correspond to Pf position of the end effector. In order to move from the position and orientation space to the joint space, we have used direct kinematics. The algorithm that we have used for choosing the right configurations between the 8 solutions produced by the inverse kinematics (which will be discussed later).

This is the basis of our motion plan, however we had to do some changes to make it work. Using this method used to move the UR5 does not allow us to know which trajectory it will have moving from one position to another one. To have more control on the trajectory we have decided to insert some intermediate points where the robot has to pass while moving from P_i to P_F. In our case the robot has just to reach some position on the table, in some specific order, and doesn't have to do a complex trajectory or reach some difficult points, that's why we have chosen to implement our motion plan like this. Passing through these intermediate points we have enough control of the trajectory. Another important factor for controlling in a better way the trajectory of our robot is a good algorithm used for choosing the better configuration between the 8 solutions produced by the inverse kinematics. Our function, called nearest, chooses the nearest configurations to the q_0: for each configuration it measures the difference vector between the configuration and q_0 (start configuration) , then it calculates the norm of this distance vector ; the result of this function is the configuration which has the smallest norm of the distance vector. In this way we are quite sure that our robot will not do strange moves to reach the goal point.

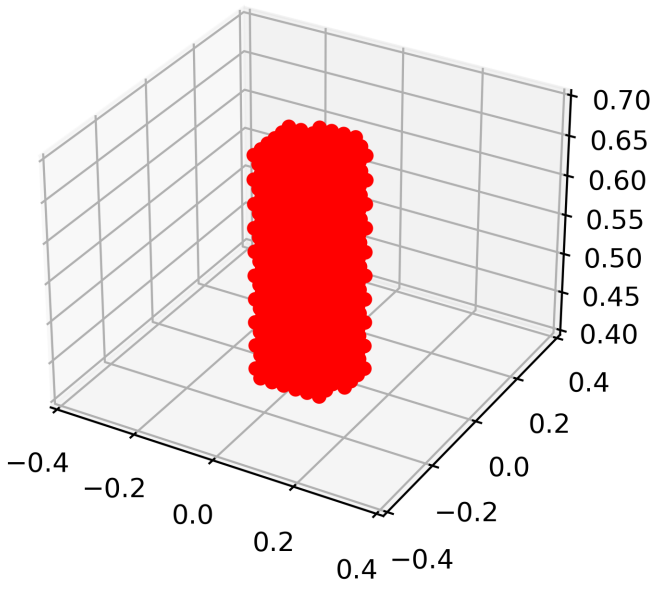
Our motion planning is quite easy, since our project is a pick and place project, and it has 3 main movements:

- Starting from the initial position (x_0, y_0, z_0) of the robot , it moves to the first intermediate point which has the same x_0, y_0 but a different z_0 which has been changed, and it's now at 20 cm from the Z of the floor.
- Now the robot reaches the second intermediate position, which has the x and y coordinates equal to the goal point (x_F, y_F), but it has maintained the same z of the previous position.
- At the end the robot is 20cm over the goal point, and what it has to do is just going down to reach the final position.

2.2 Checks

Here are some checks we have implemented:

- **check position**: while testing the behavior of our robot we realized that not all positions can be reached by it, especially those physically too far from the robot , but also near the shoulder singularity. In order to know if the position that we want to achieve is reachable or not we use a trick : firstly whether we check if the position and orientation desidered, are reachable by the robot , to do so check that at least one resultant of the inverse kinematics put inside the direct kinematics gives the same position that we have put inside the inverse kinematics. If so we are able to reach that point , in other words we are sure that the point that we want to reach is inside the reachable space. In this graph we want to highlight the points that we can't reach .



position that we can't reach

- **singularity check**: Given the 6 joint angles, it performs the direct kinematics for each joint, so that we can know the position of all of them. Then it checks whether this position is inside some defined range. In our case we have to avoid the collision with 3 plans :

- the ceiling of our environment
- the floor of our environment
- the wall behind the robot

Now we are able to predict whether the robot will hit something before making it start moving ; therefore we can deny the movement.

- **change turn side in case of collision with the wall behind** : while testing the behavior of our robot we realized that in some case the robot in order to pass from a position on the right side of the table, to a position on the left side (or vice versa) , it chose a trajectory that would have hit the wall behind. To solve this problem, and prevent the robot from not performing the task, since it noticed that it was colliding with something else, we forced manually the robot to choose another trajectory. This trajectory is exactly the same for all but one of the joints , the first one , the one in the shoulder that most influences the movement, by making it reach a final q identical to the initial one , but with an offset ti -2π , so as to impose that it turns in the opposite direction to the one initially decided upon.

3 PERCEPTION

3.1 Dataset creation

To make YOLOv5 recognize the lego blocks we had to build a training dataset containing images labelled with bounding box coordinates (rectangles) that surround our lego blocks. We used Blender, an open-source 3D creation software that allows us to make the dataset.

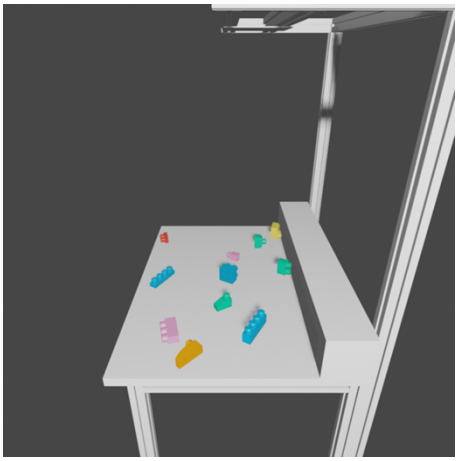
In the first place we made a dataset making the camera rotate around every single block that remained in the same position, because we were worried about over-training our model. With this method we didn't reach good results in the perception process. We decided to change method recreating a scenario as similar as possible to our simulation environment, so we added our own table to the scene, and let the lego blocks spawning on top of it. At this point the camera is static and, in every image, the only thing that changes is the set of blocks on the table and their random features:

- Position
- Rotation
- Color
- Type

For every scenario, we took a picture and then we labeled the elements with the following information:

- the class (the name of the lego Block in the following format: $Xn - Yn - Zn$)
- bounding box's vertices (the vertices of the smallest rectangle that contains the lego block)

In the following example, we have captured and labelled the scenario n° 7024



7024.jpeg (image)

```
X1-Y4-Z1 0.3522054111835039 0.603236252933911 0.05633575106597971 0.05072237272376812
X1-Y2-Z2-TWINFILLET 0.5693964770804159 0.5239250114921232 0.026956137874762698 0.031524924015034816
X2-Y2-Z2 0.4992051688565937 0.5955194476086211 0.04174544729581983 0.04763563592700959
X2-Y2-Z2-FILLET 0.6856984749422529 0.5005959281729415 0.02789471363011331 0.03740164743973686
X1-Y3-Z2-FILLET 0.41158855553876404 0.7738275110910464 0.061636056590055666 0.06669764833983085
X1-Y4-Z2 0.5561377334743494 0.7134717398138269 0.05497144312836344 0.07972244464354072
X1-Y3-Z2 0.36824975767772655 0.720942877816614 0.04455932230882298 0.06164535254411285
X1-Y2-Z2-CHAMFER 0.4862177330533436 0.6580284218835687 0.04114273386196404 0.04427724350789686
X1-Y2-Z1 0.3567722251673382S 0.5160808122775444 0.019753167382906667 0.025224172209979523
X1-Y2-Z2 0.6224819918593101 0.5802503366223972 0.03379235675233683 0.039766726759240908
X1-Y1-Z2 0.5097474085084963 0.5576625830298172 0.02940176018777174 0.023940182372155133
```

7024.txt (label)

(you can find our Python script to create the dataset here

3.2 Model Training

After that, we had obtained a dataset composed by 8000 images, so we wrote an easy Python script that split these images in three parts (folders):

- Train (70% of all images)
- Validate (20% of all images)
- Test (10% of all images)

At this point we had to start the training procedure with YOLOv5: our computation power wasn't enough to train a 8000 images dataset, so we decide to use Google Colab, a cloud-based platform that provides free access to Jupyter Notebook environments with preinstalled software packages and libraries for machine learning and data analysis that allows us to run Python scripts to train our dataset.

One important parameter in our training process is the number of epochs: an epoch is basically a complete cycle of training of the entire training dataset through a neural network, but it's important to highlight that having a high number of epochs doesn't mean achieving a more precise model: we have to pay attention not to overfit the model.

After a lot of tests, in which we changed the number of epochs and the number of images, we obtained the best results with 40 epochs for an 8000 images dataset. After that, we were able to acquire a .pt file that allows us to recognize the lego blocks correctly.

3.3 Image capturing and processing

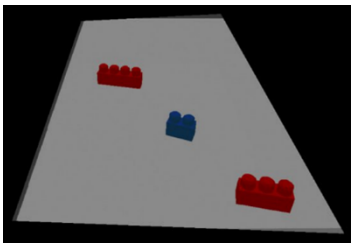
In order to capture the images, we used a ZED2: it's a Stereo Camera built for AI use that uses neural networks to reproduce human vision. We wrote all the code for the image processing and the object recognition in a Python file called *vision.py*.

The code has basically 4 main functions:

- receive_image
- recognition
- receive_pointcloud
- lego_processing

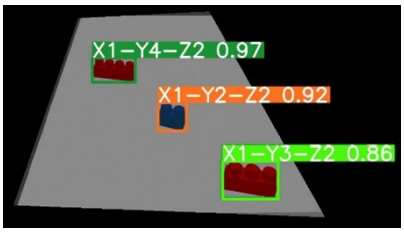
3.3.1 receive_image()

The first function is receive_image where we receive the image incapsuled in a message sent by the ZED in queue. First of all we save the image with opencv python library, cropping just our region of interest (the region of the table where the lego blocks are located). In the figure below you can see the cropped image.



3.3.2 recognition()

With this function we use YOLOv5 to recognize the lego blocks, the information obtained is the class name of the lego block and his bounding box. Since we have these coordinates, we are able to create a list which contains all the pixels into the bounding box ($[x, y]$ pair). In the figure below you can have a look to the processed image.

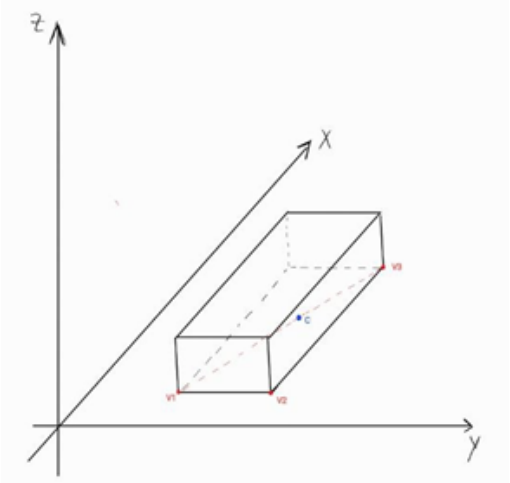


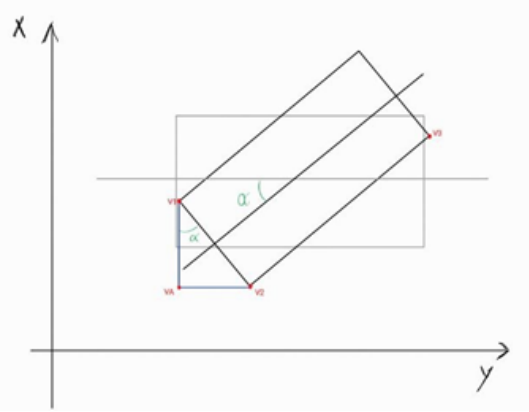
3.3.3 receive_pointcloud()

Now that we have the list created by the recognition function, we take the corresponding points in the pointcloud received from the ZED2. Once we have done it, we need to transform the frame system from the camera one, to the world one; after that, we save these points into another list.

3.3.4 lego_processing()

At this point we have to find the position and the orientation of each lego block. The algorithm we created to find the right position and orientation of the block is based on the three points showed in the figure: the rightmost vertice (V_3), the leftmost one (V_1) and the closest ((V_2) (everything about the ZED2). With these three points we're able to find the middle point of the block and the rotation on the z-axis with the Euler angles. Let's have a look at the figures:





- V_1 is defined as the vertice with the minimum y
- V_2 is defined as the vertice with the minimum x (but not the minimum y)
- V_3 is defined as the vertice with the maximum y

Based on that, C is defined as:

$$C = \left(\frac{X_{V1}+X_{V3}}{2}, \frac{Y_{V1}+Y_{V3}}{2}\right)$$

To find the right orientation of the lego block, we want to find the α angle:

$$\alpha = arctan\left(\frac{d(VA,V2)}{d(VA,V1)}\right) = arctan\left(\frac{|y1-y2|}{|x2-x1|}\right)$$

Furthermore, we can calculate the maximum height of the item to understand which side is leaning against the table. It's important to know that the height of the lego block varies depending on which sides is touching the table. Moreover, since we precisely know the dimensions of the blocks, we can adjust our initial classification of the scenario's items.

Finally, we can publish the message with the information gained.

4 HIGH LEVEL ORGANIZATION

We are able to communicate with the UR5 that implements a topic based communication: when a subscriber subscribe a topic, it will receive all the events published in that topic. A topic based infrastructure is composed by two main entities:

- **Publisher**: this is the entity that is able to public messages in the queue
- **Subscriber**: the entity that listen to all the events happening (queue publications)

Basically the publisher is the medium that connects the brain (our PC) to the UR5: the communication is bilateral, in fact the UR5 can publish his joint angles and he can also receive some messages from us. The perception part publish a message composed by the name of the current assignment (string) and a list which elements are the information gained by the lego_processing() function:

- The classification name ($Xn - Yn - Zn$ string format)
- His position: in particular it publishes a 3 values vector, the position of the center of the block (x, y, z)
- His orientation: an Euler Quaternion indicating the orientation relative to the axis

The *publisher_node* is permantly listening to the perception messages; when one of those arrives, the publisher, depending from the position and the orientation of the block, choose what to do. With the *move_to* function we take the actual UR5's joint angles (using *return_joint_states* function) and convert it to the position through direct kinematics, then with the *p2pMotionPlanIntermediatePoints* we obtain all the intermediate points to go from the actual UR5's position to the desired position. After checking the singularity collisions, we send the message (in different ways depending on if we are simulated or interfacing with the real robot).

For the simulation examples, we have created *spawnLego_pkg* which is the tool we use to spawn the lego blocks, with it we are able to spawn the lego blocks in the right spawning zone.

The lego blocks spawn always in random positions and we can also choose if we want the blocks to spawn with a random rotation or with the default one.

When the publisher receive the message from the vision , it picks one Lego at a time and places it in the chosen position according to the class of the Lego. The schema that we have decided to follow is this one :

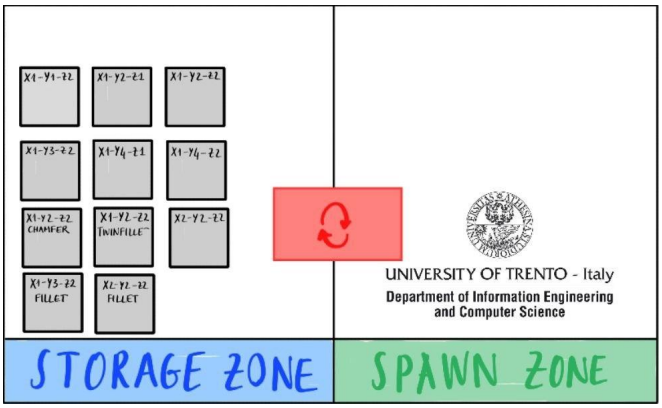


table schema

5 GAZEBO's KEY PERFORMANCE INDICATORS

KPI	VALUES
KPI1	ndeniefweio
KP2	vineoiwofoie