Project report
Introduction to Robotics
University of Trento

Matteo Beltrami & Damiano Bertolini & Massimo Girardelli

June 12, 2023

Abstract

This project uses a model of a UR5 robotic manipulator in order to pick and move lego blocks in certain positions. The blocks are first detected by using a 3D depth camera and classified by using different computer vision techniques such as YOLOv5 and OpenCV. The robot used both in simulation and in the real world is the 5th version of the 6 degree of freedom arm by Universal Robots. The camera used was the ZED 2 camera by Stereolabs. The purpose of this project is to develop a sw prototype able to recognize and move objects for a future implementation in the industrial sector.

1 Tasks

The objective of this project is to use the manipulator to pick the lego blocks standing on a table and to move them in specified locations according to the specific assignment. The blocks first need to be identified in the 3D space using the ZED stereo camera (x,y,z, rotation and orientation of the block); as a second step they need to be classified among 11 different types (also called classes); the robot can then pick them up and move them into the desired position. The tasks are 4 and here is a brief description for each of them:

1. Only one lego block is standing on the table with its base in natural contact with the ground. The brick has to be moved to a certain position, specific for its class;

2. Multiple lego blocks are standing on the table, one for each class. The bricks have to be moved to a certain position, specific for their class;

3. Multiple lego blocks are standing on the table and can be lying on one of their lateral sides or on their top. Objects of the same class have to be stacked up to form a tower;

4. The objects are positioned randomly on the stand and have to be moved to create a composite object with a known design.

2 Basic Software

As a backbone for connecting and allowing communication between the different software components the ROS middleware has been used, with topics and services sharing messages to the parties. The application software has been divided into two main areas, the former implementing the vision part making use of the output of the camera and the latter taking care of the kinematics, i.e. the motion of the robot. The vision part was mainly written in Python as it makes use of some high-end computer vision and object recognition libraries well implemented in such programming language. The kinematics part is instead mainly written in C++ and exploits among others a library for matrix manipulation.

3 Computer vision

This section reports the process of detecting and classifying the blocks; the generic tools and algorithms used in our project are Blender, YOLOv5, OpenCV and PCA techniques.

3.1 TOOLS

- **Blender** is a free and open source software that supports 3D modeling and rendering useful for synthetic datasets creation.
- **YOLOv5** is a model in the You Only Look Once (YOLO) family of computer vision models, commonly used for object detection.
- **OpenCV** is a library of programming functions mainly for real-time computer vision which contains a great variety of algorithms that can also be used for 3D pose estimation
- **Principal Component Analysis** is a Machine Learning technique for analyzing large datasets containing a high number of dimensions, decreasing the number of features while preserving the maximum amount of information

3.2 BLOCK'S DETECTION WORKFLOW

1. In order to train the YOLOv5 neural network it is necessary to have a dataset that well represents both the simulated and real environments. To generate the images we used Blender to recreate the environment and through its scripting tool it was possible to automate the generation of images with random arrangements of the blocks position, pose, color, brightness conditions and shadows for augmenting the variancy. Also labels of blocks, class and position were added automatically through blender rendering tools. The final dataset was composed of a total of 10k labeled images with a resolution of 640x360 (no higher resolution was actually required since YOLO already performs very well with 640-pixel images, a better quality is instead required

later for OpenCV algorithms), in 200 of which the table was completely empty in order to prevent overfitting and cover also the limit cases.

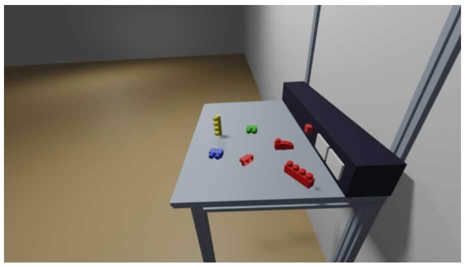


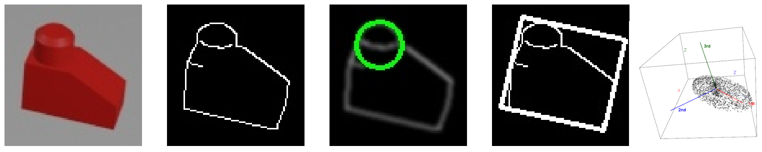
Figure 1: Dataset image generated with Blender.

2. The dataset was trained for 100 epochs through GPUs available on Google Colab; some images were left for validation (run every epoch) and others for the final test. The images were divided into sets as follows: 70% training set, 20% validation set, 10% test set and the training phase didn't involve data augmentation techniques. The precision reached by the model on the test images was 99,1% and no effects possibly caused by overfitting were detected. Weights were then downloaded for being loaded by the Python script. Since we noticed that this model was working pretty well on the images captured by the ZED camera in the real world, we decided to use it also for that environment.

3. The vision ROS node gets the Point Cloud and RGB image from the ZED camera and runs the latter through the Neural Network, giving as output the blocks detected and their bounding boxes.

4. For each block within the table boundaries (eventual false positives are discarded), the following process is executed:
   (a) The bounding box is cropped from the original RGB image
   (b) Canny algorithm is used for detecting the block's edges
   (c) The smallest rectangle circumscribing the largest contour (i.e. the block's shape) is found, therefore identifying the center and rotation of the block
   (d) A first guess of the inclination of the block is given: this is done first by detecting circles in the bounding box, then by comparing their average position with the block's center and by analyzing the derivative of the least squares polynomial fit best representing the circles' centers
   (e) Finally, Principal Component Analysis is applied to the Point Cloud cropped based on the 2D bounding box; this algorithm (in an abstract way) draws the lines (or planes) that approximate the data as well as possible in the least squares sense making the variance of the coordinates on the line or plane as large as possible. These N-Dimensional spaces are called Principal Components and PCA is able to tell us how much the Cartesian

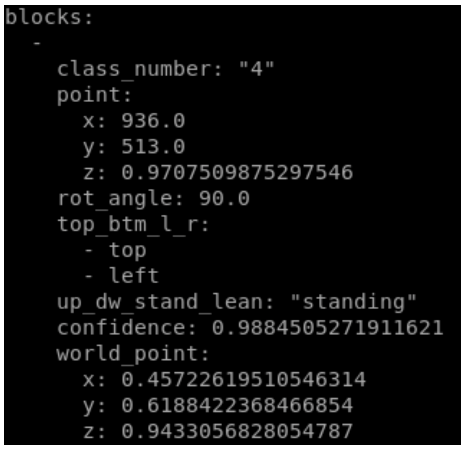
x,y and z directions are influencing these components. As an example, imagine a parallelepiped going from the left to the right of your eyes, its Principal Component will be in that direction since there is the majority of the variance of the points composing the block. Based on the first two eigenvectors we can give a good estimation of the block's 3D orientation



(a) (b) (c) (d) (e)

Figure 2: Detection and pose estimation.

5. The results are then published in a topic the kinematic part is subscribed to and contain the full 3D information about each detected block.



```
blocks:
  -
    class_number: "4"
    point:
      x: 936.0
      y: 513.0
      z: 0.9707509875297546
    rot_angle: 90.0
    top_btm_l_r:
      - top
      - left
    up_dw_stand_lean: "standing"
    confidence: 0.9884505271911621
    world_point:
      x: 0.45722619510546314
      y: 0.6188422368466854
      z: 0.9433056828054787
```

Figure 3: Vision message published.

4 Kinematics

The main components of the kinematics are:

- forward kinematics;
- inverse kinematics;
- evaluate configuration;
- p2p

FORWARD KINEMATICS

Takes as inputs joint position and computes the transformation matrix for every joint and arm and multiplies then together to get a final matrix T0e, the transformation matrix from the base to the end effector, this is also saved inside the class since it's used multiple times.

INVERSE KINEMATICS

Takes a position of the end effector as input and computes the inverse kinematics. The solution is a 8x6 matrix, one line for every possible configuration for the ur5 arm.

EVALUATE CONFIGURATION

Given a joint state, computes the IK and compares all 8 configurations with the current one, the value closer to 0 is the index of the current configuration. This index is saved and used for the rest of the procedure. This avoids problems when the arm was rotated from previous uses, especially the shoulder joint and prevents the robot from spinning and hitting the table at the start of the procedure.

P2P

The p2p is the solution we used to calculate the movement of the arm. The robot is controlled in the joint space so the solution fits the technology. It is a cubical interpolation from the starting joint configuration to the final. The function has 4 internal constraints:

1. $q(t_{start}) = q_{start}$, the first position is the start position;
2. $q(t_{finish}) = q_{finish}$, the last position is the final position;
3. $q'(t_{start}) = 0$, the speed at the start is 0;
4. $q'(t_{finish}) = 0$, the speed at the end is 0.

This guarantees a smooth transition from start to finish without sudden stops or unwanted accelerations. It takes an initial and final position in 3D and the number of iterations, here is the breakdown of the function:

1. calculates the IK of the starting and final point and get 2 joint states;
2. subtracts the 2 joint states to get and offset for each joint, we will work in the range $(0, offset)$ from now on;
3. wraps the offset in $[-180, 180)$ degrees so the joint takes the shortest path in the circle;
4. build a Matrix for the polynomial solution without coefficients;

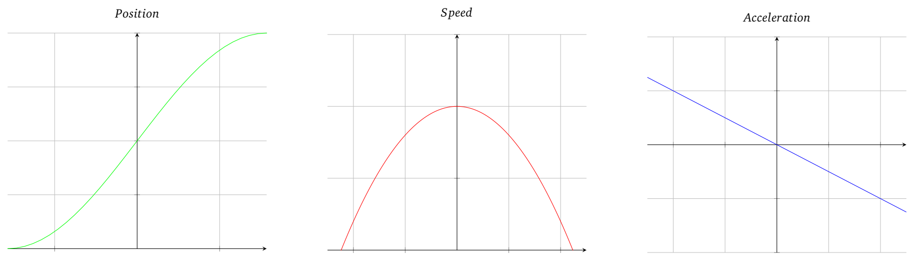


Figure 4: Position, Speed and Acceleration over time.

5. for every joint, if $M * A = B$, where A are the coefficients and B are the constraints, then $A = M^{-1} * B$;
6. for every joint and every step, with A as coefficients, find the value of the polynomial at a given time;
7. add back the starting value to the offset to get the correct value.

The result is the the form of a list of 6D vectors, one for each step.
The effetcs are visible in figure 4.

Other implementations could use 5th order polynomial to make acceleration 0 at the start and end or trapezoidal time scaling to maintain a fixed max speed.
We decided to wrap the angles to find the shortest rotation, for example a movement from 0 to 270 is shorter if we move from 0 to -90° and because the ur5 already uses angles in the range $(-360, 360)$, so if a theoretical final angle is 100° and the starting is -270 a risk of spinning the joint around itself multiple times, while in practice we can look at -270 as 90, so just a 10 rotation.
The use of a p2p plan makes the robot move in a more circular path, most of the movements are done with the first joint, this also helps to avoid singularities under the base and is really fast ($0.017s$, on a home computer for 10000 iterations), because the IK is done only twice for the whole process, other differential kinematics implementation what work in the work space may need to compute it at least once for every iteration. This p2p implementation also has weaknesses, for example it can't follow a path, to solve this problem, a linear polynomial with a parabolic blend can be used.

- conversion for rotation matrix, 3D points, ZYX Euler angles;
- real value of acos and asin for values beyond $(-360, 360)$;
- Jacobian (not used);
- differential kinematic control (not used);
- pr vector, a 6D vector with XYZ and ZYX Euler angles stacked, useful when defining a position in space without leaving the orientation undefined.

ROBOT

The Robot class is the controller of the ur5, it uses ROS to listen on $/ur5/joint\_states$ and publishes on $/ur5/joint\_group\_pos\_controller/command$ and $/gripper$ rostopic channels. A callback copies the values of the joints constantly published by the robot inside a vector to be used more easily, some of the joints are not in order when published so they are also sorted, this callback also knows if it's in a simulation where there are more than 6 joints published. This class published the commands for the robot in the Float64MultiArray format for every robot joint and eventually gripper joint. With the method move_to() the end effector moves to the desired position in the number of steps that are passed:

1. Calculates the current ee position with forward kinematics and the current joint state published from the robot;
2. Uses p2p to get a list of joint positions to follow;
3. (optional) save the path for debugging;
4. publish 1 message every 1ms, the waiting is done with ros and not with a normal sleep because it syncs the execution of every loop and not just wait without taking in consideration the execution time of the commands in the loop.

MAIN

The main starts the robot callbacks and the vision callback via ros on the rostopic "vision_results" and waits for everything to be ready. Once a non-empty vision message arrives, it starts the procedure:

1. evaluates the IK index for the robot configuration;
2. asks for mode to operate: testing (performs some default movements before going in automatic), manual (moves to a location following a txt file with the end point and rotation and moves the gripper before going in automatic), or automatic;
3. move to a default position;
4. gets the position of the next block from the vision messages, type and rotation;
5. corrects the final position based on the block and rotation;
6. go above the block;
7. lower the arm;
8. close the gripper;
9. move the arm up;
10. go to the default position;
11. go to the final position based on the class;

12. lower the arm;
13. open the gripper;
14. if there are more blocks in the list go back to 4;
15. wait for vision messages and go back to point 1.

The correction of the final position is needed for blocks that shorter and for blocks that are standing or on the side.

HELPER

The Helper class contains all the functionalities that were needed but not related to one single class, contains methods to constrain angles, conversion of coordinates from world to robot frame and legacy code.

EXTRA

Some extra tool were developed to better work on the project:

- Block spawner: C program that automatically creates a working world for Gazebo with blocks in different types, positions and numbers;
- *my_vision_messages*: ros messages to share vision results between the controller and the vision
- tests to inject static vision messages and joint positions to help develop the code in the earlier stages, written in python and C++.

5 Performance indicators

Task1	KPI 1-1	14s
	KPI 1-2	33s
Task2	KPI 2-1	184s (3 objects)

The breakdown on the timing is:

1. Vision detection;
2. path calculation;
3. movement;
4. sleep.

Note: all timescales were taken on a home pc with very limited resources and from a fresh start of the applications. The lab environment is much faster. For object recognition, since machine learning is used the first start is slow (14s on home pc and 3.5s on lab) and every run after the first is very fast (lab, total 0,5 seconds for single block and 0.6 for 5 blocks).

The path calculation is almost instantaneous, 0.017s for a 10000 step. The movement of the arm is splitted in 7 segments per block +1, each segment is divided in a number of 1ms steps, 2000 for task 1 and 4000 for task 2.
The arm can go very fast but we decided to use it in a slower configuration to avoid damaging the equipment and have time to react in case of risky situations. The sleep is both needed to have a short fraction of time to evaluate the situation and eventually stop the robot and to have time to publish the gripper via LUa script, this execution is asynchronous but it interferes with the robot commands and needs to be completed before other commands are sent, this is why there are long sleeps when the gripper is being closed or opened.
The KPI timing could be improved by launching the vision before and keep it running while the robot is starting, using the lab pc which is much faster, reducing the steps to make the robot faster, eliminate sleeps and use the commands from the control tablet to operate the gripper. This could lead to theoretical KPI of:

Task1	KPI 1-1	0.5s
	KPI 1-2	15s
Task2	KPI 2-1	44s (3 objects)