Dipartimento di Ingegneria e Scienza dell'Informazione
Accademic Year 2022/2023

Report for the course

Fundamentals of Robotics

Github repository: link

Authors:
Matteo Mascherin
Stefano Sacchet
Amir Gheser

Professors:
Luigi Palopoli
Niculae Sebe
Michele Focchi

Contents

1 Task

Overall, the task of the project is to develop the code for a robotic arm capable of moving Megablocks of different classes to precise locations despite their orientation. Each of the 10 block types must be recognized and classified properly even if it's placed upside down or lying on its side. The final task requires blocks to be moved around and piled up to build a hard-coded building such as a castle, a tower or more. The overall objective is split into 4 assignments.

- Assignment 1 There is only one block of any kind of class, naturally in contact with the ground, which has to be moved to the appropriate location.
- Assignment 2 There are several objects naturally in contact with the ground which have to be moved to the appropriate locations.
- Assignment 3 There are several objects with no orientation constraints, in contact with the ground only - they can't lean on other blocks - which have to be moved.
- Assignment 4 Given a specific construction with well known design and the necessary blocks, the robot is to complete the construction.

2 | Perception

2.1 Introdution

The ability to perceive is a key component of our robotic arm as the blocks need to be classified and labelled in 11 different ways before moving them to the appropriate location. For classification means we opted for a machine-learning-based solution, specifically YOLO, a pretrained fast R-CNN model which is then fine-tuned on a synthetic dataset.

2.2 Objective

Our model must recognise Megablocks from an image fetched from a ZED Camera and retrieve the coordinates of the former and convert them with respect to the world frame. These will then be converted into coordinates with respect to the base frame so that the robotic arm can accurately approach the block.

2.3 Getting data

In order to retrieve data we used a **ZED** Depth-camera placed on the side of the workbench. The ZED camera sends shots which are then cropped and fed to **YOLO**. From the bounding-boxes around the recognised objects drawn by the model we retrieve the central pixel coordinates. With these we can get the complete coordinates of the block using the depth sensor of the camera which are then transformed into world frame coordinates.

2.4 Method

To face the challenge we relied on a fast **R-CNN** (Region-Based Convolutional Neural Network) with pretrained weights known as **YOLO**.

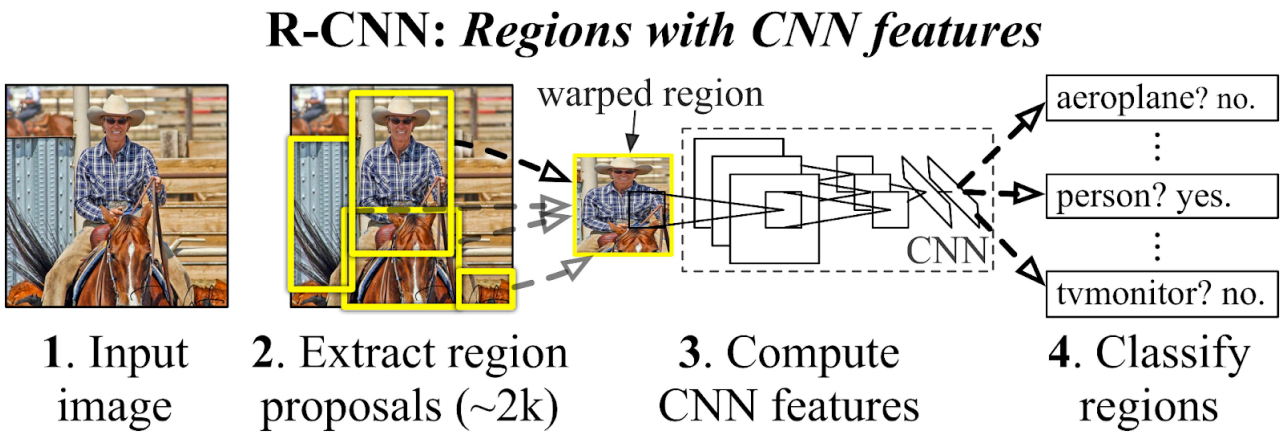


Figure 2.1: RCNN schema

This model uses a region proposal network (RPN) to generate suggestions of regions to be fed into the model for classification. Compared to an algorithm like selective search, this approach saves a lot of time. YOLO has been pre-trained on a specific dataset which allows it to recognize roughly nine thousand classes. The task requires a fewer number of classes (11) which are not amongst the previously mentioned classes. To achieve this we fine-tuned the weights by training for a one hundred epochs on a synthetic dataset. We had ten ".stl" files

at our disposal each representing one block. By running a Blender script, we imported a random number of blocks into the Blender environment and had a camera taking pictures of it, generating exactly 6000 pictures. The dataset was later expanded with pictures from simulation and from the real world. All pictures of the last two kinds were used to perform dataset augmentation via rotation. To sum up, the dataset holds roughly 6500 labelled images then split into training (60%), valid (20%) and test (20%) sets.

A few last changes were made to improve the robustness of the model. In fact, at each rotation step the colour of the block and the background were randomly changed. Moreover, we decided to randomly obscure some pixels to introduce further noise into the data and better simulate occlusion.

2.5 Algorithm

Algorithm 1 Vision node: callback code flow

```
if detectionRequest then
    image ← img.MSG_TO_CV2()                          ▷ Convert image from msg to cv2 format
    croppedImage ← image.CROP()                       ▷ Crop the image to see only the table
    blockList ← DETECT(croppedImage)
    for i = 0 to LEN(blockList) do                    ▷ Get coordinates from (x,y) pixel
        block ← RECIEVEPOINTCLOUD(pointCloud, blockList[i])
        if block ≠ None then
            blockListCoord.APPEND(block)
        end if
    end for
    block ← MIN(blockListCoord, key = "x")            ▷ Keep only nearest block to camera
    block ← BUILDMSG(block)                           ▷ Build custom massage
    TALKER(msg)                                       ▷ Publish the message
end if
```

3 | Motion

3.1 Algorithm

After the position of the block has been correctly communicated, the robotic arm needs to move it appropriately, without excessive waste of time whilst avoiding singularities.

The general plan is to have the robotic arm follow a straight line connecting the position of the end-effector and the target position. Firstly, we assess the distance from our final destination then we divide it by our desired end-effector velocity to determine at which t our goal is reached. The number of steps is tied to the desired velocity, while the dt between one step and another is linked to the ROS **publication rate**.

3.2 Forward kinematics

In order to move the robotic arm properly we need to assert the pose of the end-effector at a certain time instance. To do so, we follow the direct Kinematics approach considering an open-chain manipulator with **6 DOF**. We reasonably consider the kinematic relationship between consecutive links recursively. Each link has a reference frame representing the transformation from the previous frame to the current one. Therefore the transformation from link 0 to link 6 can be determined as follows:

$$T_n^0(q) = A_1^0(q_1)A_2^1(q_2)\dots A_6^5(q_6) \tag{3.1}$$

The forward kinematics is a systematic recursive computation of homogeneous transformation matrices where each $A_i^{i-1}(q_i)$ is a function of the single joint variable. This systematic approach is known as the Denavit-Hartenberg convention.

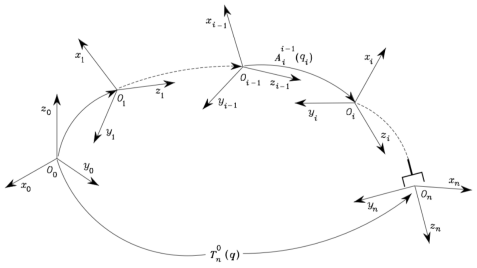
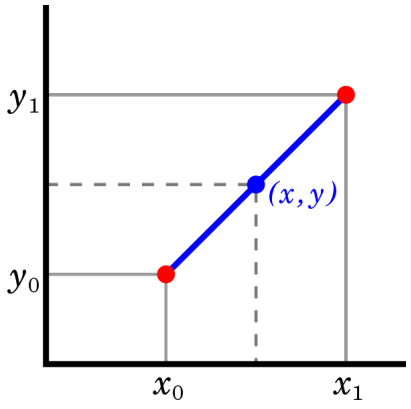


Figure 3.1: Forward kinematics

3.3 Linear interpolation

As previously explained the robotic arm is to follow a straight line which is divided into intermediate steps. We use linear interpolation, which is a means used to approximate curves into a line, to retrieve the next position to be reached at the time instance t.



To work this out we exploit the equality of the slope between $\frac{y_1-y_0}{x_1-x_0}$ and $\frac{y_1-y}{x_1-x}$ From which we get

$$y = y_0 \left(1 - \frac{x - x_0}{x_1 - x_0}\right) + y_1 \left(1 - \frac{x_1 - x}{x_1 - x_0}\right) \tag{3.2}$$

Finally, after some algebraic work we obtain the final formula found in the code:

$$y = y_1 \cdot x + y_0 \cdot (1 - x) \tag{3.3}$$

```
/**
* @brief The position to be reach at an instance t
*
* @param t time elapsed so far
* @param xef final position
* @param xe0 initial position
* @param movementTime time to finish the movement
* @return Vector3f reprensenting the position
*/
Eigen::Vector3f xe(float t, Eigen::Vector3f xef, Eigen::Vector3f xe0,
const float & movementTime){

    t = t / time
    Eigen::Vector3f x;
    x = t * xef + (1-t) * xe0;
    return x;
}
```

Where referring to 3.3 y is x, y_1 is x_{ef}, y_0 is x_{e0} and x is t.

3.4 Algorithm (reprise)

At each step we work out the current pose of the end-effector following the forward kinematics algorithm. After that, we determine the next position we need to reach and the velocity required to succeed in the aforementioned amount of steps.
Once we have obtained these parameters we use them to calculate the joint differential velocities $\dot{\boldsymbol{q}}$. As this value represents the velocity we simply multiply it for the amount of time which is passing and obtain the change required in the joints.

Algorithm 2 Differential kinematics algorithm

$x_0, Re_0 \leftarrow$ FORWARDKINEMATICS(currentjoint) ▷ Compute the current end-effector position and orientation
$distance \leftarrow (currentPos - Targetpos).\text{NORM}()$ ▷ Compute the distance
$movementTime \leftarrow \frac{distance}{velocity}$ ▷ Compute movement duration in seconds
$dt \leftarrow \frac{1}{pubblicationRate}$
for $t = 0.001$ to $movementTime$ **do** ▷ Compute a movement step every 0.001 s
  $x, Re \leftarrow$ FORWARDKINEMATICS(qk) ▷ Compute ee pos and ori at time t
  $vd \leftarrow (\text{XE}(\text{t}, \text{targetPos}, \text{x}_0) - \text{XE}(\text{t} - \text{dt}, \text{targetPos}, \text{x}_0))/t$
  $xArg \leftarrow \text{XE}(\text{t}, \text{targetPos}, \text{x}_0)$
  $\dot{q} \leftarrow$ INVDIFFKIN(currentJoint, x, xArg, vd, Re, targetOri, kp, kphi)
  $qk1 \leftarrow qk + \dot{q} \cdot dt$ ▷ New joint configuration to be published
  $qk \leftarrow qk1$
  PUBLISHJOINT(qk1)
end for

3.5 Trajectory planning

A few adjustments have been exercised to avoid singularities. Firstly, we have changed the homing procedure in order to ease grasping objects from the right and fixed it so that the links would not obstruct the camera

too much. In order to avoid moving objects on the plane while trying to grasp them, the robotic arm always approaches the blocks from the top and subsequently drops down. To clarify, if an object is lying at position $\boldsymbol{x} = \mathbf{(0.1, 0.6, 0.92)}$ the robotic arm will reach position $\boldsymbol{x'} = \mathbf{(0.1, 0.6, 1.1)}$ [1] and only then drop down to the block height to grasp the object.

Since the robotic arm follows straight trajectories we have put a few other adjustments into place in order to **avoid singularities**. For example we set two "checkpoints" one on each side of the table in order to firstly follow a wider trajectory when going from one side to another and then to avoid shoulder singularities. In fact, in some cases without these precautions, the trajectory forced the end-effector to pass just below the whole robot infrastructure, soaring the chances of shoulder and elbow singularities.

[1] Example referring to world frame coordinates

4 | Planning

4.1 Introduction

A master node, known as planner, is responsible for handling and managing the whole system by coordinating messages between the vision node and the motion node.

4.2 System

The systemic managing the robotic arm is represented as follows 4.1.

- Step 1 to step 4 describe the main process of obtaining the coordinate of a block on the workbench, until all the blocks has been moved
- Steps 5 to 8 represent the behaviour required to move the robotic arm to a certain position and to grasp the blocks.

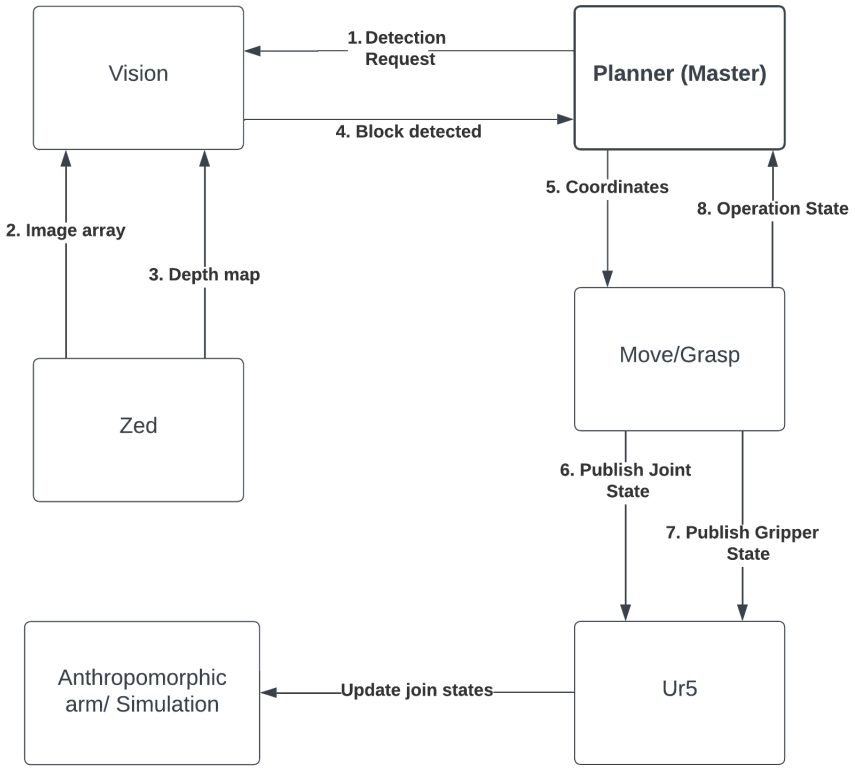


Figure 4.1: Project schema

Communication between nodes occurs via ROS (Robot Operating System), which will be extensively described in the next section. Each operation happens through a specific message type.

4.3 Message types

Our system works with four custom message types and makes use of ros **std_msgs** and **geometry_msgs**. A self-explanatory pseudo-code of the messages follows.

4.3.1 BlockInfo.msg

```
//block class
std_msgs/Byte blockClass
//block position detected by the camera
geometry_msgs/Point blockPosition
```

4.3.2 Coordinates.msg

```
//initial position of the block
geometry_msgs/Point from
//target position of the block
geometry_msgs/Point to
```

- steps 2,3 use the ZED APIs to communicate
- step 4 uses the custom **BlockInfo.msg** type
- step 5 uses the custom **Coordinates.msg** type
- steps 1, 8 happen by sending a simple byte

5 | Dependencies and third party services

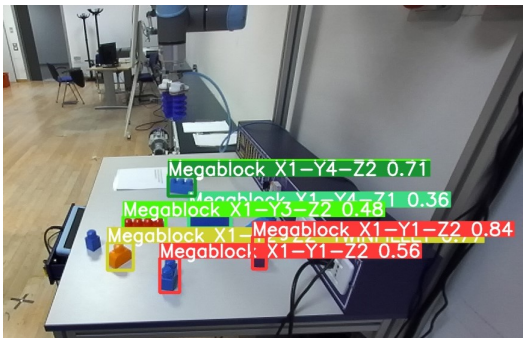
5.1 ROS

ROS (Robot Operating System) is an open-source framework for building robot software. It has been very useful as it provides useful features and effectively coordinates the system through its publishing/subscribing policy. We used ROS to coordinate the various nodes in order to separate the different tasks into different codes. Such nodes communicate through topics which they either subscribe or publish to.

This allowed us to make the code more modular and to split the various tasks (vision, move and planner) into different code parts

5.2 YOLO

YOLO (you only live once) is one of the most popular models and model architectures for object detection. We use it to detect objects on the table, classify them and obtain their coordinates from the bounding boxes. The training was carried out through the GPUs of Google Colab for about a hundred epochs.

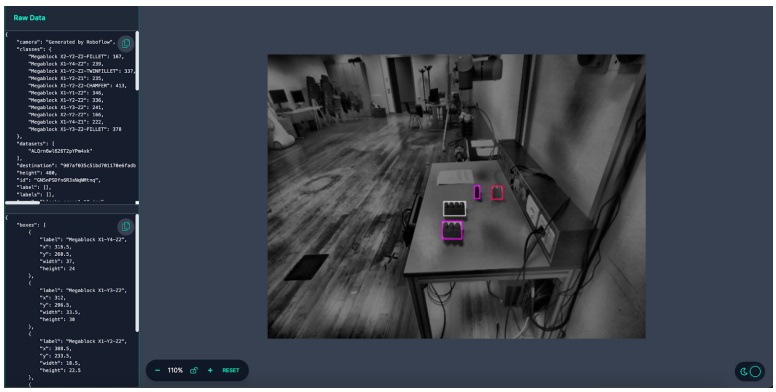


5.3 Google Colab

Google Colab is a notebook service from google which lets people run python and bash code on some CPUs and GPUs made available to the user. It has been primarily used to train the yolo on their GPUs in order to achieve the aforementioned fine tuning.

5.4 Roboflow

Roboflow is a third party service which was used for labelling purposes. Our dataset contained several synthetic images which were automatically labelled, but also some real life images and screenshots from the simulated environment which had to be annotated manually and augmented with Roboflow's service. In the image on the right you can see Roboflow's interface, a noisy image is shown with the manually drawn bounding boxes on the right side. On the other hand, the labels are on the left side.



6 | Key Performance Indicators

All data was taken based on simulation video uploaded on the github repository. Click here to watch. The end-effector velocity was limited to 0.3 m/s on the free movement and 0.1 m/s on grasping movement, for safety reasons.

6.1 KPI 1-1

This Key Performance Indicator measures how quickly each block is recognized by our model.

1. First block: **3 sec**
2. Second block: **3 sec**
3. Third block: **3 sec**

6.2 KPI 1-2

This Key Performance Indicator measures how long it takes for the block to reach it's final position after being recognized.

1. First block: **25 sec**
2. Second block: **24 sec**
3. Third block: **24 sec**

6.3 KPI 2-1

This Key Performance Indicator is simply a sum of the KPIs 1-2 for each block on the workbench.

1. First block: **28 sec**
2. Second block: **27 sec**
3. Third block: **27 sec**

7 Reflections

7.1 Planner-orientated structure

We built our project around the planner node which acts as a master that decides **when, how and where** a task is to be completed. Hence, the vision and move node act as slaves and simply listen and execute the planners instructions separately. This is crucial as it strips the communication to the essential.

7.1.1 ACK

In order to synchronize all nodes and to ensure a proper execution, the system is programmed to send an acknowledgement whenever it finishes a predefined task. Consequently, the planner is aware of what is happening and can continue instructing the other nodes.

7.2 Trajectory planning

We also noticed that certain trajectories were hazardous and by navigating them without any "checkpoints" one could come across some singularities. Hence, we divided our motions in a movement on the z axis and then on the x, y frame which significantly decreased the likelihood of singularities. We made some last tweaks when selecting the target area to ensure that movement of the robot is as safe as possible.

7.3 Subscriber synchronization

One of the main challenges we faced in the vision node was the synchronisation of the image and point cloud callbacks, because they are two different functions used to retrieve data from the zed camera. We needed to first elaborate the rectified image with YOLO to detect the blocks and then use it to calculate the coordinates thanks to the point cloud. Therefore, we ended up using the **TimeSynchronizer**, referenced from the ROS wiki which allows us to merge the two callbacks into a single one. By doing so we can process the data in the desired order.