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 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 **Moviment**.

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

For the exam we (the group K) choosed the proposal 1

1.1 Project Proposal 1

A number of objects (e.g., mega-blocks) are stored without any specific order on a stand (initial stand) located within the workspace of a robotic manipulator. The manipulator is an anthropomorphic arm, with a spherical wrist and a three-fingered gripper as end-effector. The objects can belong to different classes but have a known geometry (coded in the STL files). The objective of the project is to use the manipulator to pick the objects in sequence and to position them on a different stand according to a specified order (final stand). A calibrated 3D sensor is used to locate the different objects and to detect their mutual position in the initial stand. The project is organised as a sequence of assignments of increasing complexity.

1.1.1 Assignment 1

There is only one object in the initial stand, which is positioned with its base “naturally” in contact with the ground. The object can be of any of the classes specified by the project. Each class has an assigned position on the final stand, which is marked by a coloured shape representing the silhouette of the object.
KPI 1-1 time to detect the position of the object
KPI 1-2 time to move the object between its initial and its final positions, counting from the instant in which both of them have been identified.

1.1.2 Assignment 2

There are multiple objects on the initial stand, one for each class. There is no specific order in the initial configuration, except that the base of the object is “naturally” in contact with the ground. Each object has to be picked up and stored in the position prescribed for its class and marked by the object’s silhouette.
KPI 2-1 Total time to move all the objects from their initial to their final positions.

1.1.3 Assignment 3

There are multiple objects on the initial stand, and there can be more than one object for each class. The objects are positioned randomly on the stand but would not stand or lean on each other. An object could be lying on one of its lateral sides or on its top. Each object has to be stored in the position prescribed by its class. Objects of the same class have to be stacked up to form a tower.
KPI 3-1 Total time to move all the objects from their initial to their final positions.

1.1.4 Assignment 4

The objects on the initial stand are those needed to create a composite object with a known design (e.g., a castle). The objects are positioned randomly on the stand. An object could be lying on one of its lateral sides or on its top. The objects could also stand or lean on each other. The manipulator has to pick them up in sequence and create the desired composite object on the final stand.

1.2 Delivery Rules

The project is developed in groups. The typical group size consists of three-four members. We can also accept groups with a smaller number of members. The group is supposed to work in perfect cooperation and the workload is required to be fairly distributed. The specific contribution of each member will be exposed during the project discussion.
The delivery phase is as follows:

1. The project will have to be tested in the laboratory with the Teaching Assistant at most 5 least five days before the exam date. During the tests, small videos can be shot and used for the presentation.
2. Each group will have to deliver the package containing the full code (with doxygen documentation and a readme for use) plus a 5-6 pages report describing
   (a) the technique used for perception
   (b) the technique used for robot motion
   (c) the technique used for high-level planning
   (d) A table with the KPI measured on Gazebo
3. The delivery deadline is three days before the exam presentation
4. On the day of the exam, the students will give a 20 minutes presentation highlighting the contribution of each member. A discussion will follow in which all members are supposed to answer questions on the entire project (regardless of her/his specific assignment within the group).
5. If allowed by the time, the group could also be asked to perform a small demo session. Otherwise, we will rely on the clip shot before the exam.

1.3 GitHub Repository

Click this link Robotics Repository or put this URL in your browser:

https://github.com/LordBions/Robotics_ICE23_UNITN

2 Set up your Environment

This project will run only in Ubuntu 20.04 Focal Fossa.

2.1 Installing ROS

If you want to execute and work with our project, you need ROS installed in your computer:

1. set up your computer in order to accept software from packages.ros.org

   ```
   sudo sh -c 'echo \
   "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" \
   > /etc/apt/sources.list.d/ros-latest.list'
   ```

2. Install curl (if you haven't already installed):

   ```
   sudo apt install curl
   ```

3. Set up your keys:

   ```
   curl -s \
   https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc \
   sudo apt-key add -
   ```

4. Run the system update command to update your existing repos and packages:

   ```
   sudo apt-get update
   ```

5. Download ROS. Here there is the suggested download:

   ```
   sudo apt install ros-noetic-desktop-full
   ```

6. Once ROS is installed, you must to set up ROS environment, typing:

   ```
   echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc
   source ~/.bashrc
   ```

2.2 Installing Python3

In order to install python3:

1. Run the system update command to update your existing repos and packages:

```
sudo apt update
```

2. Download python3

```
sudo apt install python3
```

3. check python3 version (it should be 3.8.10) using:

```
python3 --version
```

2.3 Installing and Configuring Git

To install Git:

1. Update the Ubuntu system's package repository and upgrade the APT cache as well

```
sudo apt update
sudo apt upgrade
```

2. Install Git and check the version:

```
sudo apt install git
git --version
```

To configure git:

- Configure your user name

```
git config --global user.name "Your-user-name"
```

- Configure your email

```
git config --global user.email "Your-email"
```

- Check your configuration by:

```
git config --list
```

2.4 Installing Pip3

To install pip3:

1. Run the system update command to update your existing repos and packages:

```
sudo apt update
```

2. Download pip3

```
sudo apt install python3-pip
```

3. When the installation is complete, check 'pip3' version (*it should be pip 20.0.2 from /usr/lib/python3/dist-packages/pip (python 3.8)*):

```
pip3 --version
```

2.5 Installing Catkin

To install Catkin:

1. you must have the ROS repositories which contain the '.deb' for 'catkin_tools':

```
sudo sh \
    -c 'echo "deb http://packages.ros.org/ros/ubuntu `lsb_release -sc` main" \
        > /etc/apt/sources.list.d/ros-latest.list'
```

2. then type:

```
wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
```

3. Run the system update command to update your existing repos and packages:

```
sudo apt-get update
```

4. Install 'catkin_tools'

```
sudo apt-get install python3-catkin-tools
```

If you want to set up your custom catkin workspace, here you can find a guide provided by ROS that explain you how to do it.

2.6 Installing Locosim

A didactic framework to learn/test basic controllers schemes on quadruped robots that is continuosly update. To install Locosim you can follow the guide here

2.7 Yolov5

In order to install Yolov5:

1. Go to your HOME directory and clone the Yolov5 on GitHub

```
cd ~
git clone https://github.com/ultralytics /yolov5
```

2. Go inside the folder and install all requirements:

```
cd yolov5
pip3 install -r requirements.txt
```

2.8 Pytorch

To install PyTorch:

1. Update the Ubuntu system's package repository and upgrade the APT cache as well

```
sudo apt update
sudo apt upgrade
```

2. If the instance to be used supports GPU/NVIDIA CUDA cores, and the PyTorch applications that you're using support CUDA cores, install the NVIDIA CUDA Toolkit

```
sudo apt install nvidia-cuda-toolkit
```

3. use pip3 to install pythorch with CPU support only

```
pip3 install torch==1.9.1+cpu torchvision==0.10.1+cpu -f \
https://download.pytorch.org/whl/torch_stable.html
```

4.

2.9 Installing Robotics_ICE23_UNITN

Robotics_ICE23_UNITN is the project that we made for Fundamentals of Robotics exam. If you are interested you can download and use it, follow this guide from GitHub

3 Simulation Environment

3.1 ROS

ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. ROS is similar in some respects to "robot frameworks". reference

3.2 Catkin

Catkin packages can be built as a standalone project, in the same way that normal cmake projects can be built, but catkin also provides the concept of workspaces, where you can build multiple, interdependent packages together all at once. A catkin workspace is a folder where you modify, build, and install catkin packages. reference

```
workspace_folder/         -- WORKSPACE
  src/                    -- SOURCE SPACE
    CMakeLists.txt        -- The 'toplevel' CMake file
    package_1/
      CMakeLists.txt
      package.xml
      ...
    package_n/
      CATKIN_IGNORE       -- Optional empty file to exclude package_n from being processed
      CMakeLists.txt
      package.xml
      ...
  build/                  -- BUILD SPACE
    CATKIN_IGNORE         -- Keeps catkin from walking this directory
  devel/                  -- DEVELOPMENT SPACE (set by CATKIN_DEVEL_PREFIX)
    bin/
    etc/
    include/
    lib/
    share/
    .catkin
    env.bash
    setup.bash
    setup.sh
    ...
  install/                -- INSTALL SPACE (set by CMAKE_INSTALL_PREFIX)
    bin/
    etc/
    include/
    lib/
    share/
    .catkin
    env.bash
    setup.bash
    setup.sh
    ...
```

Figure 3.1: Structure of Catkin workspace

3.2.1 Source Space

The source space contains the source code of catkin packages. This is where you can extract/checkout/clone source code for the packages you want to build. Each folder within the source space contains one or more catkin packages. This space should remain unchanged by configuring, building, or installing. The root of the source space contains a symbolic link to catkin's boiler-plate 'toplevel' CMakeLists.txt file. This file is invoked by CMake during the configuration of the catkin projects in the workspace. It can be created by calling catkin_init_workspace in the source space directory.

3.2.2 Build Space

The build space is where CMake is invoked to build the catkin packages in the source space. CMake and catkin keep their cache information and other intermediate files here. The build space does not have to be contained within the workspace nor does it have to be outside of the source space, but this is recommended.

3.2.3 Development (Devel) Space

The development space (or devel space) is where built targets are placed prior to being installed. The way targets are organized in the devel space is the same as their layout when they are installed. This provides a useful testing and development environment which does not require invoking the installation step. The location of the devel space is controlled by a catkin specific CMake variable called CATKIN_DEVEL_PREFIX, and it defaults to <build space>/develspace. This is the default behavior because it might be confusing to CMake users if they invoked cmake .. in a build folder and that modified things outside of the current directory. It is recommended, however, to set the devel space directory to be a peer of the build space directory.

3.2.4 Install Space

Once targets are built, they can be installed into the install space by invoking the install target, usually with make install. The install space does not have to be contained within the workspace. Since the install space is set by the CMAKE_INSTALL_PREFIX, it defaults to /usr/local, which you should not use.

3.2.5 Result space

When ever referring to a folder which can either be a development space or an install space the generic term result space is used.

3.3 Locosim

Locosim is a didactic framework to learn/test basic controllers schemes on quadruped robots and manipulators (UR5). It is composed by a roscontrol node called ros_impedance_controller (written in C++) that interfaces a python ROS node (where the controller is written) to a Gazebo simulator. For each controller, plotting / logging utilities are available to evaluate the results together with a configuration file (LX_conf.py) to change the controller parameters. Github Reference

3.4 Gazebo

Gazebo brings a fresh approach to simulation with a complete toolbox of development libraries and cloud services to make simulation easy. Iterate fast on your new physical designs in realistic environments with high fidelity sensors streams. Test control strategies in safety, and take advantage of simulation in continuous integration tests. It is composed by a set of open source development libraries, which encapsulate all the essentials, such as common math data types, logging, 3D mesh management, and asynchronous message passing. reference

Figure 3.2: Our simulation running on Gazebo

3.5 Rviz

Rviz is a virtualization environment that let us view what the robot is seeing, thinking and doing. Programming a robot can be very difficult without the possibility of debugging. There are two main ways to put data inside Rviz world (the combination of) ;

- Rviz understand sensors (like laser scans, cameras, corner frames)
- visualization markers (cubes, arrows, lines, colours, etc)

reference

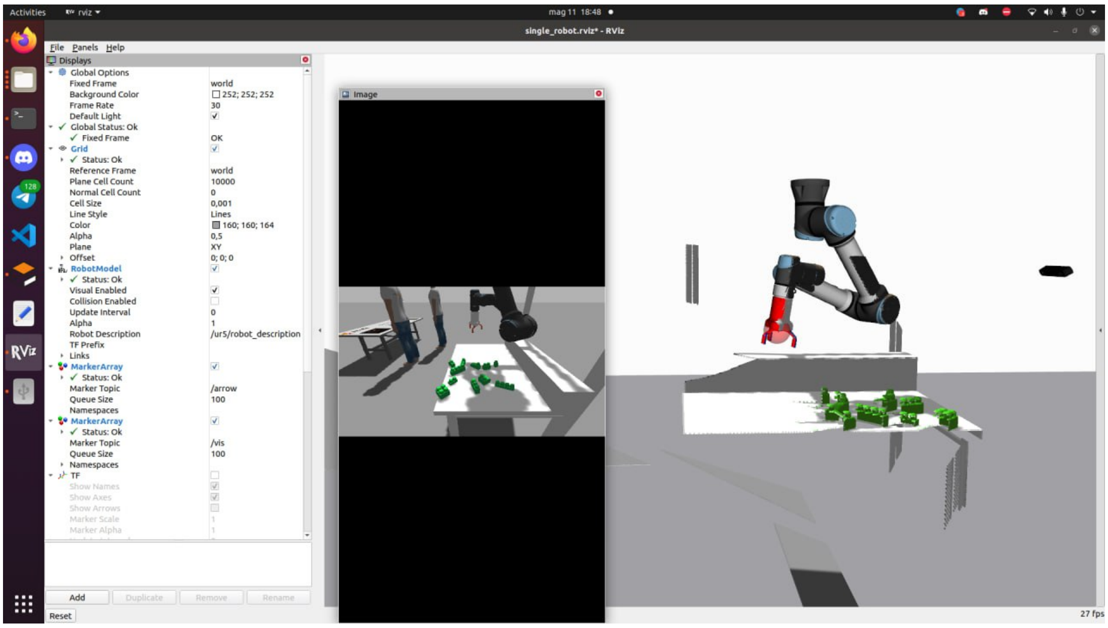


Figure 3.3: Our simulation running on Rviz

The black background window named "*image*" is the point of view of the ZED Camera, it's exactly what it is seeing.

4 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

4.1 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*, as shown in the figure 4.2.

4.1.1 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

No command: 0 The planner will ignore any packet sent by the vision module with command ID 0, the packets will be discarded.

Detect: 1 The planner receive a packet that contains all the info about the detection in charge of the vision module (e.g. classes, position, etc)

Quit: 2 Once the vision module has finished its tasks, the planner will no longer be needed, hence the it can be closed.

4.1.2 motion/src/planner.cpp

```cpp
void subDetectCommanderCallback(const motion::legoFound::ConstPtr &msg_detect); // reception and analysis of a received request
void pubDetectResulter(int risultato); // publish the result
void pubTaskCommander(bool s_ack); // publish the task
void subTaskResulterCallback(const motion::eventResult::ConstPtr &msg_event); // read ack sent by movement

Eigen::Vector3f camera2SimulationR(Eigen::Vector3f simul_lego_pos); // adapt camera point values in virtual world for robot
Eigen::Vector3f camera2RealR(Eigen::Vector3f camera_lego_pos); // adapt camera point values in real world for robot
void ungraspCommand(); // Enlarge robot fingers
void homingCommand(); // move the arm to default to avoid camera interferences

void catchCommand(Eigen::Vector3f position); // send the complete catch command to the moviment module

void selectClass(int lego_cl);  // In relation to lego received from vision, load right lego parameters
double getTimeNow(); // Returns the current time
double getInterval(double start_t); // Returns the difference between the currentTime and the start time
```

Figure 4.1: Planner Functions Headers

```cpp
// all classes relocation
#define class_00_relocation 0.4, -0.000, 0.82
#define class_01_relocation 0.4, -0.046, 0.82
#define class_02_relocation 0.4, -0.092, 0.82
#define class_03_relocation 0.4, -0.138, 0.82
#define class_04_relocation 0.4, -0.022, 0.82
#define class_05_relocation 0.4, -0.230, 0.82
#define class_06_relocation 0.4, -0.276, 0.82
#define class_07_relocation 0.4, -0.322, 0.82
#define class_08_relocation 0.4, -0.368, 0.82
#define class_09_relocation 0.4, -0.414, 0.82
#define class_10_relocation 0.4, -0.460, 0.82

// custom orientation
#define class_00_orient 0.0, 0.0, 0.0
#define class_01_orient 0.0, 0.0, 0.0
#define class_02_orient 0.0, 0.0, 0.0
#define class_03_orient 0.0, 0.0, 0.0
#define class_04_orient 0.0, 0.0, 0.0
#define class_05_orient 0.0, 0.0, 0.0
#define class_06_orient 0.0, 0.0, 0.0
#define class_07_orient 0.0, 0.0, 0.0
#define class_08_orient 0.0, 0.0, 0.0
#define class_09_orient 0.0, 0.0, 0.0
#define class_10_orient 0.0, 0.0, 0.0
```

Figure 4.2: Final Object Position in coordinates

4.1.3 motion/msg/legoFound.msg

int32 command_id
int32 lego_class
float64 coord_x
float64 coord_y
float64 coord_z
float64 rot_roll
float64 rot_pitch
float64 rot_yaw
float64 date_time
string comment
int32 send_ack

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.

4.2 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.

4.2.1 motion/src/movement.cpp

```cpp
// struct used to coordinate messages between planner and movement
struct ExecutingTask {
        int command_id;         // used to understand which task movement must execute
        double request_time;    // time to complete the task
        bool busy;              // It shows if the movement is performing a task or not
};
ExecutingTask planner_eseguendo;

Eigen::VectorXf joint_state_vector(6);
Eigen::VectorXf gripper_state_vector(3);

/*-----------------------------------------function headers-----------------------------------------*/

// reception and analysis of a received request. If it find an ack request in the end, it will publish eventResult message
void taskCommanderCallback(const motion::legoTask::ConstPtr &msg_taskCommand);
void pubTaskResulter(int risultato);    // publish

// Returns the Trajectory vector using time and positions (start and end positions)
Eigen::Vector3f getTrajectory(double time, Eigen::Vector3f begin_position, Eigen::Vector3f final_position);
// it returns the joints speed
Eigen::VectorXf getJointSpeeds(Eigen::VectorXf joint_st, Eigen::Vector3f curr_position, Eigen::Vector3f destin_position, Eigen::Vector3f

void updateJointStates(); // update joint state
float gripper2joints(float diameter); // returns the converted diameter of fingers
void nullCommandExecute(); // used when motion has no command to execute
void waitCommandExecute(int wait_time); // used to wait a command from planner
void moveProcedure(Eigen::Vector3f v_position, Eigen::Vector3f v_orientation, float dt); // used to move the arm
void graspObject(bool catchIt); // used to grasp objects (open-close fingers)
void moveDefaultPosition(); // move the arm to a default position
void fastCatchProcedure(); // used when the command received is command_fast_catch: goes to the object in a fast way
void catchProcedure(); // used when the command received is command_catch:  goes to the object using a full protocol
double getTimeNow(); // return the current time
double getInterval(double start_t); // return the interval between the start time and the current time
```

Figure 4.3: Movement Functions Headers

4.2.2 motion/msg/legoTask.msg

int32 command_id
int32 real_robot
float64 coord_x
float64 coord_y
float64 coord_z
float64 rot_roll
float64 rot_pitch
float64 rot_yaw
float64 gasp_diam
float64 dest_x
float64 dest_y
float64 dest_z
float64 dest_roll
float64 dest_pitch
float64 dest_yaw
float64 ungasp_diam
int32 w_time
float64 date_time
string comment
int32 send_ack

4.2.3 Movement commands from the planner module

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

No command: 0 Like the command ID 0 from the vision, the same for the movement it will discard the packet with ID = 0.

Test: 1 It was used in order to check if the movement received correctly the packets

Wait: 2 It is simply a timer that waits for seconds set in the specific field in the Lego-Task.msg.

Move: 3 It sends a packets contains a trajectory which declare the start-point and the end-point.

Grasp: 4 it assigns a parameter to close the grasper until the final measure will be equal to the received parameter.

Ungrasp: 5 It is the opposite of the previous command (grasp -4), hence it will open the grasper until the final measure will be the assigned one.

Default position: 6 It makes the arm move to a specific preset default position.

Fast catch: 7 It makes the arm move from the actual position directly to the object, catches it and relocate in the specific position, skipping both the passage for the central point and the ascent and discent movements. Thanks to this command it's possible to move the object faster.

Catch: 8 This is the most detailed one, since it is composed by multiple commands:

- it moves the arm passing from a central point
- it move over the object selected
- it goes down towards the object
- it catch the object (command 4)
- it goes up passing from a central point
- it moves to the final destination (figure 4.2)
- it goes down and left the object (command 5)
- it returns in the default position (command 6)

4.3 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*, kinetics.h includes the necessary classes and functions for dense matrix operations, it provides a wide range of functionalities, including matrix algebraic operations.

```
// It calculates the jacobian Matrix related to the change of the reference system
Eigen::MatrixXf jacobMatrix(Eigen::VectorXf jo_ang) {

    Eigen::VectorXf v_1(6);
    v_1 << 0, -0.425, -0.3922, 0, 0, 0;

    Eigen::VectorXf v_2(6);
    v_2 << 0.1625, 0, 0, 0.1333, 0.0997, 0.0996 + 0.14;

    Eigen::MatrixXf jac_mtx_1(6, 1);
    jac_mtx_1 <<    v_2(4) * (cos(jo_ang(0)) * cos(jo_ang(4)) + cos(jo_ang(1) + jo_ang(2) + jo_ang(3)) * sin(jo_ang(0)) * sin(jo_ang(4))) + v_2(2) * cos(jo
                    v_2(4) * (cos(jo_ang(4)) * sin(jo_ang(0)) - cos(jo_ang(1) + jo_ang(2) + jo_ang(3)) * cos(jo_ang(0)) * sin(jo_ang(4))) + v_2(2) * sin(jo
                    0,
                    0,
                    0,
                    1;

    Eigen::MatrixXf jac_mtx_2(6, 1);
    jac_mtx_2 <<    -cos(jo_ang(0)) * (v_1(2) * sin(jo_ang(1) + jo_ang(2)) + v_1(1) * sin(jo_ang(1)) + v_2(4) * (sin(jo_ang(1) + jo_ang(2)) * sin(jo_ang(3)
                    -sin(jo_ang(0)) * (v_1(2) * sin(jo_ang(1) + jo_ang(2)) + v_1(1) * sin(jo_ang(1)) + v_2(4) * (sin(jo_ang(1) + jo_ang(2)) * sin(jo_ang(3)
                    v_1(2) * cos(jo_ang(1) + jo_ang(2)) - (v_2(4) * sin(jo_ang(1) + jo_ang(2) + jo_ang(3) + jo_ang(4))) / 2 + v_1(1) * cos(jo_ang(1)) + (v_
                    sin(jo_ang(0)),
                    -cos(jo_ang(0)),
                    0;

    Eigen::MatrixXf jac_mtx_3(6, 1);
    jac_mtx_3 <<    cos(jo_ang(0)) * (v_2(4) * cos(jo_ang(1) + jo_ang(2) + jo_ang(3)) - v_1(2) * sin(jo_ang(1) + jo_ang(2)) + v_2(4) * sin(jo_ang(1) + jo_a
                    sin(jo_ang(0)) * (v_2(4) * cos(jo_ang(1) + jo_ang(2) + jo_ang(3)) - v_1(2) * sin(jo_ang(1) + jo_ang(2)) + v_2(4) * sin(jo_ang(1) + jo_a
                    v_1(2) * cos(jo_ang(1) + jo_ang(2)) - (v_2(4) * sin(jo_ang(1) + jo_ang(2) + jo_ang(3) + jo_ang(4))) / 2 + (v_2(4) * sin(jo_ang(1) + jo_
                    sin(jo_ang(0)),
                    -cos(jo_ang(0)),
```

Figure 4.4: Kinetics.h

4.4 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, the begin time, the duration time and the 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.
After this, the busy flag of the task structure return to false. While the busy flag is true, the module ignore all other packets and commands they are sent to it.

This feature is useful to get the KPI of every task the modules performs.

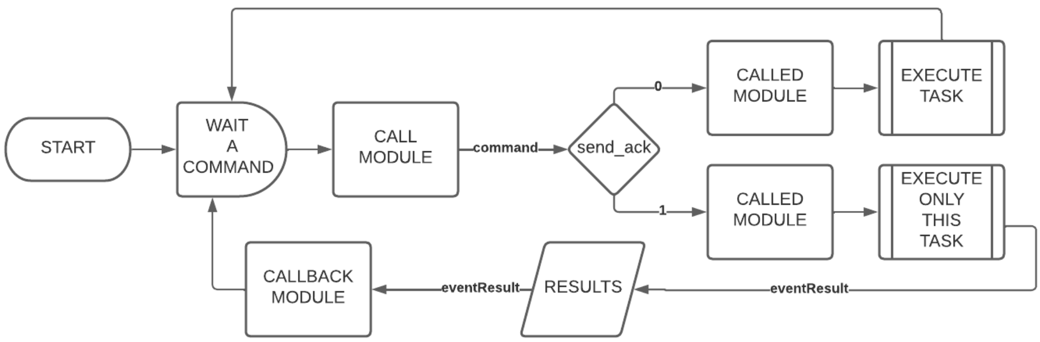


Figure 4.5: ACK Flow

4.5 Authorization system

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

To start the auth system, the planner has to begin an handshake with the movement module. Here the first sequence:

- The planner create its communication key using its hidden key and a random number generated at time.
- The planner encode its communication key with his pre-shared key (also the movement has the planner pre-shared key)
- The planner sends the encoded communication key to the movement module by LegoTask.msg
- The movement decode it using the known planner pre-shared key and saves the planner communication key for future use.
- The movement generates its communication key using its hidden key and a random number generated at the moment.
- Then the movement encode its communication key with its pre-shared key.
- The movement sends the encoded key to the planner which decodes it by the known movement pre-shared key and stores it for future use.

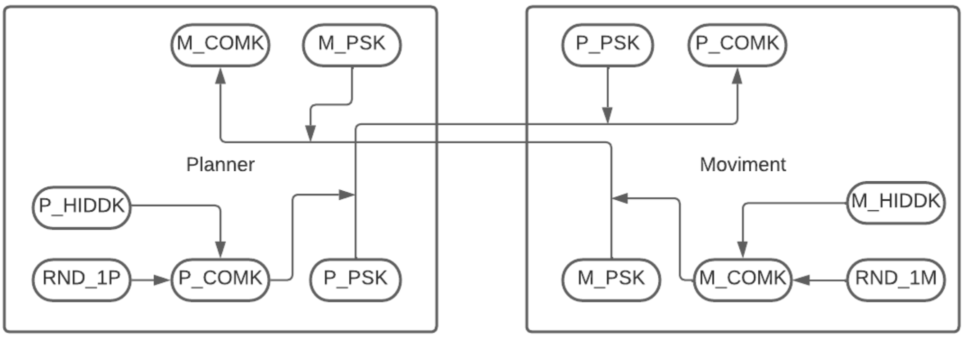


Figure 4.6: Handshake part 1

After the two modules shared their communication keys, the second part of the handshake begins:

- The planner generate a random number to use as authorization code encoder.
- The planner encodes the random number with its communication key and sends it to the movement.
- The movement module decode the encoded random numebr with the planner communication key and saves it.
- Then the movement generates its authorization key with a random function.
- The movement encodes its authorization key with its communication key then sends it to the planner.
- The planner decode the encoded auth key with the movement communication key and saves it for future use.

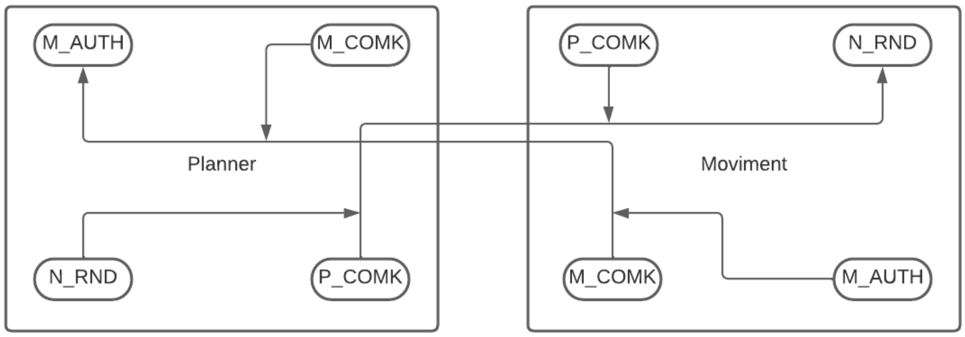


Figure 4.7: Handshake part 2

Now, after the two modules shared the authorization key and the random encode key, the handshake is concluded and the two modules can works. A normal authorization session is explained here:

- The planner generates an authorization code using the movement auth key encoded with the random key from the last handshake part.
- the planner encode the auth code with its communication key and sends it to the movement.
- The movement module decodes it with the planner communication key.
- Then the movement re-decode the decoded auth code with the random encode key form the last handshake part.
- The previous operation lets the movement check if its auth key and the planner auth code are the same.
- If positive, the movement let the requested command and after generates a new random encode key for the next auth code.
- The generated next random key is encoded with the movement communication key and sends it to the planner.
- The planner then decodes it using the movement communication key and stores the random encoder key for the next auth code
- Then the cycle restarts.

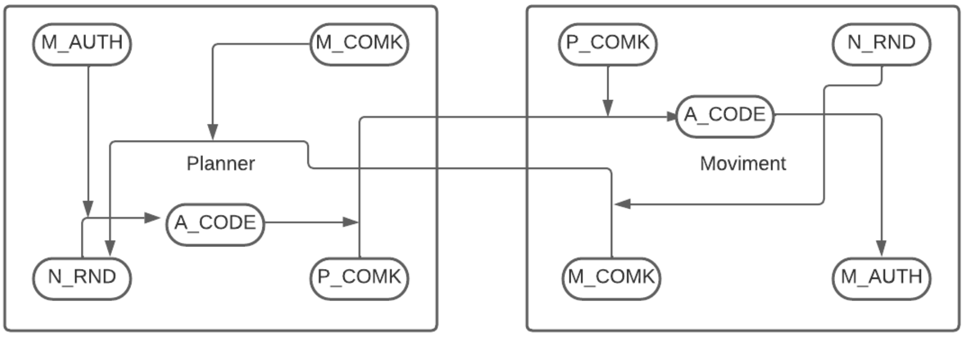


Figure 4.8: Normat auth session

4.6 spawnLego.cpp

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

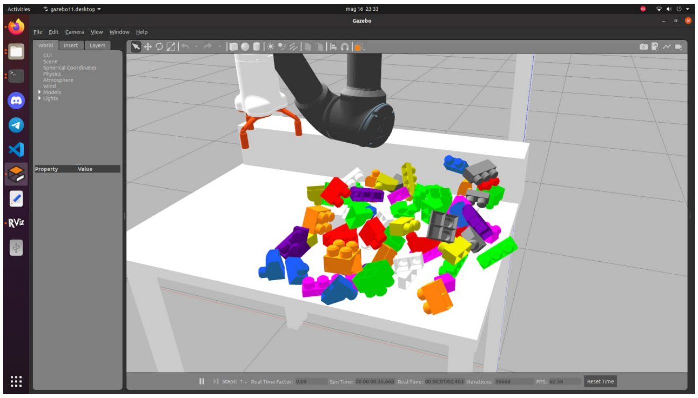


Figure 4.9: Spawn Lego Script

-a1 parameter The script will spawn a unique object randomly in the spawn area, which is positioned with its base "naturally" in contact with the ground (fixed orientation).

-a2 parameter The script will spawn an object for each class randomly in the spawn area, which is positioned with its base "naturally" in contact with the ground (fixed orientation, except for the base).

-a3 parameter The script will spawn multiple objects and there can be more than one object for each class. The objects are positioned totally random (i.e. random position, random orientation, therefore can lying on one of its lateral sides), but the distance between two objects must be keept.

-a4 parameter The script will do the same of -a3, but without distances, so could also stand or lean on each other.

-s1 parameter [beta] The script will wait the receiving of some external modules commands.
Note: It was used only for testing.

-s2 parameter [beta] Used to take pictures
Note: It was used only for testing.

no parameter Default execution: won't spawn any object.

5 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).

5.1 Tool used

5.1.1 Yolov5

We have used YOLOv5, namely *You Only Look Once* as model for object detection.

YOLOv5 is a family of compound-scaled object detection models trained on the COCO dataset, and includes simple functionality for Test Time Augmentation (TTA), model ensembling, hyperparameter evolution, and export to ONNX, CoreML and TFLite.
It accepts URL, Filename, PIL, OpenCV, Numpy and PyTorch inputs, and returns detections in torch, pandas, and JSON output formats.
On June 25th, 2020, the first official version of YOLOv5 was released by Ultralytics, a computer vision model used for detecting objects. reference

5.1.2 Pytorch

PyTorch is a fastest-growing Python machine learning framework based on the Torch library, used for applications such as computer vision and natural language processing, originally developed by Meta AI and now part of the Linux Foundation umbrella.It is free and open-source software released under the modified BSD license. Although the Python interface is more polished and the primary focus of development, PyTorch also has a C++ interface. reference
PyTorch provides two high-level features:

- Tensor computing with strong acceleration via graphics processing units (GPU)
- Deep neural networks built on a tape-based automatic differentiation system

5.1.3 Custom Roboflow version of Google Colab

In the official Ultralytics Github Repository there's a link that refers to Google Colab environment, where it is possible to use all the YOLOv5's tools in order to train the machine learning. We used the Roboflow version for some useful automations.
In this environment we had to upload the folder containing our images and values obtained using MakeSense or via Google Drive.

5.1.4 MakeSense AI

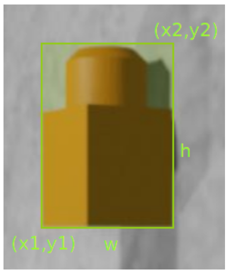
MakeSense AI is a free-to-use online tool for labeling photos. Thanks to the use of a browser it does not require any complicated installation - just visit the website and you are ready to go. It also doesn't matter which operating system you're running on - we do our best to be truly cross-platform. It is perfect for small computer vision deeplearning projects, making the process of preparing a dataset much easier and faster. Prepared labels can be downloaded in one of multiple supported formats. The application was written in TypeScript and is based on React/Redux duo. reference
Thanks to this tool we got the labels of all the dataset images.

The values of labels are obtained framing (selecting by hand a specific area of an image) and assigning to this frame a specific class, that is the name of the object family.
These values are stored into files called *annotations*, that are exportable in csv, xml or txt. The format depends on the platform that will be used for training (e.g. Roboflow uses .xml, instead yolov5 uses .txt)

obj class	X-position	Y-position	width	heigh

```
8 0.471170 0.624689 0.064544 0.105565
2 0.556368 0.553548 0.033563 0.064257
0 0.615318 0.621629 0.027539 0.065787
5 0.441910 0.558137 0.043890 0.070377
```

Figure 5.1: YOLOv5 labels values - Example



- For each images, a new file is created
- For each object, a new line is created
- For each different object in the image that we want to classify, we must define a new class, so in the example above we have 4 different objects selected.
- **<object-class>**: integer number of object { 0 - (classes-1) }
- **<x> <y> <width> <height>**: float values relative to width and height of image, it can be equal from (0.0 to 1.0]
  - For example: <x> = <absolute_x> / <image_width> or <height> = <absolute_height> / <image_height>
  - Attention: <x> <y> - are center of rectangle (are not top-left corner)

5.2 Steps

5.2.1 Create Dataset

First of all you have to create your dataset with a large pool of images (order of 1500/1700 images). The choices can't be made at random, you have to build a balanced dataset, it means that the number of images containing objects of class 1 must be similar to the number of images containing other classes. (No 1000 images of class 1 and 50 of class 3). Another important aspect is the variety of the data, that is in order to get the best results it's high recommended to use images with different lights, shadows and spawn environment.

5.2.2 Dataset structure

If you want to build a reliable machine learning model, you need to split your dataset into the training, validation, and test sets. If you don't, your results will be biased, and you'll end up with a false impression of better model accuracy.
For training and testing purposes of our model, we should have our data broken down into three distinct dataset splits.

Training Set It's the set of data that is used to train and make the model learn the hidden features/patterns in the data. In each epoch, the same training data is fed to the neural network architecture repeatedly, and the model continues to learn the features of the data. The training set should have a diversified set of inputs so that the model is trained in all scenarios and can predict any unseen data sample that may appear hereafter.

Validation Set The validation set is a set of data, separate from the training set, that is used to validate our model performance during training. This validation process gives information that helps us tune the model's hyperparameters and configurations accordingly. It is like a critic telling us whether the training is moving in the right direction or not. The model is trained on the training set, and, simultaneously, the model evaluation is performed on the validation set after every epoch. The main idea of splitting the dataset into a validation set is to prevent our model from overfitting i.e., the model becomes really good at classifying the samples in the training set but cannot generalize and make accurate classifications on the data it has not seen before.

Test Set The test set is a separate set of data used to test the model after completing the training. It provides an unbiased final model performance metric in terms of accuracy, precision, etc. It answers the question of "How well does the model perform?"

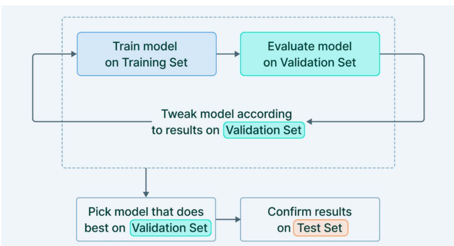


Figure 5.2: Train Vs. Validation Vs. Test

How to split the images

The creation of different samples and splits in the dataset helps us judge the true model performance. The dataset split ratio depends on the number of samples present in the dataset and the model. Some common inferences that can be derived on dataset split include:

- If there are several hyperparameters to tune, the machine learning model requires a larger validation set to optimize the model performance. Similarly, if the model has fewer or no hyperparameters, it would be easy to validate the model using a small set of data.
- If a model use case is such that a false prediction can drastically hamper the model performance—like falsely predicting cancer—it's better to validate the model after each epoch to make the model learn varied scenarios.
- With the increase in the dimension/features of the data, the hyperparameters of the neural network functions also increase making the model more complex. In these scenarios, a large split of data should be kept in training set with a validation set.

There is no optimal split percentage. One has to come to a split percentage that suits the requirements and meets the model's needs. However, there are two major concerns while deciding on the optimum split:

- If there is less training data, the machine learning model will show high variance in training.
- With less testing data/validation data, your model evaluation/model performance statistic will have greater variance.

Essentially, you need to come up with an optimum split that suits the need of the dataset/-model, but here's the rough standard split that you might encounter.

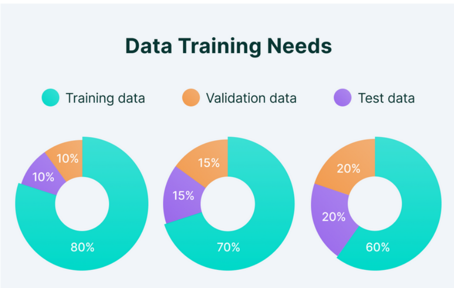


Figure 5.3: Recommended split

5.2.3 Create Labels for the annotations

We will use the MakeSense AI web tool in order to get the annotations. For each object that we want to detect we must associate to him a label, that will be then called class, since we have 11 types of legos: (an alternative is to load 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

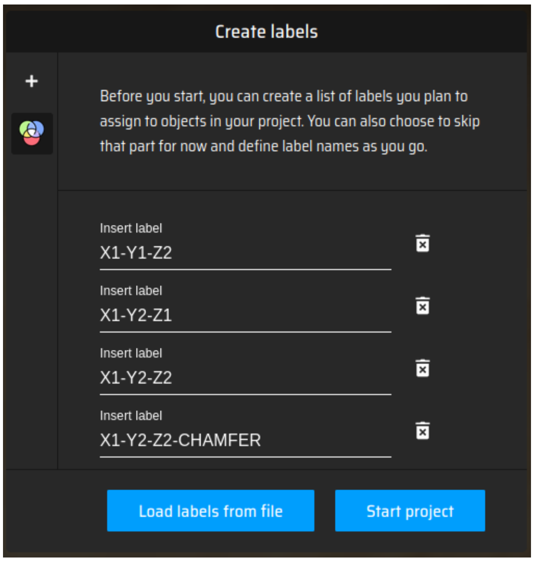


Figure 5.4: Labels in MakeSense AI

5.2.4 Create annotations

Once you have your data folders splitted in the way explained above, you can upload them to MakeSense AI in order to get the annotations. Keep in mind that you must be consistent in your choices (if a object is partial hidden, and the first time that occurs this situation you selected the hypothetic area, you must select in this way even for the next occurences). Of course you must frame all the objects in a image, without forget anyone, so label every instance of every class.

5.2.5 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 tu 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 (same images flipped 90°, different brightness, etc).

5.2.6 Roboflow Dataset Settings

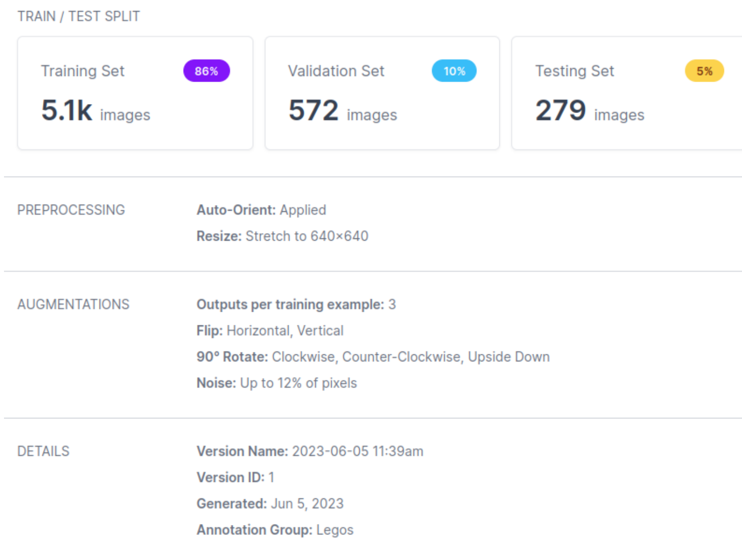


Figure 5.5: Dataset Settings

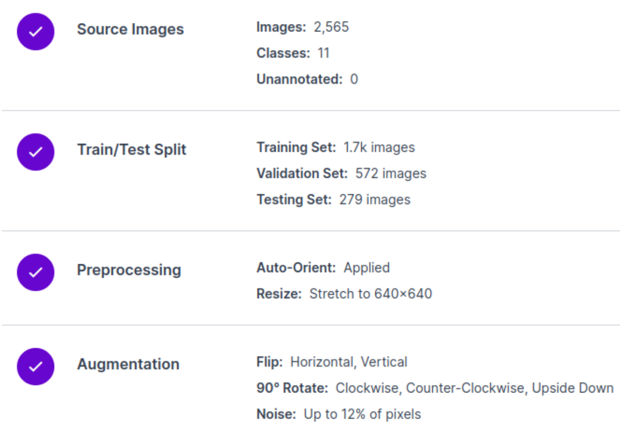


Figure 5.6: Additional settings

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.

5.2.7 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. (Note that you can have only one instance per account, furthermore the session will expire in 6 hours).
An alternative is to go in the official Ultralytics Github Repository, in the *README.md*, section *Environment* you will find a link to the Google Colab environment.
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.

Step 1 Clone the yolov5 repository and install all the necessary dependencies.

Step 2 Import dataset through the python code snippet previously copied in the roboflow platform, for instance our snippet is:

from roboflow import Roboflow
rf = Roboflow(api_key="1fFz7gZkuXvcnACwWiwR")
project = rf.workspace("roboticsiceunitn").project("robotics_ice23_unitn")
dataset = project.version(1).download("yolov5")

Step 3 Train the dataset with custom batch and epochs.
We used 70 epochs and 64 batch, with *yolov5m.pt* weights.

Step 4 Test your model and then export /runs/ folder using a compressor (e.g. Zip) and save it in local because Colab will delete it.

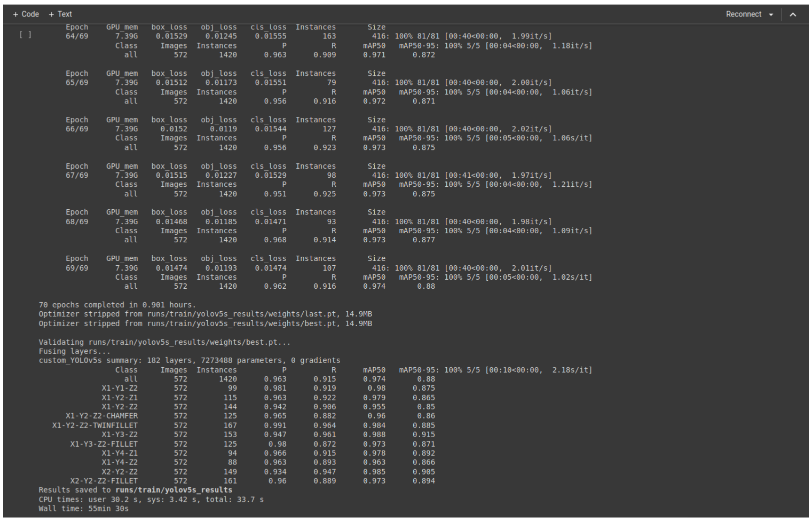


Figure 5.7: Colab results after training

5.3 Results

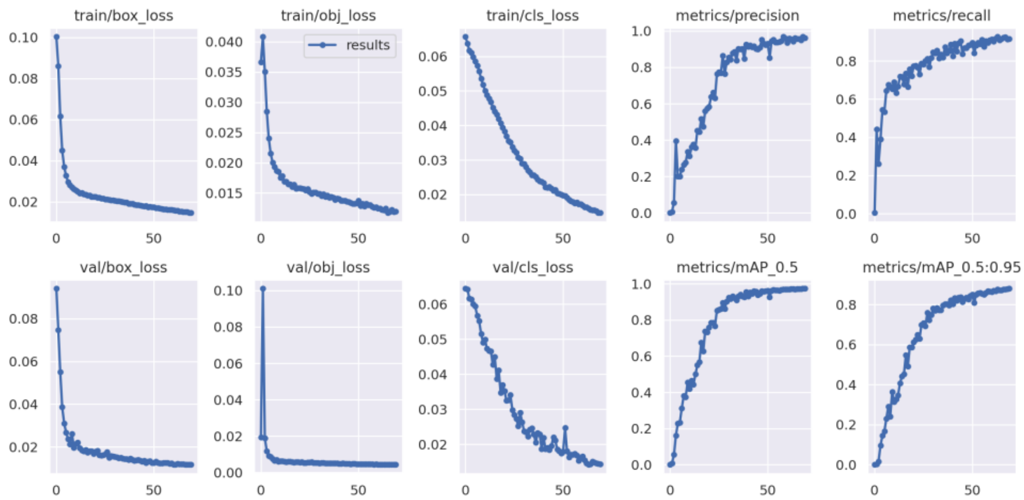


Figure 5.8: Results

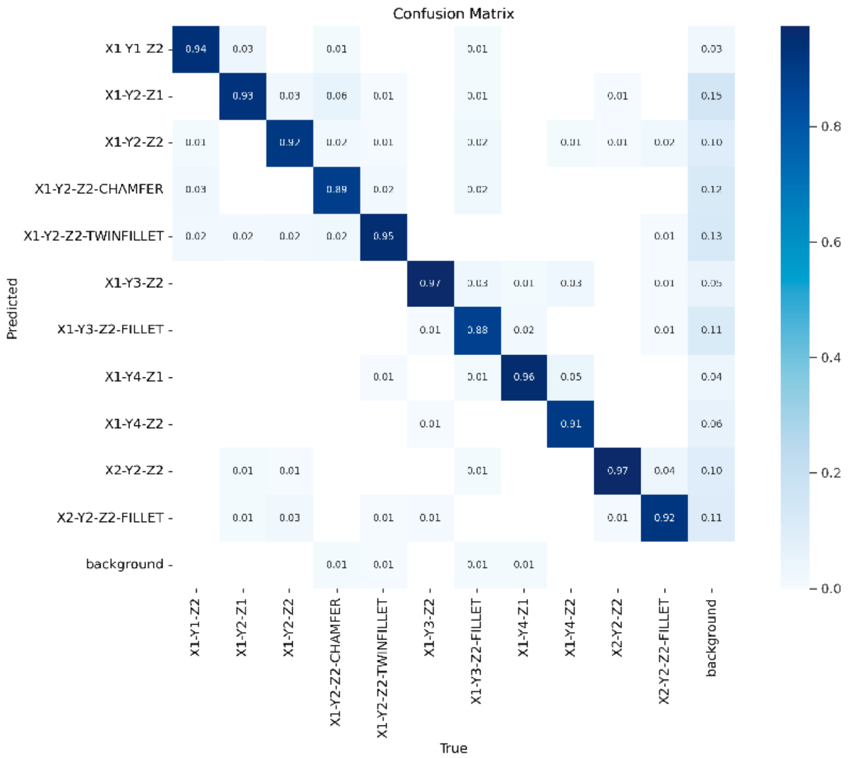


Figure 5.9: Confusion Matrix

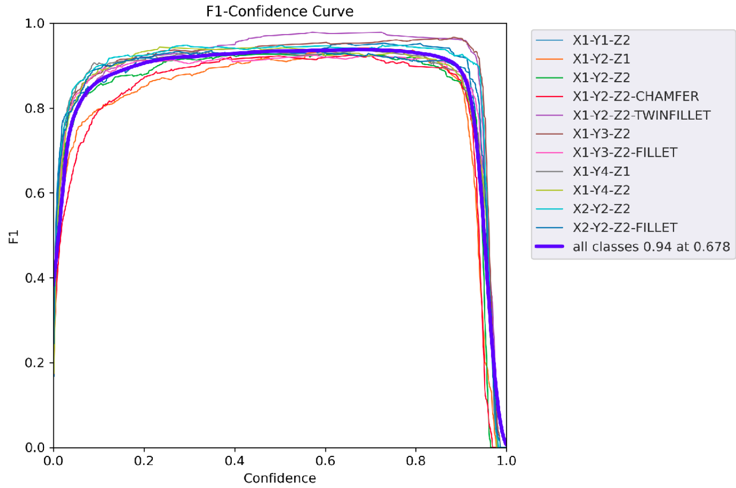


Figure 5.10: F1 Confidence Curve

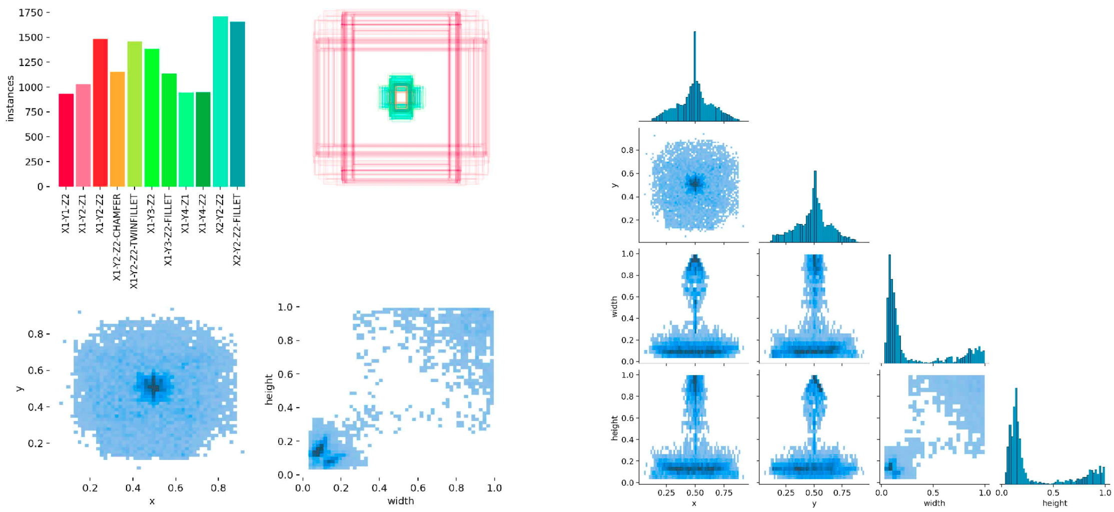


Figure 5.11: Labels & Labels Correlogram

5.4 Dataset

Our dataset in Roboflow available here
In the Colab platform, more in detail in the folder:
/catkin_ws/src/vision/yolov5_22/runs/train/yolov5s_results/ you can find the results.
Obviously, due to the large size of our dataset, we won't include it inside the repository.
Note: we needed 22 attemps before get a good result. Drive with the previous weights.

5.5 In practice

5.5.1 Instance of assignment 2

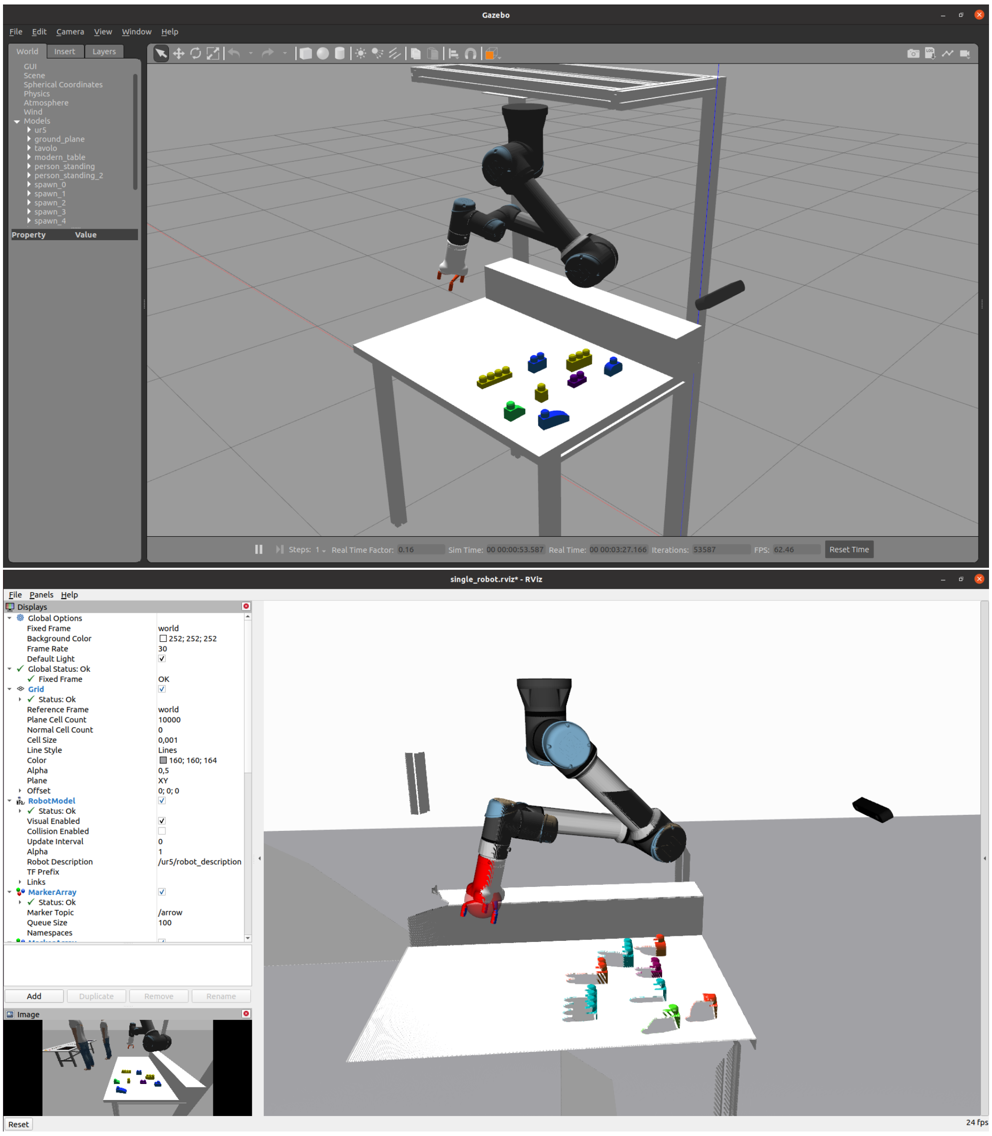


Figure 5.12: Instance of assignment 2

5.5.2 Normal Vision Results

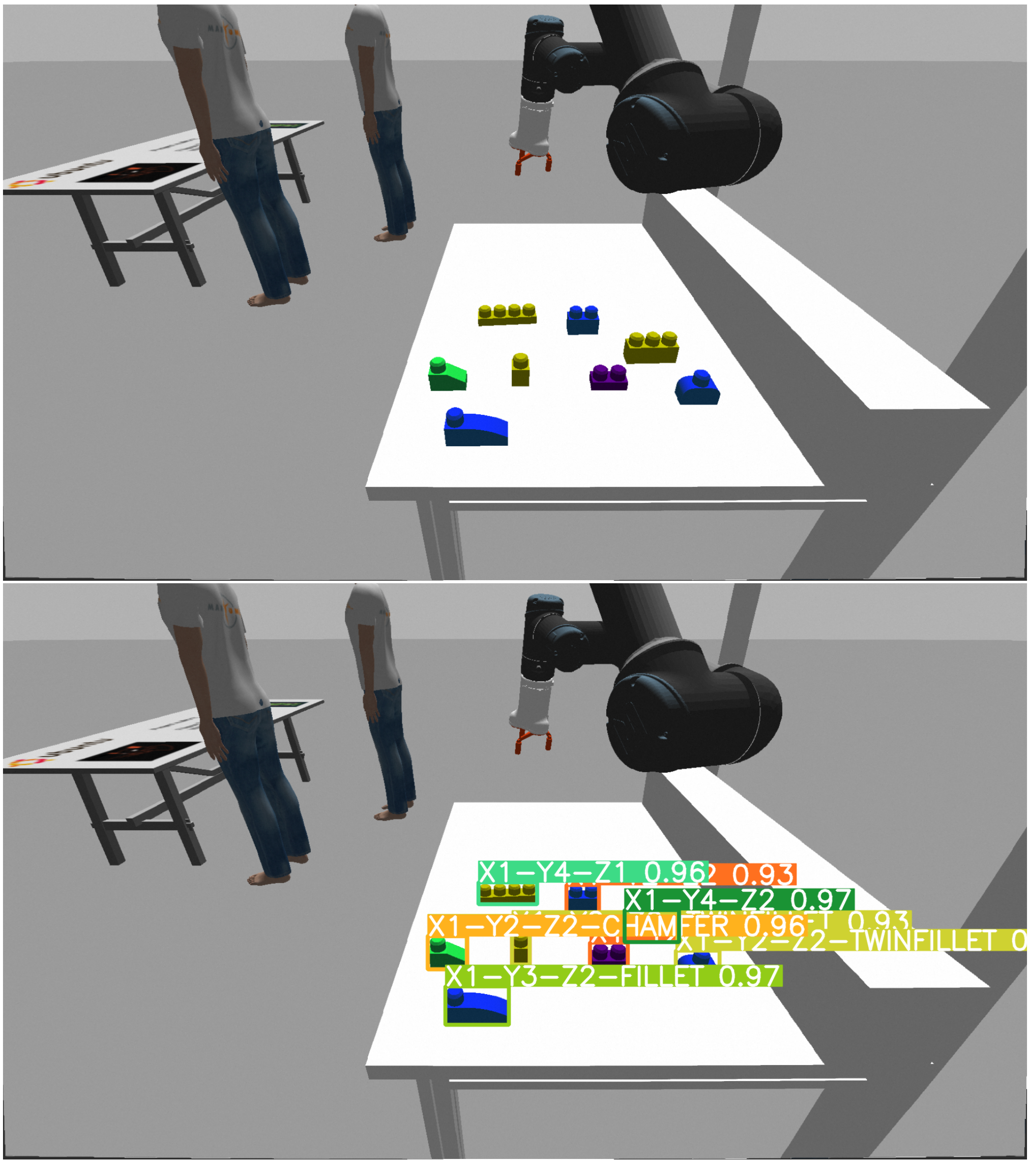


Figure 5.13: Normal Vision

5.5.3 ROI Vision Results

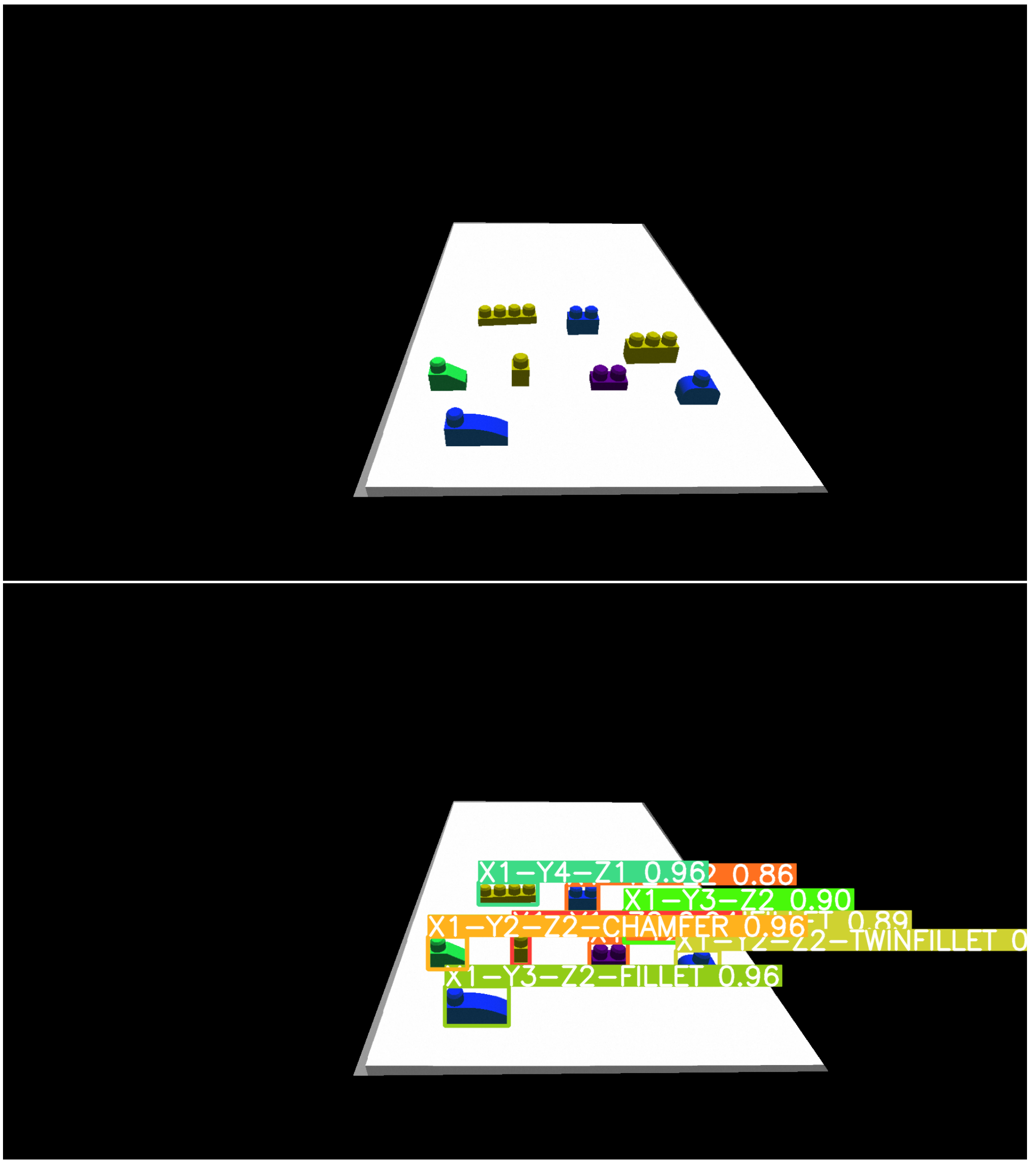


Figure 5.14: ROI Vision

6 Run The Project

After all the configuration you are able to 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," 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.

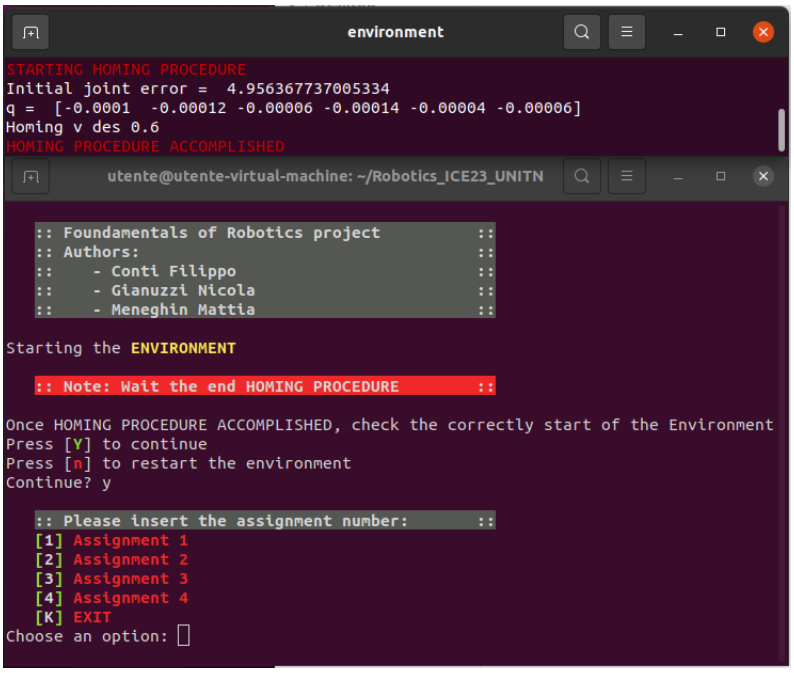


Figure 6.1: start.sh select assignment

step 3 When all modules are ready, in the *vision* one you have the possibility to re-detect images using both **normal** and **ROI**.
If the is object correctly recognized, pressing **ENTER** the arm will move to the piece of lego and will start the task assigned.

Figure 6.2: Run vision - step 4

4 Once all the tasks are finished, press **n** to kill all ROS nodes and quit all the modules correctly. (This step is necessary since Gazebo looks sensitive on closing) If you need to re-start some modules, you have to press **Y** in the last question, so you will be able to select the module that will be re-called. Notice that this section is recommended only in case of errors or bugs. The best way to act is to re-start the entire project from **step 1**.

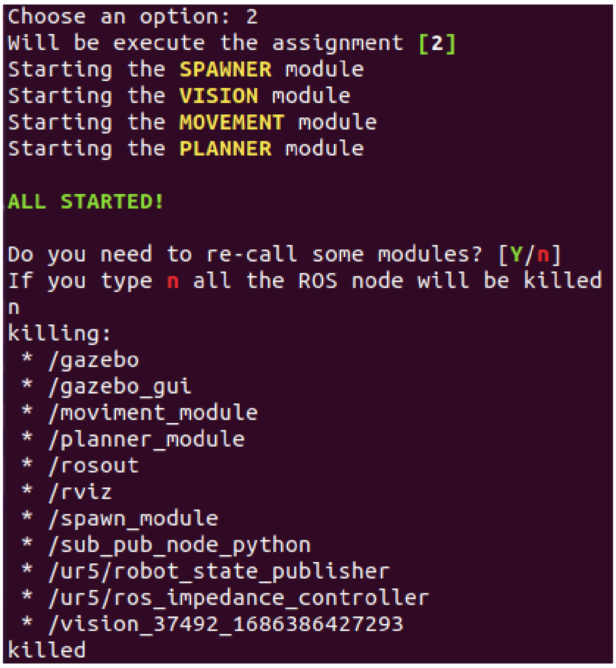


Figure 6.3: start.sh quit - step 4

7 Conclusions

In this project we faced a lot of challenges. We needed to have good knowledge in many disciplines such as mathematics, geometry, computer science, and programming and problem solving... But hard challenge doesn't freak the brave.

"Do or do not... there's no try"

Yoda.