POLITECNICO DI TORINO

Master Degree in Mechatronic Engineer

Robotics

Prof.:	Alessandro Rizzo
TAs:	David Pangcheng Cen Cheng
	Andrea Usai

I'm not a Robotic Arm

Muratore Luigi - s333098
Gennero Giorgia - s333099
Akbarov Iskandar s329650
Swaidan Moussa - s334402
Muhammad Fatir Noshab - s331898

2023/2024

Contents

Chapter 1

Introduction

Our idea is based on a ***5 degrees of freedom anthropomorphic robot manipulator***.
We drew inspiration from different online projects already built.
In this report, we will analyze the whole process we followed to complete the project, and we will discuss all the problems we ran into during our work and all the solutions we adopted to solve them.
Since we did the design part before the main concept of the robotics course we mistakenly focused on the aesthetics part more than the technical one, so we chose 5 DOF even if, during the course, we figured out that the best solution would have been 6 DOF.
This is because a robotic arm with 5 degrees of freedom has some limitations, such as:

- *Limited dexterity:*
  May struggle with tasks that require complex manipulation or reaching objects from all angles.

- *Less versatile:*
  May not be suitable for a wide range of tasks compared to a 6 DOF arm.

A secondary problem was that the wrist we built was not spherical.
Due to these problems, we could use neither some procedures and some approximations treated during the lectures nor the analytic solution of Inverse Kinematics but the numerical one.
At the end, we still obtained great results, and we are very pleased with our project.

Chapter 2

CAD

Using modeling and CAD programs we moved the project to a 3D space, we chose SolidWorks and Fusion360 to design the robot and to set the requirements.

We chose SolidWorks because it can include a particular plug-in that allows to export the project in URDF, a really helpful file that converts shapes and requirements into a sort of code that includes the properties and characteristics of the robot.
We had some problems with it, but we will see later more details.
We used Fusion360 instead, because there was the possibility to create some animations directly inside the program.
Thanks to that, we did some loop movements in a very easy way to better visualize our idea.

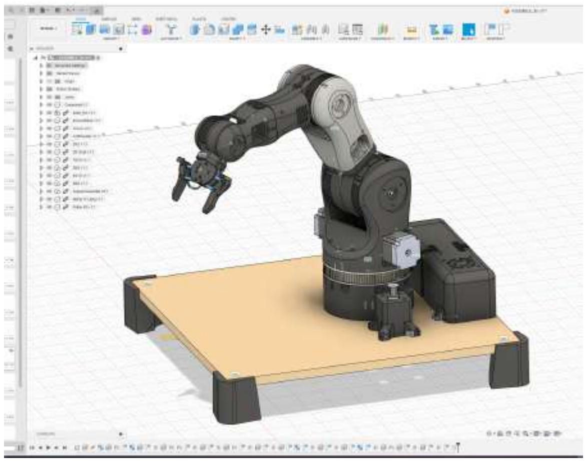


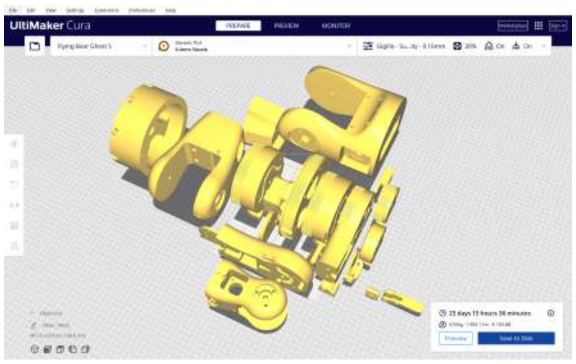
Figure 2.1: Modelling on Fusion360

Chapter 3

Bill of materials

The robot has been totally 3D printed with a Flybear ghost5 3d printer using PLA filament and built at home.
In order to complete the 3d printing part, including problems and faulty print that needed to be printed again, we used just over 4 kg of plastic, so 5 rolls of PLA, and it took about 250 hours overall.

(a) Flybear ghost5 3d printer



(b) Analysis in Cura slicer

3.1 Motors

We mainly used stepper motors for all the movements except for the end effector where we used a servo motor. In particular, we assembled:

- 1 Nema 14 for the first joint
- 2 Nema 23 for the second joint
- 1 Nema 17 with an internal reduction of 1:5 for the third joint
- 1 Nema 17 for the fourth joint
- 1 Nema 17 for the fifth joint
- 1 Servo of 20 kg for the gripper

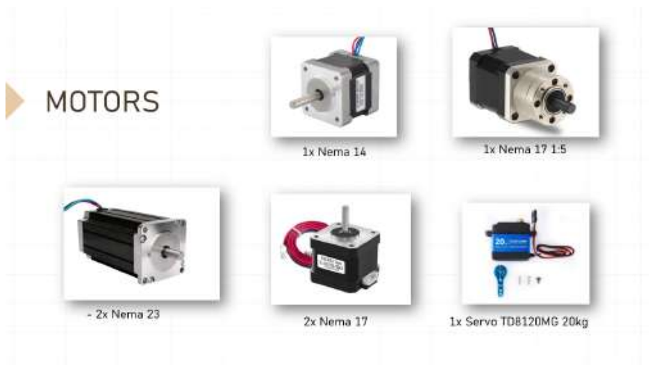


Figure 3.2: Motor used in the project

3.2 Drivers

All the motors are controlled in power by six tb6560 drivers.
The driver is necessary because the stepper motor operates by accurately synchronizing with the pulse signal output from the controller to the driver itself. Using switches mounted on the driver board, we could set all the parameters specific to each motor, such as the nominal current, excitation current, and step ratio.

Figure 3.3: Tb6560 Driver

3.3 Power supply

The whole system is supplied by a 24V 13A power supply.
All the drivers require 24 volts, so they are all connected in parallel to the output of the power supply. The servo, instead, needs about 6 volts, and it is connected to a smaller power supply in a cascade of the main one.
The boards have a separate circuit, just to be sure to not supercharge them.

Figure 3.4: Power supply

Chapter 4

Kinematics

During the course, we studied direct and inverse kinematics to analyze all the main motion aspects of a robot.
In robotics, kinematics is the branch of study concerned with the relationship between the **geometry** of a robot's structure (links and joints) and the **motion** of its end effector (the gripper or tool at the tip).
Kinematics doesn't take into account the forces or torques that cause the movement but focuses purely on the geometric relationships.
The key aspects of robot kinematics are:

- *Degrees of Freedom (DOF):*
  This refers to the minimum number of independent coordinates required to specify the position and orientation of a robot's end effector.
  As "Robotics: Modelling, Planning and Control - Siciliano" reports:
  `The degrees of freedom should be properly distributed along the mechanical structure in order to have a sufficient number to execute a given task. In the most general case of a task consisting of arbitrarily positioning and orienting an object in three-dimensional (3D) space, six DOFs are required, three for positioning a point on the object and three for orienting the object with respect to a reference coordinate frame.`
  As already said in the *introduction* section, a manipulator with less than six DOFs cannot take any arbitrary position and orientation in space.
  This was our first mistake.

- *Coordinate Frames:*
  Different parts of a robot are referenced using coordinate frames, which define the origin and orientation for measuring positions and movements.

- *Forward Kinematics (FK)*

- *Inverse Kinematics (IK)*

Kinematics is fundamental for:

- *Robot Control:*
  Kinematic calculations are used to generate control signals for the robot's motors, allowing it to move to specific points or follow desired paths.

  In our specific case, we used a **decentralized control system**, in particular an **independent joint control** architecture.
  This approach was chosen due to its simplicity in implementation and scalability for potential future modifications to the arm's design.
  In this architecture, each of the 5DOF stepper motors is controlled by a dedicated driver and controller. The control system transmits individual commands to each joint controller specifying the desired displacement, by means of step value.
  Additionally, to keep the project cost-effective, we opted for a simpler control system **without encoders**.
  The implementation of encoders in the future could enhance the control system's precision by providing real-time feedback on the actual joint positions, potentially enabling closed-loop control for improved accuracy.

- *Motion Planning:*
  Kinematic analysis helps plan trajectories for the robot's end effector to avoid obstacles and reach target positions efficiently.

- *Robot Programming:*
  Kinematic relationships are used in robot programming languages to specify desired movements and control robot behavior.

The performance of the robot can be analyzed by means of parameters, as:

- *Workspace volume*:
  is the reachable area or volume within which a robot's end-effector can operate.
  In our case, the shape of the workspace is close to a truncated irregular pyramid, with a volume of about 0.19 m^3.

- *Accuracy*:
  that in brief is how closely a robot's actions match the intended outcome.
  We estimated an error between 1 mm and 3 mm.

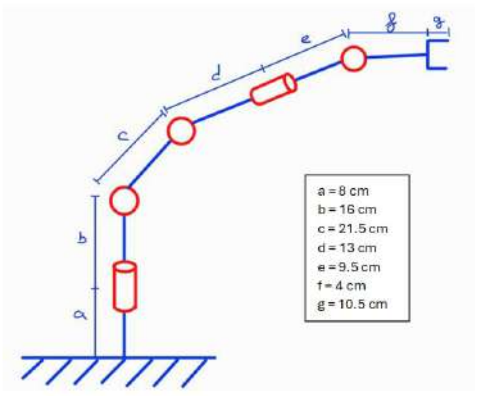
- *Repeatability*:
  that instead is how consistently a robot performs the same action over multiple attempts.
  In our case, is approximately between 0.5 mm and 1 mm.

4.1 Forward Kinematics

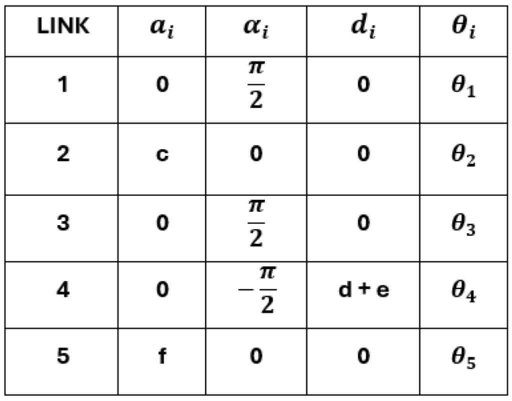
This problem involves calculating the position and orientation of the end effector given the joint angles (inputs) of the robot.
It's like solving a geometric puzzle to find the endpoint based on the joint rotations.

Where is and ***how*** is our end effector ***oriented***?

Firstly, we would to have a kinematics model of our manipulator, and we started studying the direct kinematics problem with the **Denavit–Hartenberg convention**.
Thanks to this convention, we could describe each frame considering the previous one using **4 parameters**.
We schematized the manipulator in an open chain with 5 revolute joints, 6 links, and the gripper as an end effector, as it is shown in the picture below with DH parameters.



(a) Scheme of the manipulator



LINK	a_i	α_i	d_i	θ_i
1	0	$\frac{\pi}{2}$	0	θ_1
2	c	0	0	θ_2
3	0	$\frac{\pi}{2}$	0	θ_3
4	0	$-\frac{\pi}{2}$	d + e	θ_4
5	f	0	0	θ_5

(b) DH parameters

Then we performed the Denavit-Hardenberg analysis to place the frames and have a kinematics model of the robot.

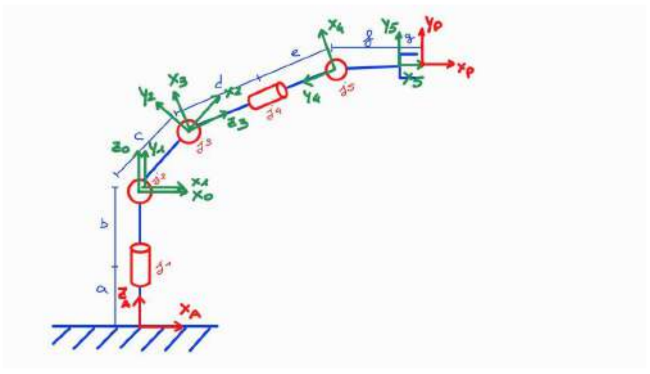
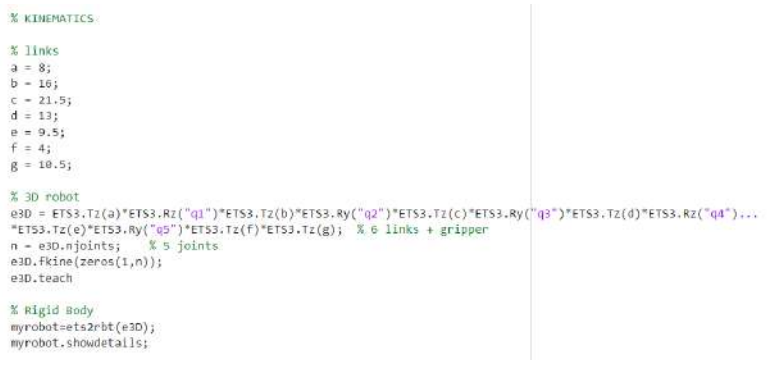


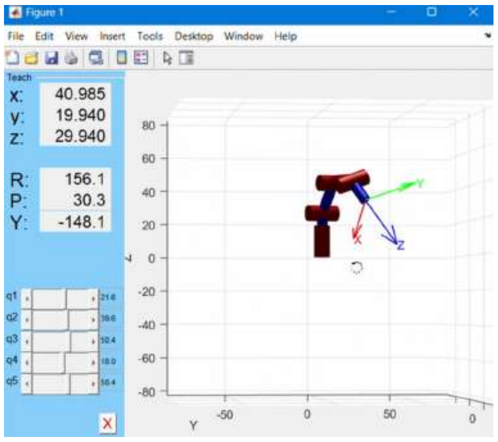
Figure 4.2: Robot's DH representation

We studied Forward Kinematics on MATLAB.



```
% KINEMATICS

% links
a = 8;
b = 16;
c = 21.5;
d = 13;
e = 9.5;
f = 4;
g = 10.5;

% 3D robot
e3D = ETS3.Tz(a)*ETS3.Rz("q1")*ETS3.Tz(b)*ETS3.Ry("q2")*ETS3.Tz(c)*ETS3.Ry("q3")*ETS3.Tz(d)*ETS3.Rz("q4")...
*ETS3.Tz(e)*ETS3.Ry("q5")*ETS3.Tz(f)*ETS3.Tz(g);  % 6 links + gripper
n = e3D.njoints;    % 5 joints
e3D.fkine(zeros(1,n));
e3D.teach

% Rigid Body
myrobot=ets2rbt(e3D);
myrobot.showdetails;
```

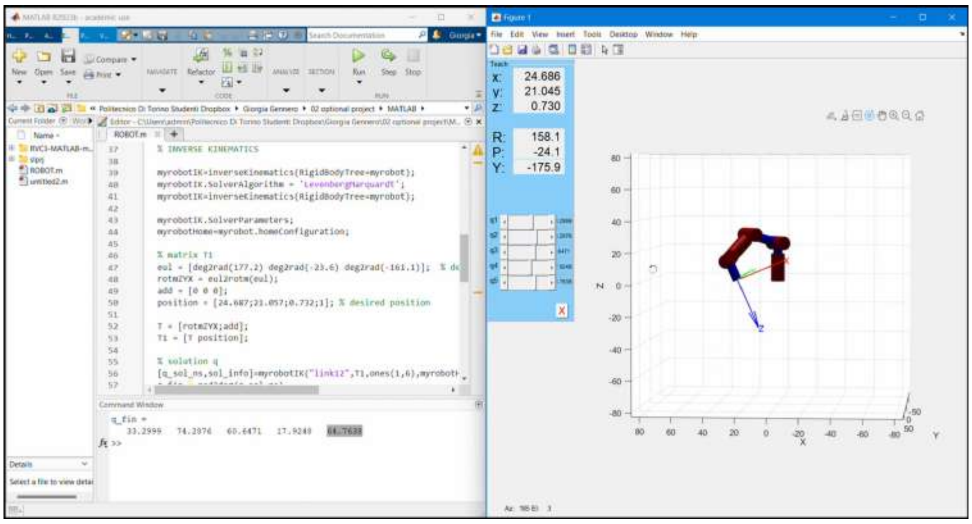
(a) MATLAB script



(b) Robot model in MATLAB

4.2 Inverse Kinematics

Given a desired position and orientation for the end effector (goal), we needed to determine the required joint angles (inputs) to achieve that position.
In this case, it is like working backward to find the joint configurations that reach the target location.
We mainly studied the Inverse Kinematics by means of MATLAB.



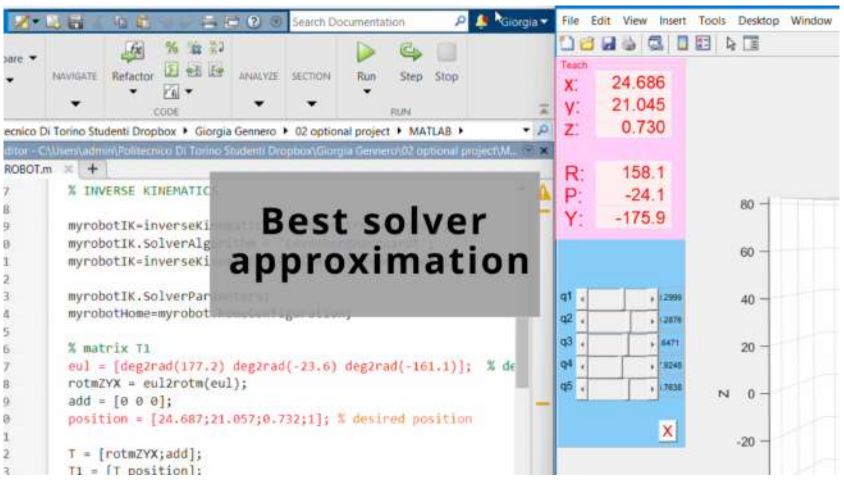
```
% INVERSE KINEMATICS

myrobotIK=inverseKinematics(RigidBodyTree=myrobot);
myrobotIK.SolverAlgorithm = 'LevenbergMarquardt';
myrobotIK=inverseKinematics(RigidBodyTree=myrobot);

myrobotIK.SolverParameters;
myrobotHome=myrobot.homeConfiguration;

% matrix T1
eul = [deg2rad(177.2) deg2rad(-23.6) deg2rad(-161.1)];  % de
rotmZYX = eul2rotm(eul);
add = [0 0 0];
position = [24.687;21.057;0.732;1]; % desired position

T = [rotmZYX;add];
T1 = [T position];

% solution q
[q_sol_ns,sol_info]=myrobotIK("link12",T1,ones(1,6),myrobot
```

```
q_fin =
    33.2999    74.2876    60.6471    17.9248    64.7633
fx >>
```

Figure 4.4: On the left is the MATLAB script and on the right is the result of the computation in terms of position and angles

As already mentioned in the *introduction* section, due to some technical and design problems, such as the wrist or the degree of freedom, we couldn't use the procedure and the solver treated

during the lectures, so we needed to do extra computations to reach our goals.
Here, after some computations, we found the best solver approximation with the minimum error between our desired angle position and the angle computed by the solver.

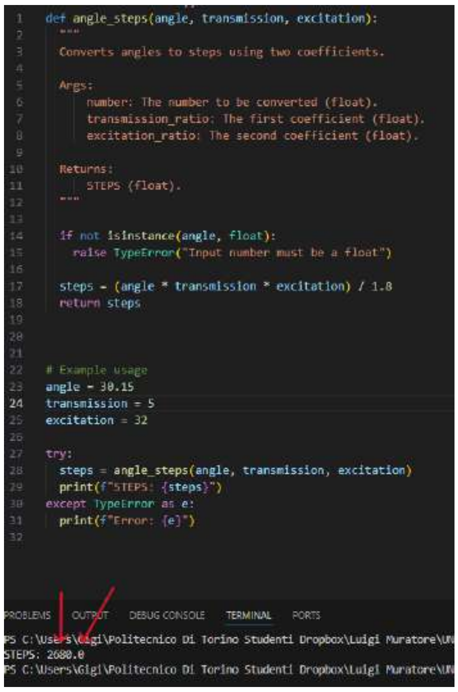

Search Documentation
Giorgia
File Edit View Insert Tools Desktop Window
NAVIGATE
Refactor
CODE
ANALYZE
SECTION
Run
Step
Stop
RUN
ecnico Di Torino Studenti Dropbox ▸ Giorgia Gennero ▸ 02 optional project ▸ MATLAB ▸
ROBOT.m
Best solver approximation
Teach
q1
q2
q3
q4
q5

```
% INVERSE KINEMATIC

myrobotIK=inverseKi
myrobotIK.SolverAlg
myrobotIK=inverseKi

myrobotIK.SolverPar
myrobotHome=myrobot

% matrix T1
eul = [deg2rad(177.2) deg2rad(-23.6) deg2rad(-161.1)];  % de
rotmZYX = eul2rotm(eul);
add = [0 0 0];
position = [24.687;21.057;0.732;1]; % desired position

T = [rotmZYX;add];
T1 = [T position];
```

```
X: 24.686
Y: 21.045
Z: 0.730

R: 158.1
P: -24.1
Y: -175.9
```

Figure 4.5: Comparison between position and angle desired and solver solution

As already said, once we found the correct angle of each joint, we needed to convert them into steps for the stepper motors by means of a simple script in Python.
It took into account the *transmission ratio* and *excitation ratio*.



```python
def angle_steps(angle, transmission, excitation):
  """

  Converts angles to steps using two coefficients.

  Args:
      number: The number to be converted (float).
      transmission_ratio: The first coefficient (float).
      excitation_ratio: The second coefficient (float).

  Returns:
      STEPS (float).
  """


  if not isinstance(angle, float):
    raise TypeError("Input number must be a float")

  steps = (angle * transmission * excitation) / 1.8
  return steps



# Example usage
angle = 30.15
transmission = 5
excitation = 32

try:
  steps = angle_steps(angle, transmission, excitation)
  print(f"STEPS: {steps}")
except TypeError as e:
  print(f"Error: {e}")
```

```
PS C:\Users\Gigi\Politecnico Di Torino Studenti Dropbox\Luigi Muratore\UN
STEPS: 2680.0
PS C:\Users\Gigi\Politecnico Di Torino Studenti Dropbox\Luigi Muratore\UN
```

Figure 4.6: Python script to convert angle in steps

In our case, the transmission ratio goes from *1* to *6.5*, while the excitation ratio is either *4* or *8* or *32*. The constant *1.8* is characteristic of the stepper motor, and it is essentially the angular distance the motor moves in a single step.

Chapter 5

Control

We designed different control algorithms.
We first based the control system on an *Arduino MEGA 2560*, such that allows us to control all the motors manually using 3 joysticks with 2 axes each.

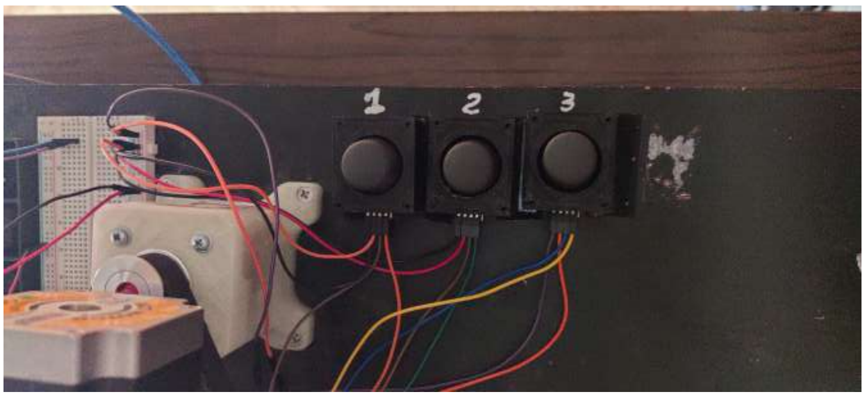


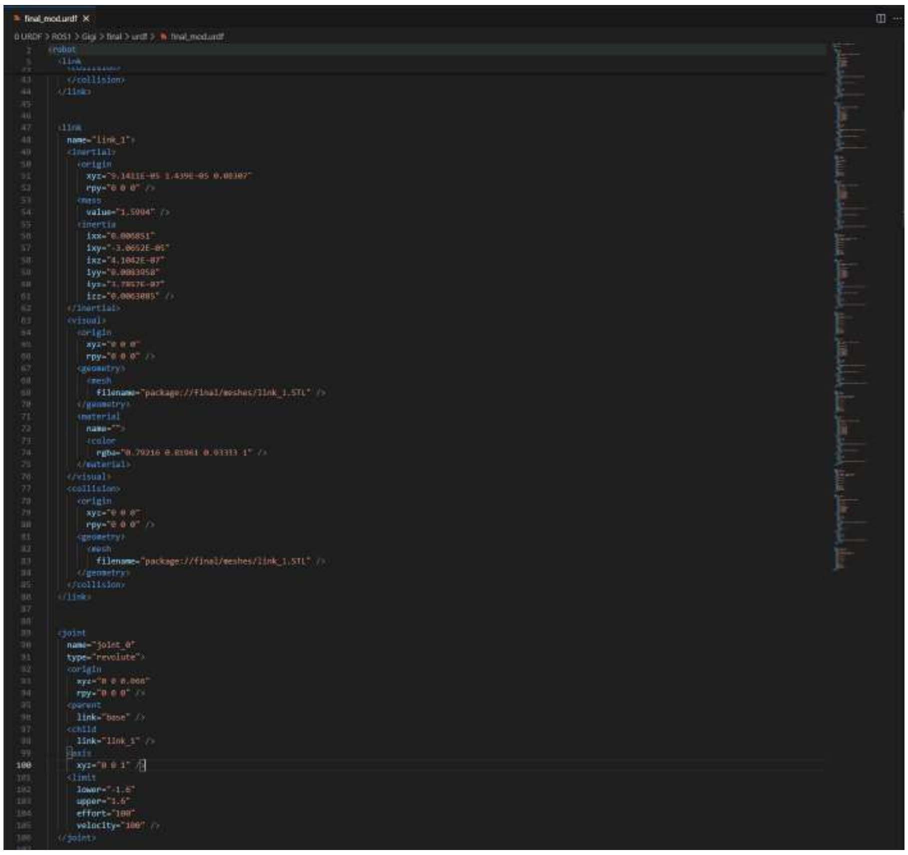
Figure 5.1: Structure ...

The Arduino Mega, however, has some limitations.
It is a single-task microcontroller, and even if we can achieve a kind of simulated multitasking through clever coding techniques, it still has one core and can only execute one instruction at a time.
So, to obtain faster computation and since in the future we would like to implement different sensors in the control system, we moved to a ***Raspberry Pi 4b*** that has enough power to run a *Linux* system with *ROS2*.
In particular, we used ***Linux Ubuntu 22.04*** with ***ROS2-Humble***.

5.1 URDF

To move the project we just created from a CAD program to whatever else program to study motion planning and Direct and Inverse Kinematics, there is a need for an external file that

describes the main characteristics of the robot's structure.
In this chapter we are going to talk about URDF file.
This is a specific file that describes all the properties of the robot regarding links and joints. In particular, it contains characteristics such as mass, origin, geometry, material, inertia, limits, and collisions.



```
final_mod.urdf
0 URDF > ROS1 > Gigi > final > urdf > final_mod.urdf
<robot
  <link
      </collision>
  </link>


  <link
    name="link_1">
    <inertial>
      <origin
        xyz="9.1411E-05 1.439E-05 0.00307"
        rpy="0 0 0" />
      <mass
        value="1.5904" />
      <inertia
        ixx="0.006851"
        ixy="-3.0652E-05"
        ixz="4.1042E-07"
        iyy="0.0083958"
        iyz="1.7957E-07"
        izz="0.0063085" />
    </inertial>
    <visual>
      <origin
        xyz="0 0 0"
        rpy="0 0 0" />
      <geometry>
        <mesh
          filename="package://final/meshes/link_1.STL" />
      </geometry>
      <material
        name="">
        <color
          rgba="0.79216 0.81961 0.03333 1" />
      </material>
    </visual>
    <collision>
      <origin
        xyz="0 0 0"
        rpy="0 0 0" />
      <geometry>
        <mesh
          filename="package://final/meshes/link_1.STL" />
      </geometry>
    </collision>
  </link>


  <joint
    name="joint_0"
    type="revolute">
    <origin
      xyz="0 0 0.068"
      rpy="0 0 0" />
    <parent
      link="base" />
    <child
      link="link_1" />
    <axis
      xyz="0 0 1" />
    <limit
      lower="-1.6"
      upper="1.6"
      effort="100"
      velocity="100" />
  </joint>
```

Figure 5.2: URDF file

5.2 Simscape Multibody Link

Thanks to the Simscape Multibody Link tool on SolidWorks, we could export the XML file.

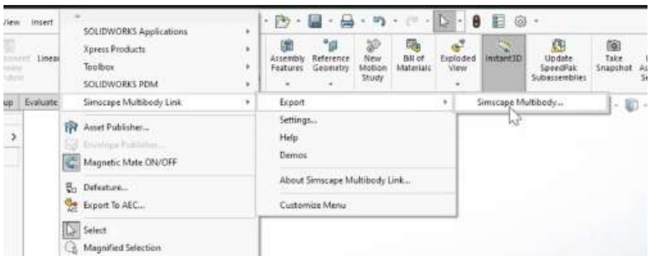


Figure 5.3: SolidWorks window to export CAD in Simscape Multibody model

This is another useful extension that allowed us to create the model on MATLAB/Simulink as well.
Once the XML file and all the files correlated with it have been exported, the Simscape Multibody model is generated with the function *smimport* on the MATLAB terminal:

```
smimport(”project_file_name”)
```

It will open a Simulink page with the model built.
Setting the parameters as limits, speed, and acceleration we could simulate our model checking that everything worked.

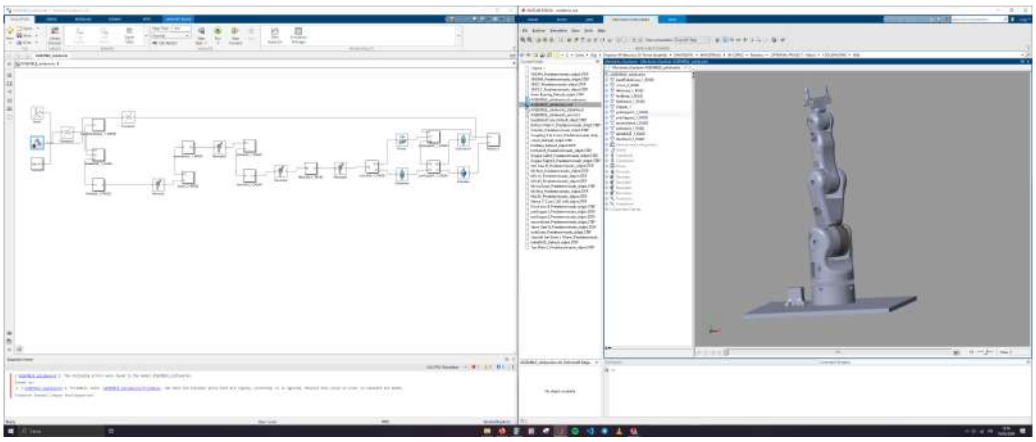


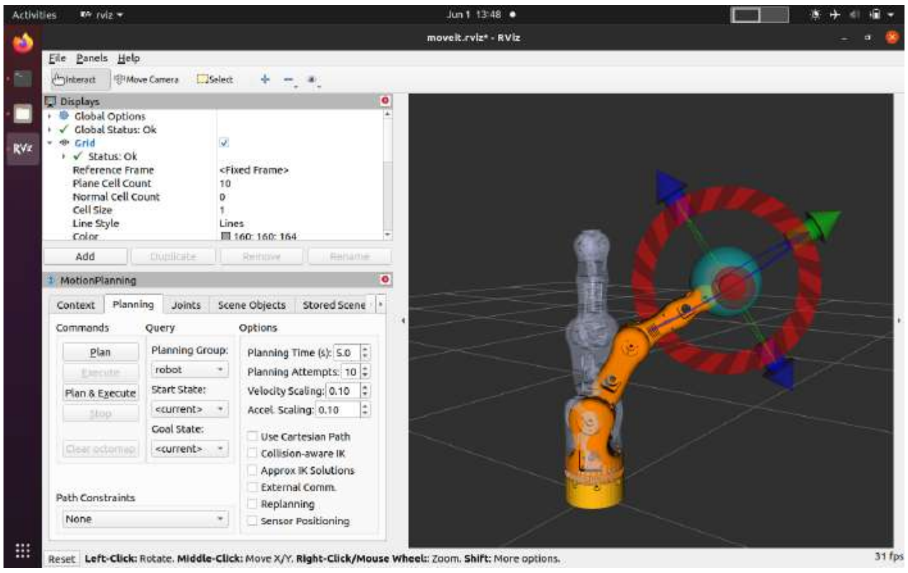
Figure 5.4: Simulink model of the robot. On the left is the structure of the model and on the right the model generated

5.3 MoveIt

With the URDF file, we could finally implement the model on MoveIt, a special tool that allows to plan trajectories and Inverse Kinematics with different types of solvers.
The programmers declare:

```
   "MoveIt is the most widely used software for manipulation, it
incorporates the latest advances in motion planning, manipulation,
3D perception, kinematics, control, and navigation.
MoveIt is state-of-the-art software for mobile manipulation."
```

Thanks to that, we computed our first test, trying to take a cube and move it to another place in the workspace.

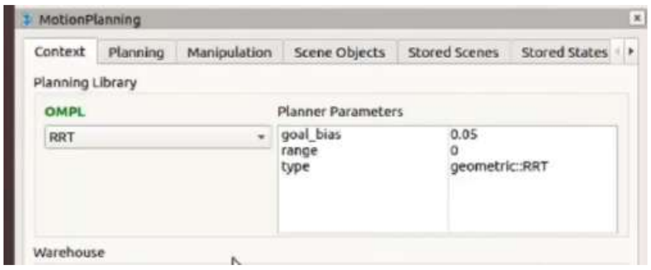


Global Status: Ok
Status: Ok
Reference Frame <Fixed Frame>
Plane Cell Count 10
Normal Cell Count 0
Cell Size 1
Line Style Lines
Color 160; 160; 164

Planning Group: robot
Start State: <current>
Goal State: <current>
Path Constraints: None
Planning Time (s): 5.0
Planning Attempts: 10
Velocity Scaling: 0.10
Accel. Scaling: 0.10

Figure 5.5: MoveIt window - planning section

We set the environment by putting the robot in its “zero position”, then we added a virtual cube in the workspace.
For all the computation, we used an already existing library of MoveIt called *OMPL - RRT* planning library.



Planning Library
OMPL: RRT

Planner Parameters

goal_bias	0.05
range	0
type	geometric::RRT

MotionPlanning
Context Planning Manipulation Scene Objects Stored Scenes Stored States
Warehouse

Figure 5.6: OMPL-RRT planning solver

This is one of the planning algorithms in MoveIt, in particular:

- ***OMPL (Open Motion Planning Library)***:
  this is a free and open-source software library that provides a collection of algorithms for solving motion planning problems.
  It doesn’t deal with specific robot kinematics or dynamics but focuses on the core task of finding a collision-free path for a robot to move from a starting configuration (*pose*) to a goal configuration while avoiding obstacles in the environment. It offers a variety of planning algorithms like *RRT, RRT*, PRM (Probabilistic Roadmap)*, and others. It provides a modular framework where different components, like collision checkers and visualization tools, could be integrated. It is freely available and widely used in robotics research and development.

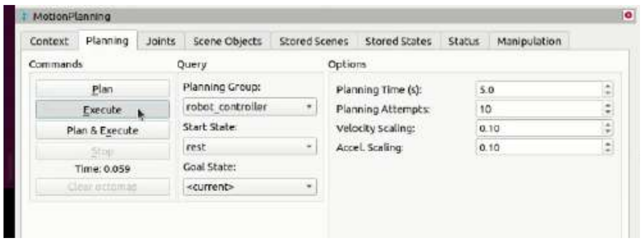
- ***RRT (Rapidly-exploring Random Tree)***:
  RRT is a sampling-based motion planning algorithm.
  It iteratively builds a tree-like structure in the robot's configuration space (all possible poses the robot can take).
  At each step:

  - It randomly samples a configuration in the space.
  - It finds the closest point in the tree to this random sample.
  - It extends the tree towards the random sample from the closest point, checking for collisions.

  If the extension reaches the goal configuration or reaches its maximum allowed length, the search stops. Otherwise, the new configuration is added to the tree.
  It is relatively simple to implement, efficient for high-dimensional spaces, and probabilistically complete. The drawback is that it might not find the shortest path, and can get stuck in local minima.

In essence, OMPL-RRT offers a powerful and versatile tool for solving motion planning problems because it combines the efficient exploration capabilities of the RRT algorithm with the flexibility and modularity of the OMPL framework.

After choosing the solver we started working on the interface between the robot and the cube, so we started planning the trajectory:

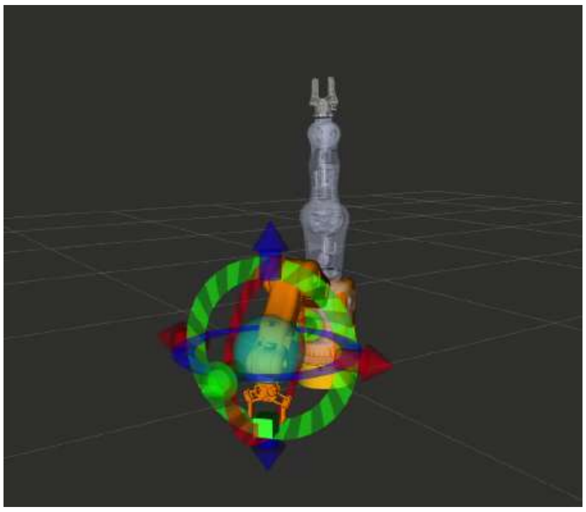


Time: 0.059

Planning Group: robot_controller
Start State: rest
Goal State: <current>

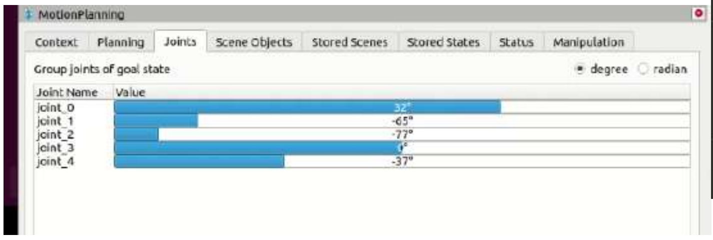
Planning Time (s): 5.0
Planning Attempts: 10
Velocity Scaling: 0.10
Accel. Scaling: 0.10

(a) Planning window on MoveIt



(b) Generation of the trajectory on the simulated world

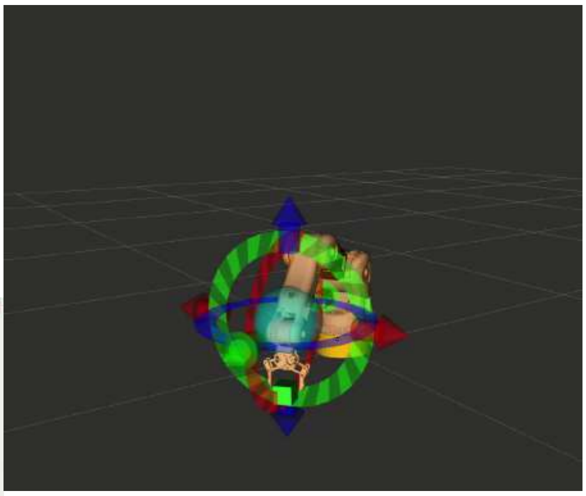
Once we did that, we executed the planning computed by the solver, and we obtained the Inverse Kinematics solution with all the angles related to each joint.



Group joints of goal state [✓ degree] radian

Joint Name	Value
joint_0	32°
joint_1	-65°
joint_2	-77°
joint_3	0°
joint_4	-37°

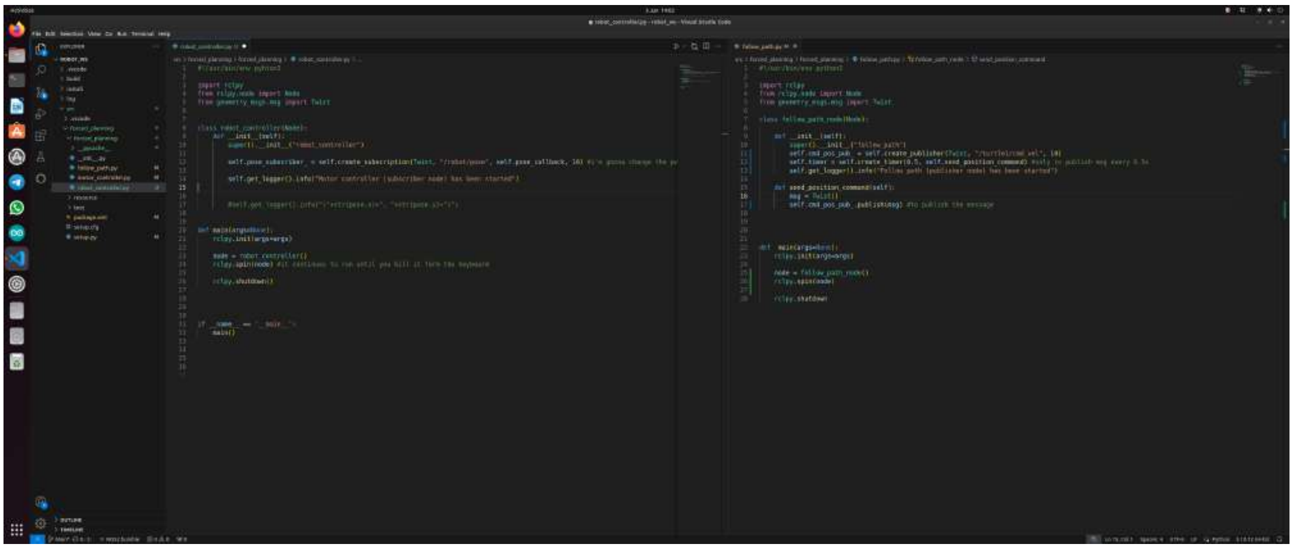
(a) Joint window on MoveIt


(b) Execution of the trajectory on the simulated world

Since we did not have a direct transmission between the motor's shaft and joint, we had to find all the transmission factors before putting the angles on the script, as we already saw in the *Inverse Kinematics* section.

5.4 ROS

Working with ROS, we had to create a sort of network to manage to communicate and exchange data. We created a package with a subscriber that takes our input data, such as angle positions or points in the workspace, and a publisher that sends them to another node that runs the script for the robot.



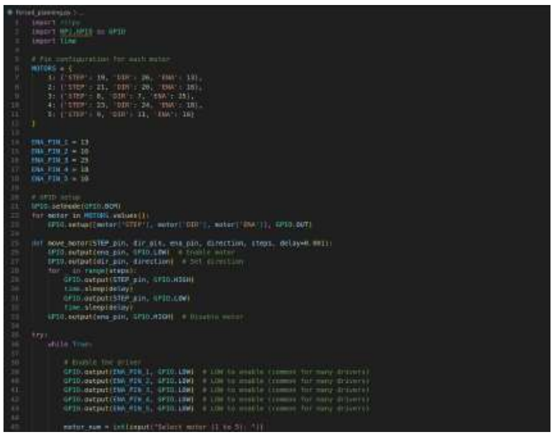
```
#!/usr/bin/env python3

import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist


class robot_controller(Node):
    def __init__(self):
        super().__init__("robot_controller")

        self.pose_subscriber_ = self.create_subscription(Twist, "/robot/pose", self.pose_callback, 10) #it's gonna change the p

        self.get_logger().info("Motor controller (subscriber node) has been started")

        #self.get_logger().info("("+str(pose.x)+", "+str(pose.y)+")")


def main(args=None):
    rclpy.init(args=args)

    node = robot_controller()
    rclpy.spin(node) #it continues to run until you kill it with the keyboard

    rclpy.shutdown()



if __name__ == '__main__':
    main()
```

```
#!/usr/bin/env python3

import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist

class follow_path_node(Node):

    def __init__(self):
        super().__init__("follow_path")
        self.cmd_pos_pub_ = self.create_publisher(Twist, "/turtle1/cmd_vel", 10)
        self.timer = self.create_timer(0.5, self.send_position_command) #only to publish msg every 0.5s
        self.get_logger().info("Follow path (publisher node) has been started")

    def send_position_command(self):
        msg = Twist()
        self.cmd_pos_pub_.publish(msg) #to publish the message



def  main(args=None):
    rclpy.init(args=args)

    node = follow_path_node()
    rclpy.spin(node)

    rclpy.shutdown
```

Figure 5.9: Defining ROS nodes in Python with VSC (Visual Studio Code)

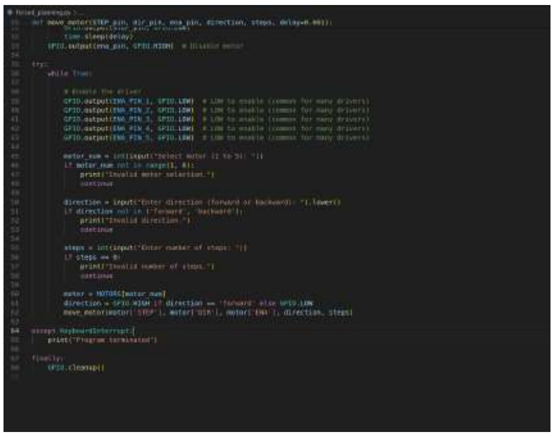
5.5 Scripts

We used Python for all the coding parts.
Before creating the ROS network, we tried some scripts directly with the robot, setting the pin of each motor and defining functions that moved the joints with predefined angles.



```python
import rospy
import RPi.GPIO as GPIO
import time

# Pin configuration for each motor
MOTORS = {
    1: {'STEP': 19, 'DIR': 26, 'ENA': 13},
    2: {'STEP': 21, 'DIR': 20, 'ENA': 16},
    3: {'STEP': 8, 'DIR': 7, 'ENA': 25},
    4: {'STEP': 23, 'DIR': 24, 'ENA': 18},
    5: {'STEP': 9, 'DIR': 11, 'ENA': 10}
}

ENA_PIN_1 = 13
ENA_PIN_2 = 16
ENA_PIN_3 = 25
ENA_PIN_4 = 18
ENA_PIN_5 = 10

# GPIO setup
GPIO.setmode(GPIO.BCM)
for motor in MOTORS.values():
    GPIO.setup([motor['STEP'], motor['DIR'], motor['ENA']], GPIO.OUT)

def move_motor(STEP_pin, dir_pin, ena_pin, direction, steps, delay=0.001):
    GPIO.output(ena_pin, GPIO.LOW)  # Enable motor
    GPIO.output(dir_pin, direction)  # Set direction
    for _ in range(steps):
        GPIO.output(STEP_pin, GPIO.HIGH)
        time.sleep(delay)
        GPIO.output(STEP_pin, GPIO.LOW)
        time.sleep(delay)
    GPIO.output(ena_pin, GPIO.HIGH)  # Disable motor

try:
    while True:

        # Enable the driver
        GPIO.output(ENA_PIN_1, GPIO.LOW)  # LOW to enable (common for many drivers)
        GPIO.output(ENA_PIN_2, GPIO.LOW)  # LOW to enable (common for many drivers)
        GPIO.output(ENA_PIN_3, GPIO.LOW)  # LOW to enable (common for many drivers)
        GPIO.output(ENA_PIN_4, GPIO.LOW)  # LOW to enable (common for many drivers)
        GPIO.output(ENA_PIN_5, GPIO.LOW)  # LOW to enable (common for many drivers)

        motor_num = int(input("Select motor (1 to 5): "))
```

(a) Defining the pin motors and the libraries



```python
def move_motor(STEP_pin, dir_pin, ena_pin, direction, steps, delay=0.001):
        time.sleep(delay)
    GPIO.output(ena_pin, GPIO.HIGH)  # Disable motor

try:
    while True:

        # Enable the driver
        GPIO.output(ENA_PIN_1, GPIO.LOW)  # LOW to enable (common for many drivers)
        GPIO.output(ENA_PIN_2, GPIO.LOW)  # LOW to enable (common for many drivers)
        GPIO.output(ENA_PIN_3, GPIO.LOW)  # LOW to enable (common for many drivers)
        GPIO.output(ENA_PIN_4, GPIO.LOW)  # LOW to enable (common for many drivers)
        GPIO.output(ENA_PIN_5, GPIO.LOW)  # LOW to enable (common for many drivers)

        motor_num = int(input("Select motor (1 to 5): "))
        if motor_num not in range(1, 6):
            print("Invalid motor selection.")
            continue

        direction = input("Enter direction (forward or backward): ").lower()
        if direction not in ['forward', 'backward']:
            print("Invalid direction.")
            continue

        steps = int(input("Enter number of steps: "))
        if steps <= 0:
            print("Invalid number of steps.")
            continue

        motor = MOTORS[motor_num]
        direction = GPIO.HIGH if direction == 'forward' else GPIO.LOW
        move_motor(motor['STEP'], motor['DIR'], motor['ENA'], direction, steps)

except KeyboardInterrupt:
    print("Program terminated")

finally:
    GPIO.cleanup()
```

(b) Main part where all the functions are called

In the rightmost picture, there are the functions that allow choosing both which motor move and the angle of rotation.
While, in the first part of the leftmost picture, it shows the pin definition part, where we put three variables for each motor.
In particular, to properly set a stepper motor, there must be three parameters:

- *Direction*:
  Controls the rotational direction of the stepper motor. A common logic level (usually LOW) sets the motor to rotate in one direction (e.g., clockwise). The opposite logic level (usually HIGH) sets it to rotate in the other direction (e.g., counterclockwise).

- *Step*:
  Triggers the motor to take one incremental step (rotation by a specific angle). Each pulse sent to this pin advances the motor by a defined angular increment (called a step angle), which depends on the motor's specifications and the driver's micro-stepping configuration. The frequency of pulses on this pin determines the motor's speed. Higher pulse rates lead to faster rotation. Microstepping allows for finer control of the motor's movement by dividing each full step into smaller sub-steps.

- *Enable*:
  Controls whether the driver outputs power to the motor windings. When the Enable pin is set to the active logic level (often LOW), the driver is enabled, and the motor can rotate when Step pulses are received. When the Enable pin is set to the inactive logic level (often HIGH), the driver is disabled, and the motor windings are not energized, even if Step pulses are sent.

This can be used to:
- Stop the motor quickly (by holding the Step pin low while disabling)
- Reduce power consumption when the motor is not in use
- Implement safety features by disabling the motor in certain conditions

Here's a table summarizing the key points:

Parameter	Function	Active Logic Level	Inactive Logic Level
Direction	Sets rotational direction	LOW	HIGH
Step	Triggers a motor step	Pulse	No Pulse
Enable	Enables/disables power to motor windings	LOW	HIGH

.

By controlling these three parameters, we could achieve precise movements.

Chapter 6

Tests

6.1 Pick and Place

The goal of this first test was to use the robot for *Pick and Place* operations.
Specifically, we put a cube in the workspace and, as already discussed in the *MoveIt* section, we computed the trajectories to move it to a second position using Inverse Kinematics approach and MoveIt solver.

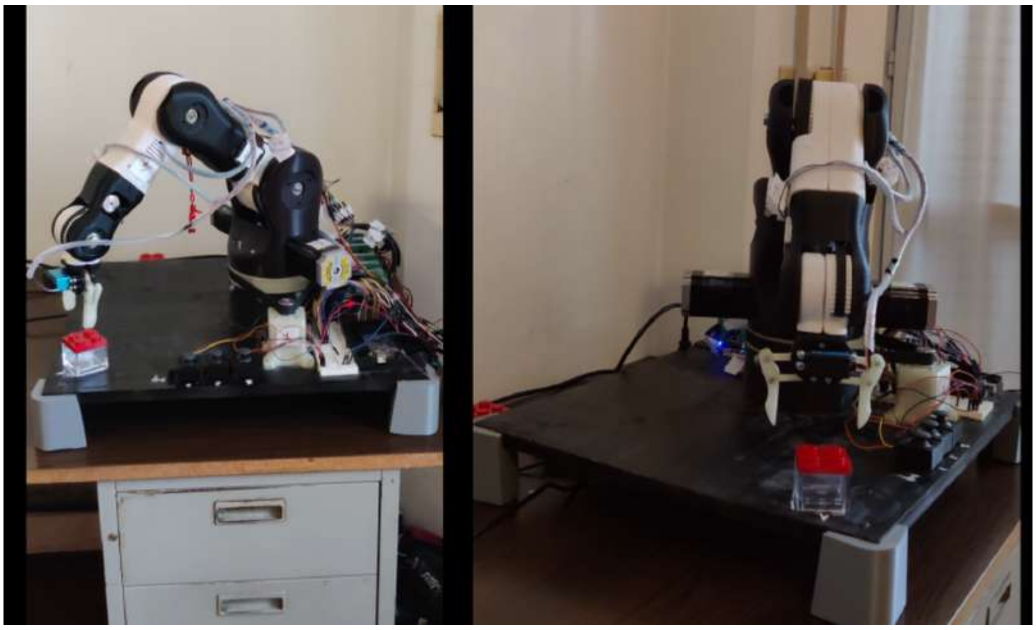


Figure 6.1: First position of "Pick and Place" test

In this first position we had an accuracy error of 2 mm, we suspect mainly due to the pulley of the first motor that skidded a little bit.
All the other motors attained the position with an error of less than 1 mm.

Second position:

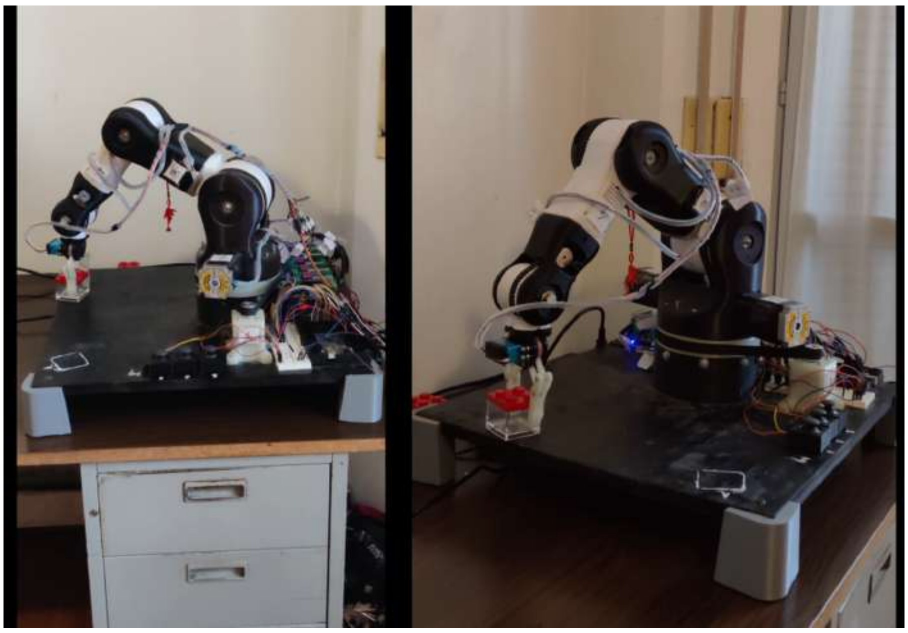


Figure 6.2: Second position of “Pick and Place” test

Also in this case we had some problems regarding the accuracy due to mechanical assembling issues, but we solved them. In the end the robot reached the correct position to move the cube and, playing with Python code, we were able to move the whole system very smoothly.

6.2 I’m not a Robotic arm

I’m not a Robotic arm was more like an entertaining challenge than a learning experience because it needed technical movements that produced errors in the Inverse Kinematic solver due to the fact that we had one degree of freedom less than the solver required.
After some attempts and some approximation, we reached the goal.

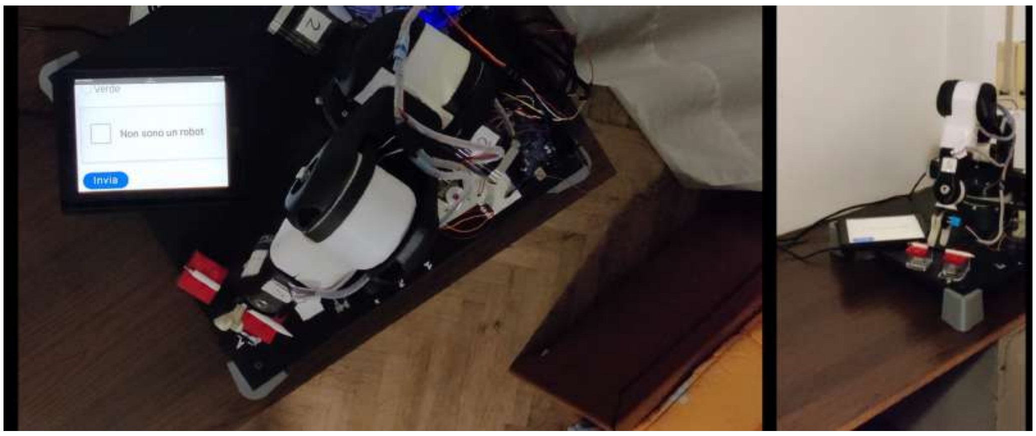


Figure 6.3: First position of “I’m not a robotic arm” test

Second position

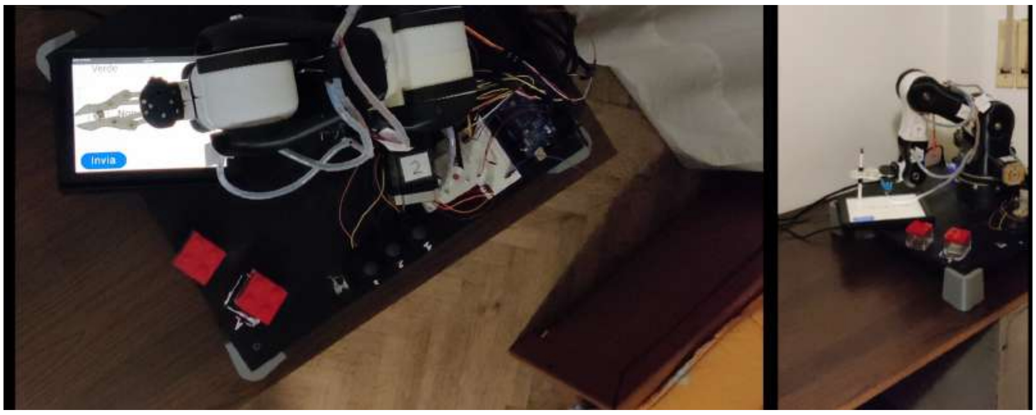


Figure 6.4: Second position of “I’m not a robotic arm” test

In this case, the Python scripts produced a lot of errors, we are still checking why, but we could not do a single transition movement from the first position to the second one, so it took three more, and it seems not very smooth.
Other than that, the Inverse Kinematic solution was correct, and all the positions were reached successfully.

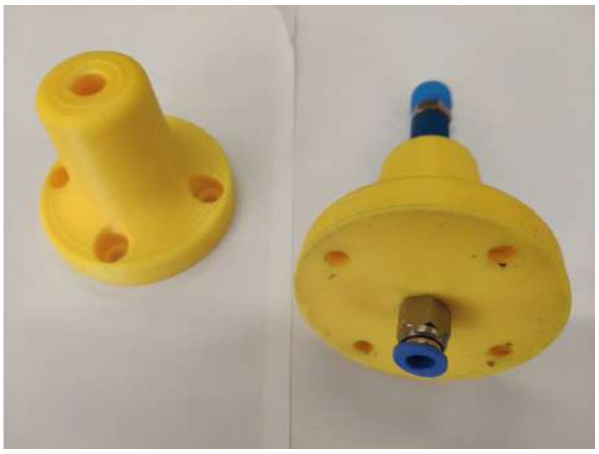
Chapter 7

Future upgrades

Looking at the future, we have a lot of upgrades or implementations in mind.
Actually, we are already working on some of them, while about the last we need more study.

7.1 Vacuum gripper

One of the upgrades we are working on is a new end effector solution.
We would like to substitute the classic gripper, the "two fingers" one, with a new concept using a vacuum system and suction cups to hold objects.
On the left there is a design we drew that we are currently testing and on the right there is the company from where we drew inspiration that produces end effectors with this technology:

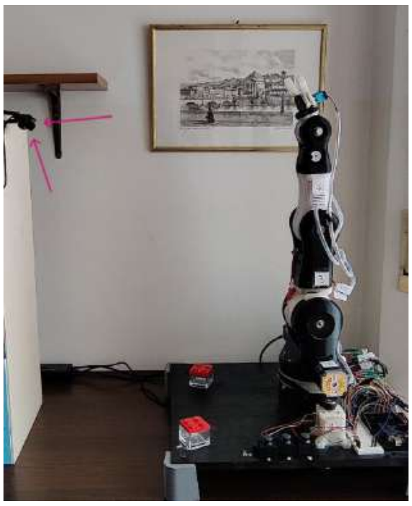


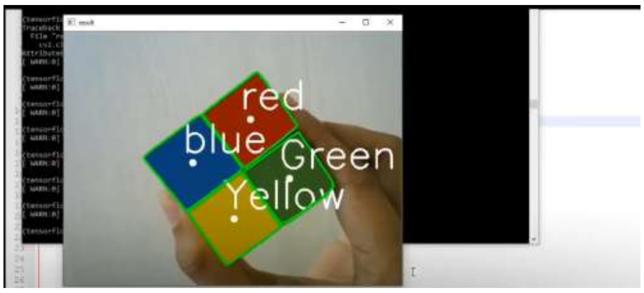
We are studying the effort needed to efficiently hold an object so the power necessary to be generated by a pump.

7.2 Stereo camera

The second upgrade, almost ready to be implemented, is a stereo camera.
It is a type of camera with two or more lenses that allows to simulate human binocular vision,

and therefore gives it the ability to capture three-dimensional images.
Our goal is to use the stereo cameras for making stereo-views and 3D pictures, so process them with algorithms of object recognition and object classification. In this optic, we could put objects wherever in the workspace and let the robot process the data and decide what to do according to their colors and their shapes.
So far we put a simple camera in front of the robot, outside the workspace, that is able to recognize the color of the object placed in the workspace and some simple shapes. We are working with OpenCV on Python, in particular with algorithms of edge detection for the shape and HSV colorspace for color detection.





Since the camera is not a stereo-camera, it is not able to detect the exact position of the object and so gives the robot the information to compute the inverse kinematics.
However, it is possible to put an object in a position that the robot already knows and perform the recognition algorithms.
We are aiming to take a stereo camera or create it using two cameras and calibrating them, and configure it directly on the top of the robot.

7.3 Artificial Intelligence

A very interesting implementation could be the *Artificial Intelligence*.
Nowadays, it is very easy to play with different types of AI so why don't do something useful for our projects.

7.3.1 Voice control

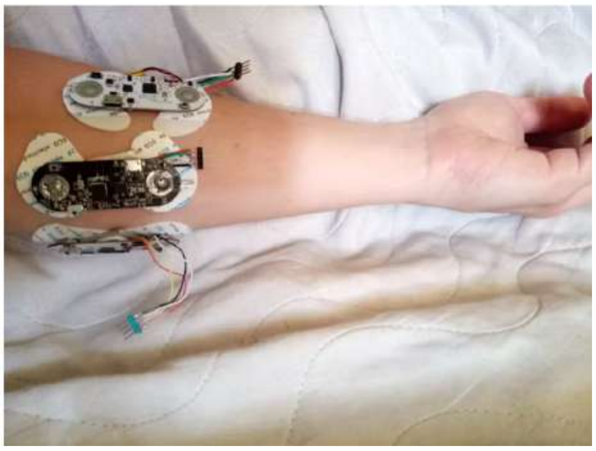
A very simple step could be to add a voice control to the system, so something that recognizes the voice and, using predefined words, could perform some tasks associated to the word itself.

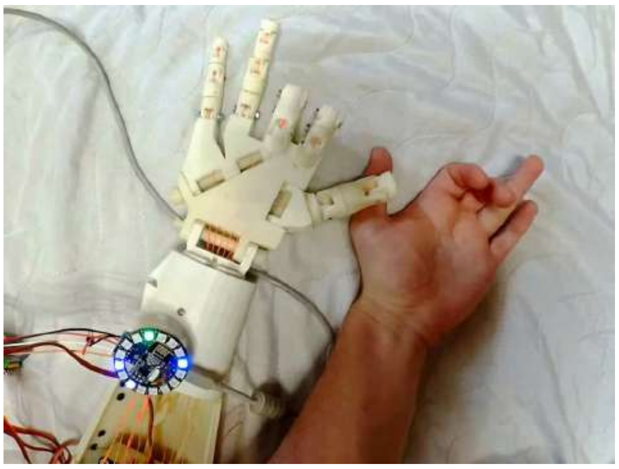
7.3.2 Code generator

The AI may also be used to generate code, and maybe train the algorithms to improve the accuracy at every trajectory done.
The last two could be merged, so the last step would be to combine *code generator* with *voice control*, to ask for a specific task, that may not yet be defined, and let the **AI** work on it to generate the code for the robot.

7.4 EMG control

The last upgrade we tough is to combine this project with another project we are working on, which is based on Electromyography (EMG).
This is, actually, a diagnostic procedure to assess the health of muscles and the nerve cells that control them (motor neurons), but we use it to read signals from the muscles and process them to perform predefined tasks.
Electromyography, in fact, is exactly the technique for evaluating and recording the electrical activity produced by skeletal muscles.





It could be very interesting to try to combine these two projects to achieve a remote control of the robot arm using electrodes placed on a human body.