Department of Information Engineering and Computer Science

Bachelor's Degree in
Information, Communications and Electronics Engineering

Robotics Project

Lecturer

Prof. Luigi Palopoli

Prof. Niculae Sebe

Prof. Michele Focchi

Prof. Placido Falqueto

Students

Mattia Meneghin [210561]

Filippo Conti [218297]

Nicola Gianuzzi [209309]

Academic Year 2022/2023

Contents

Abstract

This is a summary of the detailed report, that is available into the *documents* folder. Please refer to that file.

This project is based on ROS, namely Robot Operating System. More in detail we utilized the noetic ROS version and Locosim, a didactic framework to learn/test basic controllers schemes on quadruped robots. The project is composed by two main modules: **Vision** and **Motion**.

As for the vision we used YOLOv5 and PyTorch framework for the real time object detection through the ZED-Camera. The simulation was done in Gazebo and Rviz, exploiting ROS functionalities. All of these operations are possible thanks to Catkin, a build system of ROS, that combines CMake macros and Python scripts to provide some functionality on top of CMake's normal workflow and it was necessary in order to create the ROS nodes.

1 Introduction

1.1 GitHub Repository

Click this link Robotics Repository

1.2 Setup the environment

- ROS
- python3
- Git
- pip3
- Catkin
- Locosim
- Gazebo
- Yolov5
- Eigen3
- Ipython
- PyTorch

Please refer to the detailed report to get a step-by-step guidelines.

1.3 Tasks assignments for each group components

- Conti Filippo: Planner, movement, kinetics and spawner coding with the ACK and authorization system.
- Gianuzzi Nicola: Start script, Environment set, spawner coding and documentations
- Mattia Meneghin: Vision training and recognition, videos and documentation.

2 Motion

The package motion is a Catkin package that contains 3 folders:
src It contains 2 executable files called *planner.cpp* and *movement.cpp*
msg It contains 2 message files called *legoFound.msg* and *legoTask.msg*
include It contains the *kinetics.h* library

2.1 Planner

Planner is the executable in charge of communicate with the *vision* package. The planner subscriber receives form this package through the vision publisher the lego class identified, its position and orientation. At the receiving side will arrive a message structured following the instructions inside the file stored in the motion package (*motion/msg/legoFound.msg*), it will have all the default destination coordinates for each class of lego specified in the *planner.cpp*.

2.1.1 Planner commands from vision

This commands are received from vision and informs how the planner module has to do. Then the planner computes and send specific commands to the movement module. That commends are explained in the movement section.

0. **No Command**: planner will ignore any packet sent by the vision
1. **Detect**: planner receive all the info about the detection of the vision
2. **Quit**: Once the vision has finished its tasks, the planner can be closed

By the ID class provided, it knows how to put the corresponding lego in its destination on the table. Once the planner has all the necessary info, exposes the legoTask.msg using the publisher to the Movement module.

2.2 Movement

It receives the *motion/msg/legoTask.msg* from the Planner and gets the info about:

- command to execute without caring about the class, most frequent is the 0: *catch_obj*
- initial coordinates
- diameter gripping and the final object position [optional]

Inside the *movement.cpp* there are basic instructions to move the UR5 robot referring to a library called *Kinetics*, (stored in /motion/include.kinetics.h).
Once the movement has terminated the task, send an ack to the planner, that checks that everything is done correctly and completely, then the planner communicates to the vision that requested the task the end of the jobs. At this point if the vision doesn't require further tasks, the workflow is complete, otherwise it will go on with the remaining requests.

2.2.1 Movement commands from the planner module

These commands are received from the planner module and sent as execution to the robot

- **No Command**
- **Test**: check if the movement received correctly the packets
- **Wait**: waits for seconds set in the specific field in the LegoTask.msg
- **Move**: sends a packets contains a trajectory (start-point & end-point)
- **Grasp**: assigns a parameter to close the grasper
- **Ungrasp**: assigns a parameter to open the grasper
- **Default Position**: makes the arm move to a specific preset default position
- **Fast-catch**: moves the arm from the actual position directly to the object, catches it and relocates in the specific position ((composition of several commands)
- **Catch**: moves the arm passing from a central point, then over the object, goes down towards it, catches it, goes up passing from a central point, moves to the final destination, goes down and left the object, returns to the default position. (composition of several commands)

2.3 Kinetics

Kinetics is the library that contains all the primitive instructions in order to calculate angles, spatial displacement, matrix and all the mathematical functions related.
Stored in */motion/include/kinetics.h*,it includes the necessary classes and functions for dense matrix operations, it provides a wide range of functionalities, including matrix algebraic operations.

2.3.1 Euler's singularities

It is possible to have some issues when the legos stand in the central part of the table exactly under the robot, because of the Euler's singularities. However we implemented a basic solution, it is the catch function, that avoids these critical areas rather than the **fast-catch**. The catch command passes through a middle point, which is far from Euler's singularity area. Problems may still occurs if the object to catch/to relocate stands exactly inside the critical area.

This is necessary in order to execute the assignment 2, due to the small size of the table, which must have the space both to spawn 11 different kinds of lego and to relocate each of them, hence it's quite hard to handle this small spaces for this amount of legos.

2.4 Authorization System

The planner can be run with the parameter -s which enable the secure authorization system. This feature lets the planner send commands to the movement module with a specific authorization code that will be verified before executing the requested command. This system needs the ACK feature in every command the planner sends.

2.5 Acknowledgement system

It is available a feature that allows you serialize all the commands the planner and the movement modules receive.
Since the communication between two different modules is done through messages with command IDs, each of these commands can request an acknowledgement message.
If you want an execution goes back to its caller, *send_ack* must be equal to **1**. It enforces the called module to complete the requested command and send back the results via event message. Viceversa if you just need the execution without caring about its completions, just leave "*send_ack*" equal to **0**.

When an ACK is read, the module saves the informations about the received command in a task structure where there are the command ID, begin time, duration time and status. The status will be busy = true as long as the process is executed. When the process is finished, the task structure gets the result of this task and sends an eventResult.msg.
Then the busy flag of the task structure returns false. While the busy flag is true, the module ignore all other packets and commands they are sent to it.

2.6 spawnLego.cpp

We developed a script that generates the lego objects on the table, which allows you to choose both position and orientation (even distances between objects). The spawner has the ability to select the assignment number and also special features.

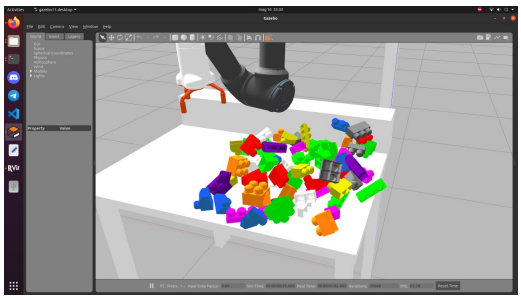

Figure 2.1: Spawn Lego Script

1. **-a1** spawn 1 object randomly in the spawn area, base in contact with the ground
2. **-a2** spawn an object for each class randomly in the spawn area, base in contact with the ground
3. **-a3** spawn multiple objects totally random, (even more than 1 object for each class)
4. **-a4** same of -a3, but without distances, so could also stand or lean on each other
5. **-s1** [**beta**] Wait the receiving of some external modules commands (Just for testing)
6. **-s2** [**beta**] Take pics (Just for testing)
7. **no parameter** Default execution: won't spawn any object.

3 Vision

This module is in charge of practice the deep learning to create an "artificial neural network" that can learn and make intelligent decisions on its own, in our case the goal is to recognize the 11 different models (legos) starting from a dataset (pool of 2500 images).
To do it we used:

- MakeSense AI
- Yolov5
- Pythorch
- Roboflow
- Custom Roboflow version of Google Colab

3.1 Steps

1 - Create Labels for the annotations Build the *label.txt* file: one class per row

0. X1-Y1-Z2
1. X1-Y2-Z1
2. X1-Y2-Z2
3. X1-Y2-Z2-CHAMFER
4. X1-Y2-Z2-TWINFILLET
5. X1-Y3-Z2
6. X1-Y3-Z2-FILLET
7. X1-Y4-Z1
8. X1-Y4-Z2
9. X2-Y2-Z2
10. X2-Y2-Z2-FILLET

2 - Create annotations Upload the images to MakeSense to get the annotations.

3 - Test and refine the dataset Once we had all the annotations, we use the Roboflow tool in order to check our dataset health and get the best split. After the upload of the images Roboflow will check all the annotations, it will split the dataset into **train**, **valid** and **test**.
Roboflow suggests you to use 640x046 format images, but it's very difficult to handle all the images in that way, but in the *generate* tab you are able to select all your preferences. A great way to operate is to follow the advices given by Roboflow. To improve the accuracy of the model there is the section *augmentation*, that performs tranforms of your existing images to create new variations and increase the number of images (flipped 90°, different brightness, etc).

Now you are ready to launch the train machine, in the *versions* tab, wait for some uploads, then the train will start. Once the train is finished (300 epochs) you can try the model by uploading for instance a video. Furthermore they provide you the possibility to export the dataset elaborated through both a .zip file and a snippet code, this one is necessary to train our model into Google Colab.

4 - Train into Google Colab provided by Ultralytics Follow the link it will open a Google Colab environment customized by roboflow, in which you will use the previous exported snippet in order to train the model in Roboflow.
An other way to do it is by uploading the .zip dataset into the same Google Drive in which you have saved the Colab instance. Once signed-in, create a copy on drive of the Colab notebook, run the first part of code (setup section) in order to install the YOLOv5 folders, import the .zip and unzip it in the dataset folder.

1. Clone the yolov5 repository and install all the necessary dependencies
2. Import dataset through the python code snippet previously copied in roboflow
3. Train the dataset with custom batch and epochs.
   We used 70 epochs and 64 batch, with *yolov5m.pt* weights.
4. Test your model and then export /runs/ folder (Colab will delete it)

3.2 Vision Results

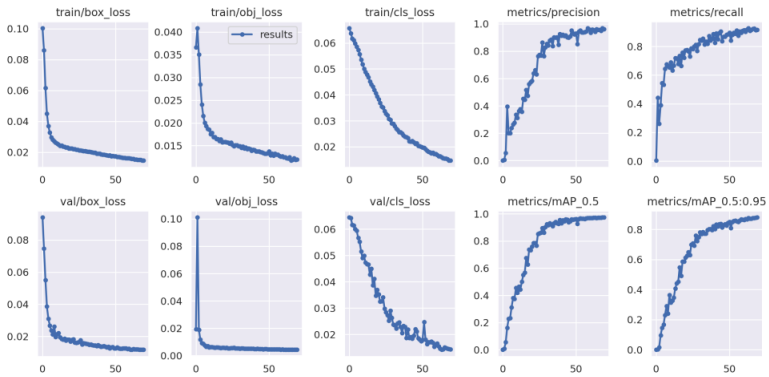

Figure 3.1: Results

3.3 Detection misses

We noticed that there are some misses during the detection, in particular running the assignment 2. Rarely it misses the objects (no detection) sometimes it recognizes the wrong class especially in case of similar geometry shapes (i.e. class 3 - 6). The main reason is the point of view of the camera Of course it is quite impossible to get 100 per cent of precision detection The solution is to increase the number of annotations of the most frequent objects missed in the dataset.
As shown in the previous slide, we got 97 per cent of accuracy so that we decide to keep this acceptable values

3.4 Detection with online computer

The Pytorch may invoke the download of Ultralytics/Yolo5 by internet if is not present in the repository. However a good connection on the executing computer is recommended.

4 Run The Project

step 1 First of all move into catkin folder with **cd Robotics_ICE23_UNITN/catkin_ws/**.
Build the packages by typing **catkin_make install**.
If you get some issues, just repeat the command until you get 100%.
(could be necessary even 4 times: fail @37%, 57%, 87%)

step 2 Go back in the main folder **cd ..** and execute the start script **bssh start.sh**.
Now you have to wait for Gazebo and Rviz to open. Once open, the UR5 robot homing procedure will start, which is shown in a dedicated terminal named *environment*.

Once you get the message "HOMING PROCEDURE ACCOMPLISHED," (̃30s) you can move to the start script terminal, after checking the correct start of the environment (on Gazebo), you can continue using **Y**. If bugs occur, just type **n**, all the ROS node will be killed and the environment will be re-started.
Then select an assignment from the list.

step 3 After choosing the assignment it will spawn the amount of lego related, then you have to select the operating way, with the possibility to execute in a *secure mode* by typing **S** or in a *normal mode* with **N**.
The script provides you the possibility to EXIT [**k**], Restart[**r**] or Rebuild and restart[**R**]

step 4 When all modules are ready, in the *vision* one you have the possibility to re-detect images using *normal* by typing **a** or the *table area only* using **t**.
If the is object correctly recognized, press **c** to continue and execute all the tasks assigned.

step 5 Once all the tasks are finished, press **k** to kill all ROS nodes and quit all the modules correctly. (This step is necessary since Gazebo looks sensitive on closing).
Here you can even re-call some modules if they are previously failed.

4.1 KPI values

We measured the key performances indicators for every modules of our project. Is important to remember that KPI values depends on the computational power of the environment and how the software was coded.
Test results in a computer with Ryzen 5 and GTX 1660 Super:

- Detection KPI: ̃1 s
- One lego relocation KPI: ̃27 s
- a 1° KPI: ̃32 s
- a 2° KPI: ̃363 s in total
- a 3° KPI: ̃204 s in total

KPI times refer to the time in the simulated world and may differ from the real time.