POLITECNICO DI BARI

DEI DEPARTMENT

MASTER'S DEGREE IN AUTOMATION ENGINEERING

Robotics: Industrial Handling

Prof. Dr. Eng. Lino Paolo

Project:

'Dracarys 5J' - 5DOF Robotic Arm

Students:
Francesco Stasi
Davide Tonti

Academic Year 2024-2025

Contents

1 Introduction

This project is dedicated to the development and implementation of a five-degree-of-freedom anthropomorphic manipulator, whose kinematic structure is depicted in Figure 1. The primary objective is to design and construct a modular robotic arm that is entirely 3D-printable and released as open-source, thereby fostering system reproducibility and accessibility within the robotics community.

The robotic system is meticulously designed using advanced 3D modeling software. Subsequently, the individual components are fabricated through additive manufacturing techniques, specifically utilizing 3D printing technology. The actuation system employs motors that operate under an open-loop control scheme, ensuring a balance between system simplicity and functional efficiency.

A key distinction in motor selection is the implementation of stepper motors, which feature rotors capable of executing predefined rotational increments without the loss of step precision. In contrast, servo motors are equipped with an internal control logic that facilitates high-precision movements, thereby enhancing the accuracy and reliability of the manipulator.

The operational hardware architecture of the robotic system comprises dedicated motor drivers and a control board, which collectively govern the motion of the manipulator. The overarching objective is to enable the end effector to follow predetermined motion trajectories characterized by a trapezoidal joint profile, ensuring smooth and efficient actuation.

The final phase of the project will be devoted to extensive testing and validation procedures to rigorously assess the manipulator's performance. These evaluations will determine whether the system adheres to the prescribed technical specifications and functions reliably across diverse operational scenarios.

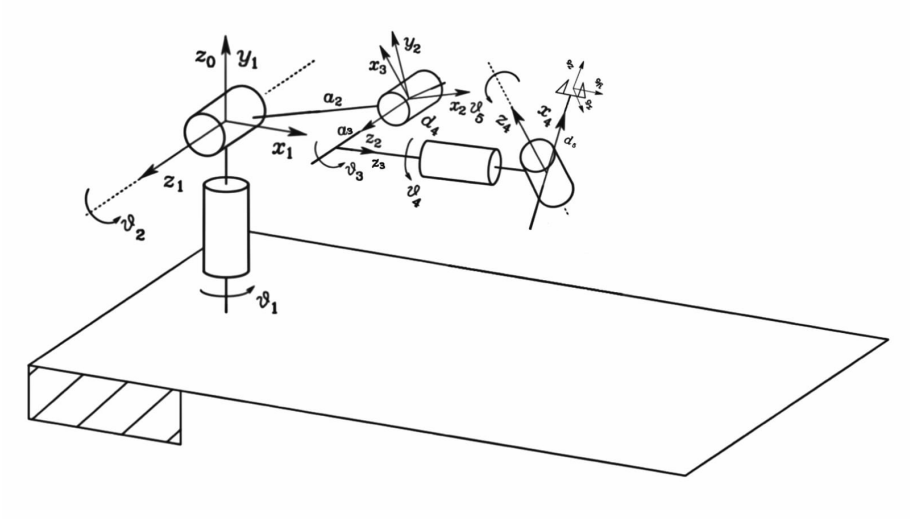


Figure 1: Robotic arm 5 DOF

2 Mechanical structure

2.1 3D model with Solidworks

Before the 3d model is made, the types of engines to be adopted are decided. The manipulator to be made has 5 degrees of freedom, having to opt for an initial open-loop control, stepper motors are chosen for the anthropomorphic part, and servo motors for the remaining part. For joint two, we think about redundancy, dedicating two motors to it. The steppers used will be NEMA 17, which are easy to control and source. The two servomotors for wrist control are MG996r.
It is possible to observe the motors in the assembly in fig.2a.

The 3d model was made using Solidworks software [1].
The final assembly that makes up the 3D model is divided into several categories: external parts, electromechanical components, tubular components, main parts, and secondary parts (fig.2c).
The external parts mainly include components taken from the SolidWorks toolbox, such as bolts, bearings, and gears (fig.2b).
The electromechanical parts include the stepper motors and servomotors, which are essential for the robot's operation. The main structure of the robot consists of the main parts, which need to be produced using 3D printing. Lastly, the secondary parts, while not necessary for the robot's assembly and operation, serve purely aesthetic purposes.

It starts with the realization of the base in which the first motor is anchored to move the first joint. This is accomplished by a ratio of sprockets that move the manipulator that is free to slide on a bearing. The bearing is made by creating a groove in which balls with a diameter of 8 mm each can slide.

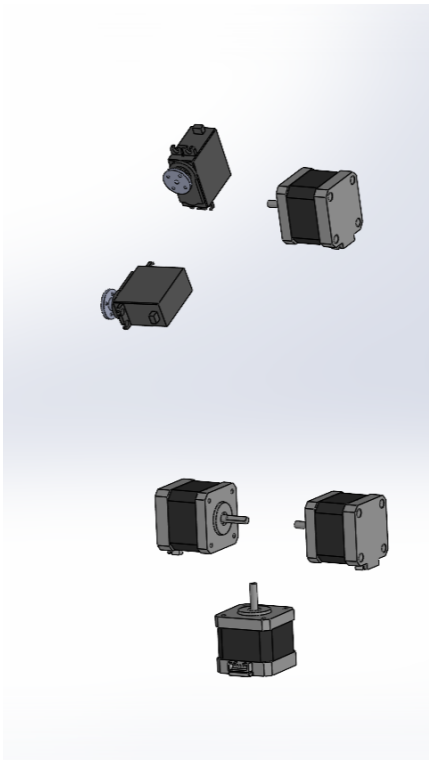
Stepper motors are used for each joint of the anthropomorphic part of the robot. Each stepper is coupled with straight-toothed gears in a pinion-crown configuration. Table 1 shows the transmission ratios, calculated as the ratio between the number of teeth on the crown and the number of teeth on the pinion. In our case, the transmission ratio will be useful for adjusting the speed and torque of the stepper motor.

Joint	Pinion Teeth	Gear Teeth	Transmission Ratio
1	15	64	$4.2\overline{6}$
2	9	54	6
3	9	44	$4.\overline{8}$

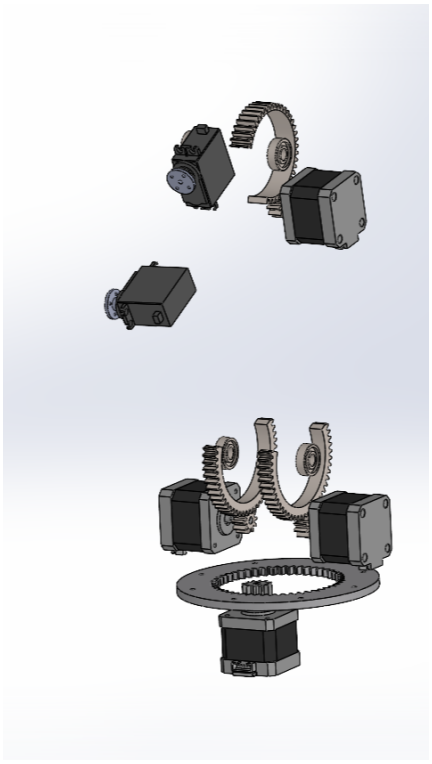
Table 1: Table with joints and transmission ratios

Joints 4 and 5 form the wrist of the manipulator and are driven by two servomotors. Unlike stepper motors, these are directly connected to the factory pinion, without intermediate transmissions, and make use of proprietary supports.

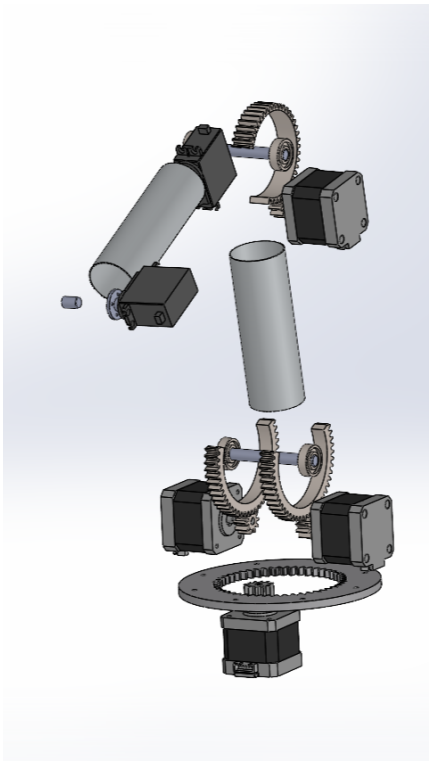
The manipulator body consists of all the parts that hold motors and mechanical parts together, and these are the 3D printable components (fig. 2d). The complete assembly includes the end effector, which, as a modular component consists of an imported assembly. The complete result, considering additional parts, is shown in fig 3.



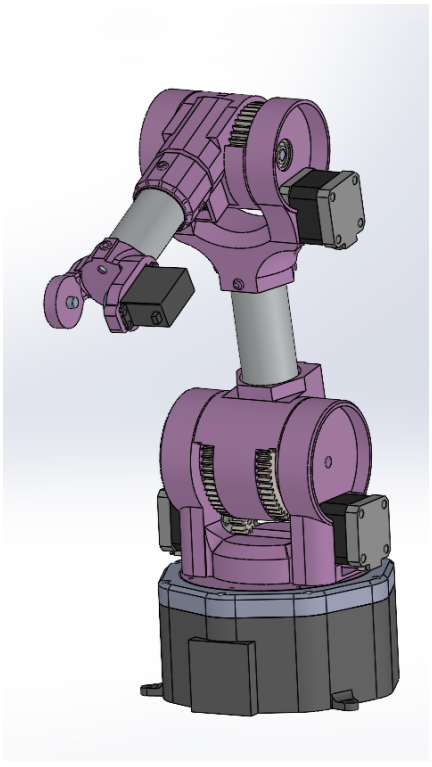
(a) Motor parts assembly



(b) Toolbox parts assembly



(c) Mechanical parts assembly



(d) Body parts assembly

Figure 2: Decomposed assembly

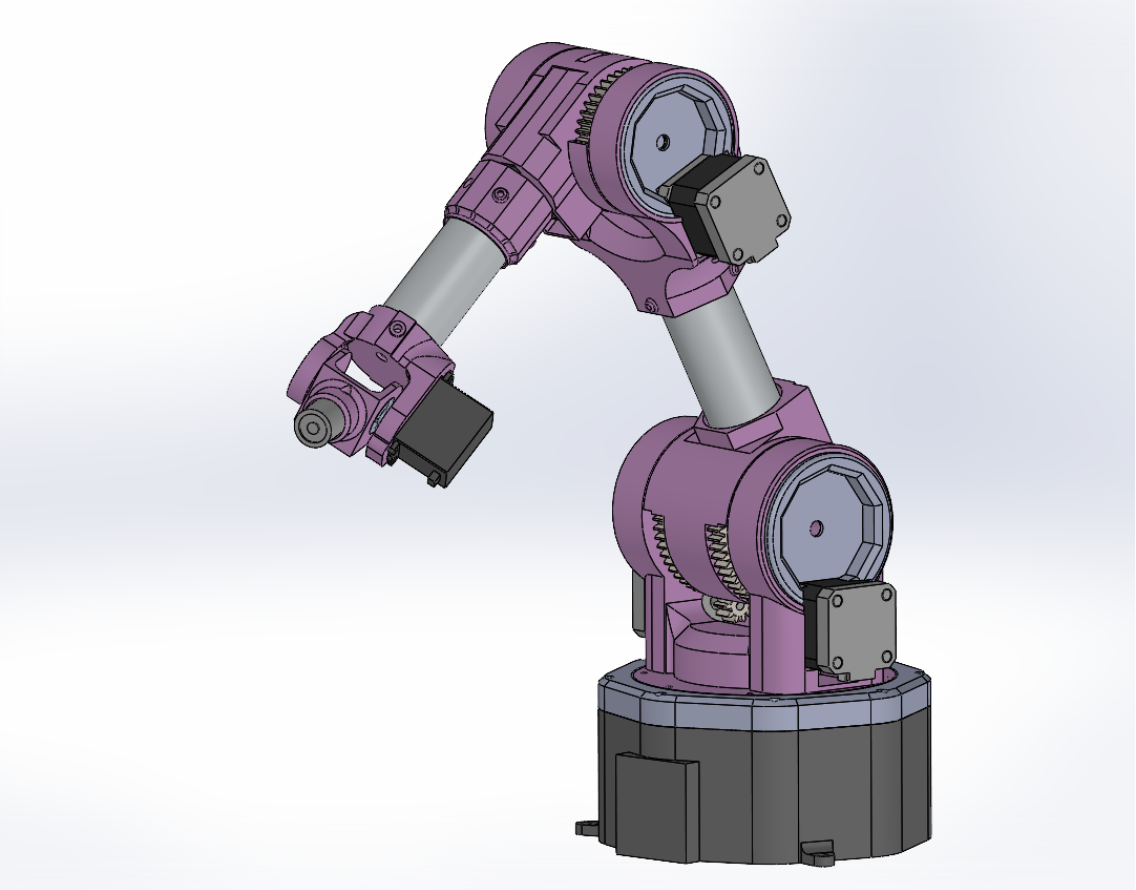


Figure 3: Assembly result

2.2 End effector

The initial design of the end effector features a casing housing a cylindrical electromagnetic component. This configuration serves as an effective foundational model due to its straightforward design and ease of control. To validate trajectory execution, an electromagnet capable of lifting a metallic sphere will suffice for preliminary testing.

The modularity of the structure allows for the integration of diverse end-effector types through appropriate modifications. For example, an adaptive parallel gripper, whose 3D model is available in a public repository [2], could be incorporated as an alternative.

All subsequent tests documented in this report will utilize the first end effector design for consistency.

2.3 Material selection and 3D printing

To ensure adequate robustness, the main components are printed in ABS, while the secondary parts use PLA. Certain elements, such as the gear wheels, are initially produced using an FDM printer with ABS.

However, this approach has some drawbacks: the noise generated by gear meshing is high, and precision is compromised. A potential solution would be to print these parts in resin, significantly improving accuracy. In this case, a more durable resin—specifically designed for mechanical components—would need to be used.

Figure 4: Component printing

3 Hardware

This chapter lists and briefly describes the hardware used for our scope.

3.1 STM32 NUCLEO-F446RE Overview

The STM32 NUCLEO-F446RE (Fig.5) is an advanced microcontroller of the STM32 family, built around an ARM Cortex-M4 32-bit core. Operating at frequencies of up to 180 MHz and equipped with a floating-point unit (FPU), it is designed to handle applications requiring high computational performance. With extended support for Digital Signal Processing (DSP) instructions, this microcontroller is ideal for tasks such as motor control, signal processing, and other advanced embedded applications.

It features 512 KB of flash memory for stable program storage and 128 KB of SRAM for dynamic data management during runtime. Its high-performance clock system ensures precision and efficiency even in demanding workloads. The configurable GPIO ports allow seamless interfacing with external devices such as sensors and actuators, making it highly versatile for various applications.

The STM32 NUCLEO-F446RE integrates a comprehensive set of peripherals, including 12-bit ADC and DAC for analog signal management, advanced timers for precise motor control and PWM generation, and communication interfaces such as I2C, SPI, USART, CAN and USB 2.0 Full-Speed. Additionally, the DMA (Direct Memory Access) controller enables direct data transfers between memory and peripherals, reducing CPU load and improving overall performance.

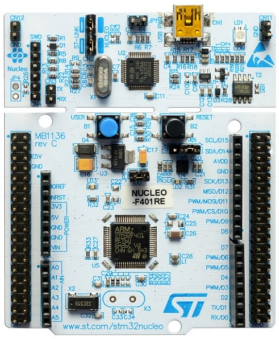


Figure 5: STM32 NUCLEO-F446RE

3.2 NEMA 17 stepper motor

The NEMA 17 is a widely used stepper motor, employed in applications such as 3D printers, CNC machines, robotics, and automation. The voltage we will use to power it, to achieve maximum performance, is 24V.
It typically provides a torque between 0.3 and 0.65Nm, with an angular step of 1.8° (200 steps per revolution).
It can have 4 or 6 wires, depending on whether it is configured as bipolar or unipolar. A common issue with this motor is overheating at high currents. It can also be noisy at low speeds. The specific model we are using is a bipolar one with a torque of 0.59Nm and a maximum current of 2A.

3.3 TB6600 stepper driver

The TB6600 driver is a controller for bipolar stepper motors, commonly used in automation and robotics applications. It is designed to handle currents up to 4.5 A and voltages up to 40 V, making it suitable for medium-sized motors. The driver offers various microstepping modes (e.g. 1/1, 1/2, 1/8, 1/16) for precise control of movement and resolution. In addition, it includes protection features such as detection of overcurrent, overvoltage, and overheating, ensuring safe and reliable operation. The TB6600 is often used in 3D printers, CNC machines, and other computer-controlled equipment.

The maximum current our nema 17 steppers support is 2A. Microstepping can be set to increase the precision of the motors; however, this decreases the maximum torque that can be delivered. With these considerations, we can modify the dip switches on the board. We will use a total of 4 drivers for the corresponding 4 motors. Note that two of these (the ones used to move the second coupling) will have to move with a synchronous PWM, in the opposite direction.

To control a motor with this driver, it will then be sufficient to set the maximum current drawn by the motors, choose a microstepping, and connect the 3 control GPIOs to the MCU. Specifically, the MCU must handle the enable that unlocks the motors, a pwm whose frequency handles the motor speed, and a gpio dedicated to the direction of rotation.

3.4 MG996r servomotor

The MG996R is a Tower Pro type servo motor that allows precise control of angular position. It is an analog servo motor with a typical rotation range of 180 °.

The MG996R provides a torque of approximately 10 Kg·cm at 4.8 V and 12 Kg·cm at 6 V. The rotation speed is about 0.19 seconds per 60 ° at 4.8 V and 0.15 seconds per 60° at 6 V. It operates with a supply voltage between 4.8 V and 7.2 V, and in our case, we will power it with 5 V. The control cable consists of three wires: red for power, brown or black for ground, and yellow or orange for the PWM signal.

The MG996R is controlled via a PWM signal (Fig. 6) signal sent to the signal pin (yellow/orange). The pulse width determines the position of the servo motor: a pulse of 1.5 ms corresponds to the center position (0°), a pulse of 0.5 ms moves the servo to the extreme position in the counterclockwise direction (-90°), while a pulse of 2.5 ms moves it to the extreme position in the clockwise direction (+90°). The control of the rotation angle is therefore achieved by varying the respective duty cycle.

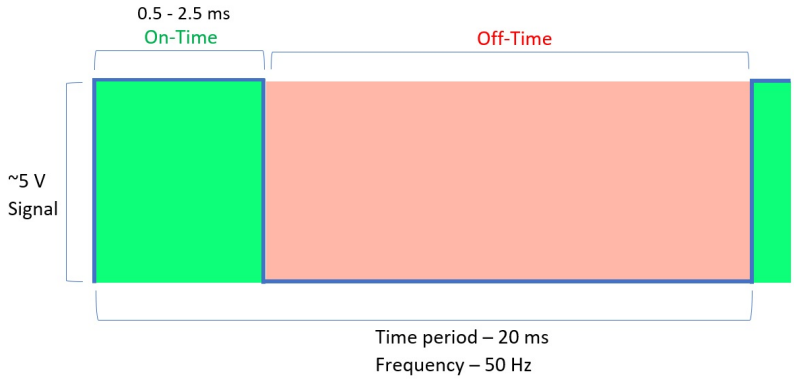


Figure 6: PWM control signal for servo

3.5 End effector actuator

The end-effector can be of a different nature. In the first simpler case, the use of an electromagnet, it requires only a high or low logic value for enable, so the use of another relay as in our case might be fine.
As a coil, the electromagnet's power will depend on the number of windings and the current drawn. The electromagnet used, mounted in the modeled support, is shown in Figure 7.

Figure 7: Electromagnet integrated into our custom end-effector support system

If we talk about a gripper, we implement, for example, DC motors, of which it is essential to control speed (thus supply voltage) and direction. This type of control would require a more complex dedicated circuit.
A first simple actuation strategy could involve maintaining a fixed speed for opening and closing the gripper (at constant voltage), only reversing the DC motor's supply polarity. This can be achieved with a simple H-bridge implemented using MOSFETs or relays.

3.6 Other components

Voltage to the various stepper motors is provided by a 24V power supply with a maximum deliverable current of 12A. A step-down device is also required to provide, from 24V, the 5V voltage useful for all secondary peripherals. To force the movement of the manipoalator, even when it is running, the stepper enable must be activated. However, for servomotors, it is necessary to disconnect the power supply; this is done by means of relays.

3.7 Schematic

The circuit diagram in figure 8 shows the interconnections between the components, with the labels indicating the connections to the STM32F446RE board, including the PWM signals generated by the timers and the GPIOs configured as command outputs.

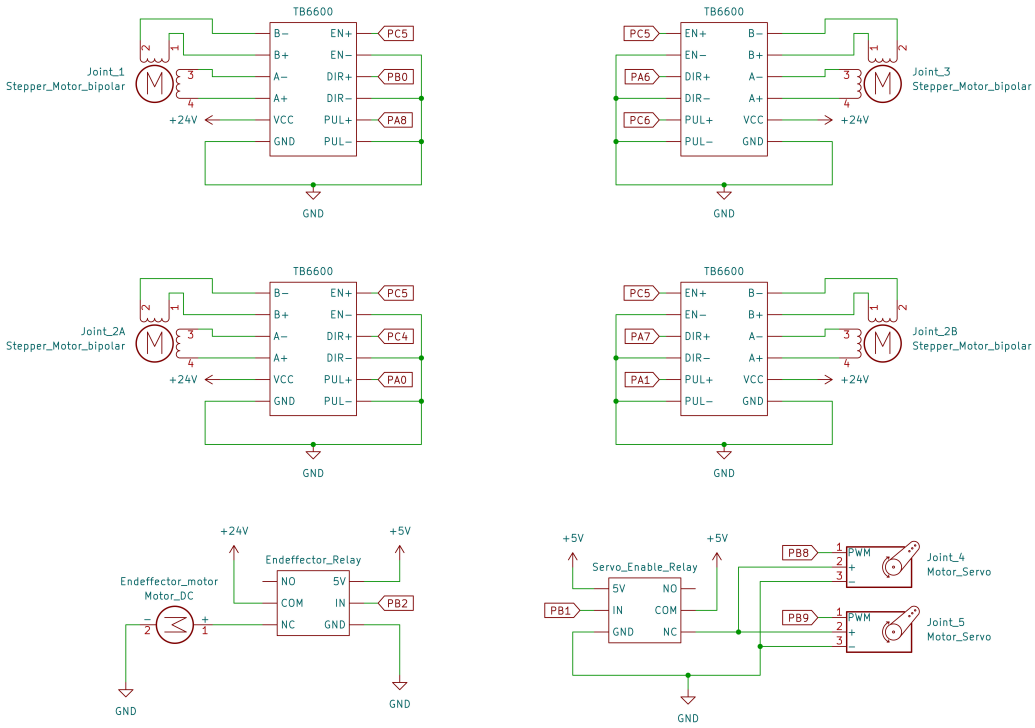


Figure 8: Kicad schematic

4 Operating Principles

4.1 Control Strategy and Trajectory Planning

Let us briefly quote a paragraph from the book that clearly encapsulates the problem to be addressed:
"The joint space control problem is articulated in two subproblems. First, manipulator inverse kinematics is solved to transform the motion requirements xd from the operational space into the corresponding motion qd in the joint space. Then, a joint space control scheme is designed that allows the actual motion q to track the reference inputs. However, this solution has the drawback that a joint space control scheme does not influence the operational space variables xe which are controlled in an open-loop fashion through the manipulator mechanical structure. It is then clear that any uncertainty of the structure (construction tolerance, lack of calibration, gear backlash, elasticity) or any imprecision in the knowledge of the end-effector pose relative to an object to manipulate causes a loss of accuracy on the operational space variables." [3]

So the first steps we planned were first to test each stepper motor motion individually, understand the operating range of the stepper motors by running the "worst-case motion" per joint, after which we took points on MATLAB in the operating space and through inverse kinematics, set the values of the joint variables. Finally, we tested on the actual robotic arm by recalibrating the values of the joint variables to make it travel a predetermined trajectory. The problem with open-loop control is that the error between theoretical and actual position is assumed to be zero and thus there is no loss of stepper motor steps. This assumption is quite strong but may make sense in the case where we limit the maximum speed frequency in testing and in the case where, as in our case, we approach it with a trapezoidal speed control algorithm. Next, we evaluate the Torque-PPS (pulses/s) graph of the stepper motor (Fig. 9), noting how it is not possible to reach a high speed instantaneously without loss of steps due to a minimum torque required by the motor unless you start with a lower speed and gradually go up.

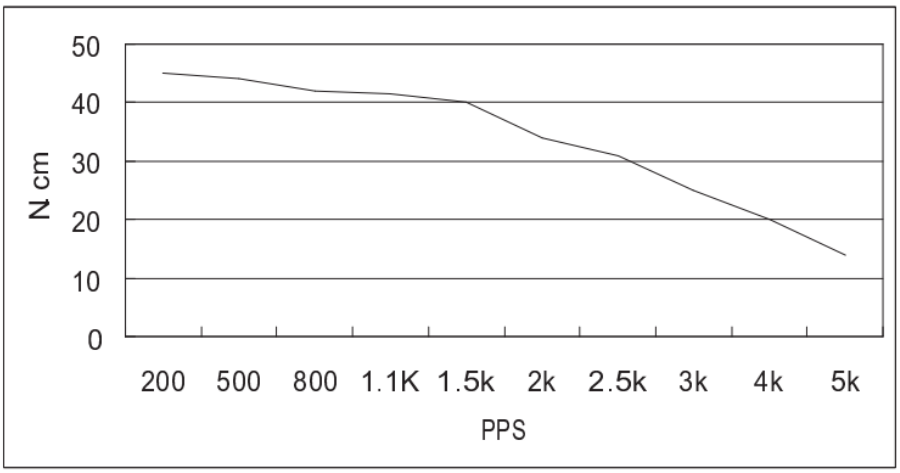


Figure 9: PPS-Torque Diagram

4.2 Preliminary Test - Frequency Range Selection

The frequency of the PWM (Pulse Width Modulation) that controls the stepper motors plays a fundamental role in their operation. Frequencies that are too low will tend to make the motor noisy and cause it to overheat, while high frequencies result in lower torque—which is particularly important in our case, as the PWM frequency is closely linked to the motor's angular velocity. Considering the number of steps the motor completes in a full revolution (200 for the steppers used), the angular velocity at the motor shaft is given by 1.
The angular velocity ω (in rounds per second) of a stepper motor is given by the pulse frequency f_{step} divided by the number of steps per revolution N and microstepping m, thus expressing the relationship between speed and control signals.

$$\omega = \frac{f_{\text{step}}}{N \cdot m} \tag{1}$$

Before implementing a trapezoidal motion profile on stepper motors, it is necessary to determine a **frequency range** (and thus joint speeds) that ensures **sufficient torque** to smoothly move each joint. This prevents **step losses**, a critical requirement for **open-loop control** systems like ours.

A basic test involves comparing:

- **Theoretical position**: Obtained via software using a dedicated 16 bit step-counting timer
- **Actual position**: Measured physically (accounting for the gear ratio, see table 1)

The frequency of the STM32 timer generating the PWM signal can be set by adjusting:

- **Prescaler**
- **Auto-reload register (ARR)**

using the formula:

$$f_{\text{PWM}} = \frac{f_{\text{clock}}}{(\text{Prescaler} + 1) \times (\text{ARR} + 1)} \tag{2}$$

The test results are summarized in Table 2:

joint	ARR_{min}	PSC_{min}	f_{min}	ARR_{max}	$PSCmax$	f_{max}	Ratio	micro-step	master TIM	slave TIM
1	65535	4	256.35	19999	4	840.00	4.27	4	TIM1	TIM4
2	65535	4	256.35	32999	4	500.09	6	4	TIM2	TIM3
3	65535	4	256.35	1999	4	8400.00	4.9	8	TIM8	TIM5

Table 2: Range Table

As mentioned in Chapter 2.1, for the second joint, two twin motors are used, which are controlled mirrored, which means that they receive the same PWM signal and identical configuration.

In addition to the appropriate frequency range, a microstepping value allowed by the driver is selected for each motor.
Microstepping is a technique used in stepper motors to divide each full step into smaller increments by controlling the current in the motor's two windings in a sinusoidal manner. This allows for smoother and more precise movements, reducing vibrations and noise. The main advantage is achieving greater precision and movement smoothness; however, it results in lower available torque at very small microsteps.

4.3 MatlabKinematics.m

MATLAB's Robotics Toolbox calculates joint variables (θ_1, θ_2, ...) and trajectories using functions like `inverseKinematics`, but these are theoretical values. In practice, adjustments are needed to account for real-world factors like friction, mechanical backlash, and physical limits.

From the 3D model, we extracted the structural constraints for each joint of the manipulator. The virtual robot was modeled in MATLAB (Fig. 10) using Denavit-Hartenberg (DH) parameters derived from the SolidWorks assembly. Subsequent hardware validation necessitated parameter optimization to account for non-ideal joint behavior.

Listing 1: MatlabKinematics

```
clear;close all;

a1=220/100; a2=7.5/100; a3=(65.5+130)/100; a4=30/100; %[mm]

link1 = rigidBody("link1");
link1.Joint = rigidBodyJoint("joint1","revolute");
link1.Joint.setFixedTransform([0 pi/2 0 0],"dh");
link1.Joint.HomePosition=0;

link2 = rigidBody("link2");
link2.Joint = rigidBodyJoint("joint2","revolute");
link2.Joint.setFixedTransform([a1 0 0 0],"dh");
link2.Joint.HomePosition=pi/2;
link2.Joint.PositionLimits=deg2rad([-230/2 +230/2]);

link3 = rigidBody("link3");
link3.Joint = rigidBodyJoint("joint3","revolute");
link3.Joint.setFixedTransform([0 pi/2 a2 0],"dh");
link3.Joint.HomePosition=deg2rad(-43);
link3.Joint.PositionLimits=deg2rad([-270/2 +270/2]);

link4 = rigidBody("link4");
link4.Joint = rigidBodyJoint("joint4","revolute");
link4.Joint.setFixedTransform([0 -pi/2 a3 0],"dh");
link4.Joint.PositionLimits=deg2rad([-270/2 +270/2]);

link5 = rigidBody("link5");
link5.Joint = rigidBodyJoint("joint5","revolute");
link5.Joint.setFixedTransform([0 pi/2 0 0],"dh");
link5.Joint.PositionLimits=deg2rad([-180/2 +180/2]);

link6 = rigidBody("end_eff");
link6.Joint = rigidBodyJoint("end_eff","fixed");
link6.Joint.setFixedTransform([0 0 a4 0],"dh");

myRobot = rigidBodyTree(DataFormat="row");

myRobot.addBody(link1,myRobot.BaseName);
myRobot.addBody(link2,link1.Name);
myRobot.addBody(link3,link2.Name);
myRobot.addBody(link4,link3.Name);
myRobot.addBody(link5,link4.Name);
myRobot.addBody(link6,link5.Name);

myRobot.homeConfiguration
myRobot.showdetails

figure(1);
myRobot.show(myRobot.homeConfiguration);

```

```
ik = ...
    inverseKinematics('RigidBodyTree',myRobot,'SolverAlgorithm','LevenbergMarquardt');

ikWeights = [0 0 0 1 1 1]; % [orientation_weight, position_weight]

desiredPosition = trvec2tform([-1.19, 0.077, 1.602]); %  position example

[configSol, solInfo]= ik("end_eff", desiredPosition, ikWeights, ...
    myRobot.homeConfiguration );

figure(2)
myRobot.show(configSol);
```

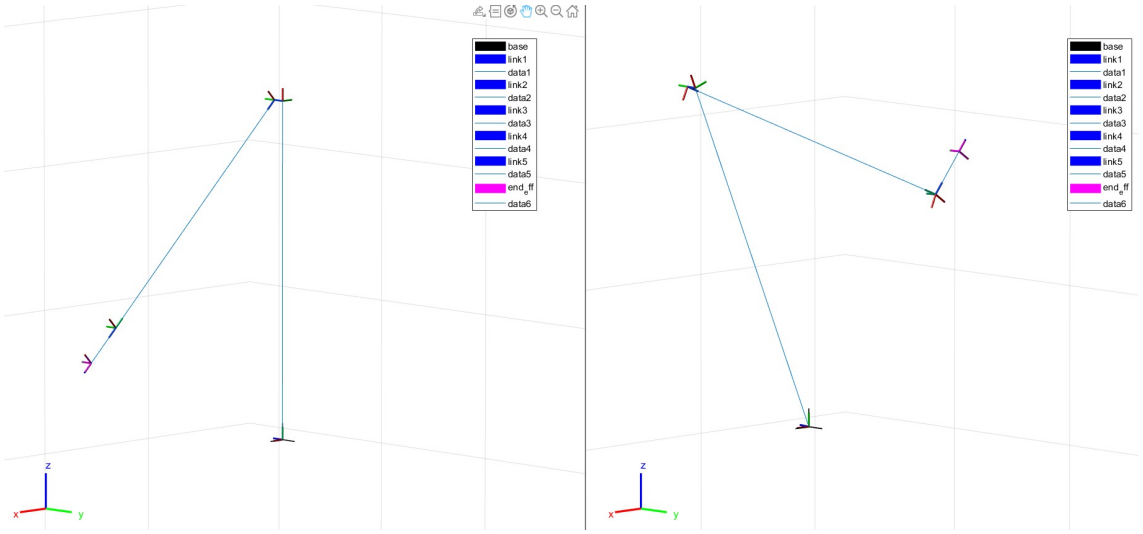


Figure 10: Matlab Example Inverse Kinematics from left to right

4.4 Trapezoidal Profile Minimum Time

Trapezoidal velocity profiles aim to minimize the settling time of the load velocity in rest-to-velocity motion by optimizing the acceleration time and jerk. These profiles are characterized by symmetrical trapezoidal shapes, where the acceleration time is minimized while considering the limit value for motor acceleration.To achieve the minimum settling time, the acceleration and deceleration phases of the trapezoidal profile must be carefully chosen. The acceleration time must be optimized to ensure that the motor reaches the desired velocity as quickly as possible, while also minimizing the residual vibrations that can occur.

The trapezoidal approach divides the trajectory into three distinct phases:

- Acceleration phase: The robot accelerates from rest to its desired velocity, which is less than the maximum velocity reachable by that specific motor. During this phase, the acceleration rate is controlled to avoid sudden jerks.
- Constant velocity phase: Once the robot reaches its desired velocity, it maintains a constant speed. This phase ensures smooth motion without abrupt changes.
- Deceleration phase: As the robot approaches the target position, it decelerates to come to a stop. Similar to the acceleration phase, controlled deceleration prevents sudden stops.

In our case the minimum time trapezoidal profile has been implemented where it is possible to coordinate the start and end times in unison of the various motors through the debugger by acting on the desired frequency and *accel_ rate* of the motor. It is however possible in the future to modify the code making it possible through a priori controls to act on the overall time of the trajectory.

5 The most important parts of the Code

The development of code and the configuration of peripherals are simplified with the STM32CubeIDE [4], an integrated development environment that, in combination with the STM32CubeMX, provides a user-friendly graphical interface for the setup of peripherals. Developers benefit from tools for real-time debugging and performance optimization. The software ecosystem includes HAL (Hardware Abstraction Layer) and LL (Low-Layer) libraries, along with example projects provided by STMicroelectronics, accelerating prototyping and development.
The pin configuration for this project required the use of timers in master and slave mode, interrupts and non-interrupts, gpio and so on. The final configuration is made available in the repository [5].

5.1 stepper.h/c

The following code is the header of the stepper library, and includes the stepper struct and the main functions.

Listing 2: stepper.h

```
1
2  typedef struct {
3      //timers
4      TIM_HandleTypeDef *pwm_timer; //master timer
5      TIM_HandleTypeDef *position_timer; //SLAVE timer
6      //
7      GPIO_TypeDef *direction_port;
8      uint16_t direction_pin;
9      //stepper data
10     float stepper_resolution;
11     float step_per_rev; // = 360/stepper_resolution;
12     uint16_t microstep;
13     float step_scale; //= step_per_rev*microstep;
14
15 } stepper_obj;
16
17 typedef enum {
18     CLOCKWISE = 1, COUNTERCLOCKWISE = 0
19
20 } direction_str;
21 void stepper_init(stepper_obj *stp, TIM_HandleTypeDef *pwm_timer,
22         TIM_HandleTypeDef *position_timer, float stepper_resolution,
23         uint16_t microstep, GPIO_TypeDef *direction_port,
24         uint16_t direction_pin);
25
26 void stepper_move(stepper_obj *stp, direction_str direction, float position,
27         float freq_desired);
```

In practice with stepper motors you have to calculate the number of steps for each input angle that it has to make. This calculation changes based on the specifications and use cases of the motor, to make everything as agnostic as possible, the calculation is done in the init function and future movements will be based on this.

Listing 3: stepper.c

```
1 int n_steps_a[3]; //steps that the single timer have to perform
2 int arr_des_a[3]; //arr that should be reached
3
4 static int flag_configured_timer2; //for the timer2 config
5
```

```
6  //float freq_des_steps; //DEBUG
7
8  void stepper_move(stepper_obj *stp, direction_str direction, float position,
9          float freq_desired) {
10
11     int arr_des;
12     float freq_des_steps;
13     int i = 0;
14     int n_steps = stp->step_scale * position / 360.0f; //[n_steps]
15
16     freq_des_steps = stp->step_scale * freq_desired / 360.0f; //[n_steps/s]
17
18     arr_des = (HAL_RCC_GetPCLK2Freq() * 2 / freq_des_steps)
19             / (stp->pwm_timer->Instance->PSC + 1) - 1;
20
21     HAL_GPIO_WritePin(stp->direction_port, stp->direction_pin, direction); ...
           //DIRECTION
22
23     if (stp->pwm_timer->Instance != TIM2) {
24
25         //reset_timers(stp);
26
27         __HAL_TIM_SET_AUTORELOAD(stp->position_timer,
28                 (n_steps * (stp->pwm_timer->Instance->PSC + 1)) - 1);
29         __HAL_TIM_SET_COMPARE(stp->pwm_timer, TIM_CHANNEL_1,
30                 __HAL_TIM_GET_AUTORELOAD(stp->pwm_timer)/2);
31
32         HAL_TIM_PWM_Start_IT(stp->pwm_timer, TIM_CHANNEL_1); //START PWM
33
34     } else { //set parameters for the timer2 separately cause it has 2 channel
35         if (flag_configured_timer2 != 1) { //this cause the second stepper must ...
               be equal to the first one
36
37             //reset_timers(stp);
38             __HAL_TIM_SET_AUTORELOAD(stp->position_timer,
39                     (n_steps * (stp->pwm_timer->Instance->PSC + 1)) - 1);
40             __HAL_TIM_SET_COMPARE(stp->pwm_timer, TIM_CHANNEL_1,
41                     __HAL_TIM_GET_AUTORELOAD(stp->pwm_timer)/2);
42
43             __HAL_TIM_SET_COMPARE(stp->pwm_timer, TIM_CHANNEL_2,
44                     __HAL_TIM_GET_AUTORELOAD(stp->pwm_timer)/2);
45
46         } else {
47             HAL_TIM_PWM_Start_IT(stp->pwm_timer, TIM_CHANNEL_1); //START PWM
48             HAL_TIM_PWM_Start_IT(stp->pwm_timer, TIM_CHANNEL_2); //START PWM)
49
50         }
51         flag_configured_timer2 ^= 1;
52
53     }
54     n_steps = n_steps * (stp->pwm_timer->Instance->PSC + 1); //update the var and ...
           bring the value outside
55
56     if (stp->pwm_timer->Instance == TIM1) {
57         i = 0; //timer joint 1
58     } else {
59         if (stp->pwm_timer->Instance == TIM2) {
60             i = 1; //timer joint 2
61         } else
62             i = 2; //timer joint 3
63     }
64     n_steps_a[i] = n_steps;
65     arr_des_a[i] = arr_des;
66
67 }
```

The stepper_move function calculates a series of values to assign to variables such as the number of steps to take assigned to the ARR of the slave timer, frequency, desired ARR value, etc.
Once the various operations have been performed, the timers are configured and the PWM signal is generated in interrupt mode; there is also a check to prevent the timer from being set twice.
In the end the timer arrays are updated to perform the minimum time trapezoidal control, it will be explained better later.

5.2 servo.h/c

Listing 4: stepper.h

```
    //ccr for 50hz
#define CCR_MAX 7499.0
#define CCR_MIN 1499.0
#define ANGLE_MAX 90.0 //+-deg

#include "main.h"

typedef struct {
    //timers
    TIM_HandleTypeDef *pwm_timer; //master timer
    float unit;

} servo_obj;

void servo_init(servo_obj *srv, TIM_HandleTypeDef *pwm_timer);

void servo_move(servo_obj *srv, float position);
```

The previous code is the header of the servo motor. The servo is controlled by a 50Hz PWM signal and, based on this, the corresponding CCR has been computed also considering that it is possible to go even beyond the datasheet values. The values have been calculated considering the range 0.5-2.5ms and an ARR of 60000;

Listing 5: servo.c

```
void servo_init(servo_obj *srv, TIM_HandleTypeDef *pwm_timer) {
    srv->pwm_timer = pwm_timer;
    srv->unit = (CCR_MAX - CCR_MIN) / (2 * ANGLE_MAX);
    //pwm=50hz
    srv->pwm_timer->Instance->PSC = 27;
    srv->pwm_timer->Instance->ARR = 60000 - 1;
    __HAL_TIM_SET_COMPARE(srv->pwm_timer, TIM_CHANNEL_1,
            (int) ((CCR_MAX + CCR_MIN) / 2));
    HAL_TIM_PWM_Start(srv->pwm_timer, TIM_CHANNEL_1); //START PWM)
}


void servo_move(servo_obj *srv, float position) {

    int ccr;
    //saturation
    if (position > ANGLE_MAX)
        position = ANGLE_MAX; //max angle position available
    if (position < -ANGLE_MAX)
        position = -ANGLE_MAX; //min angle position available

    //compute ccr value
    if (position != 0)
        ccr = (int) ((CCR_MAX + CCR_MIN) / 2) + (int) (srv->unit * position);
```

```
25
26      else {  //position==0
27          ccr = (int) ((CCR_MAX + CCR_MIN) / 2); /*0 deg */
28      }
29      __HAL_TIM_SET_COMPARE(srv->pwm_timer, TIM_CHANNEL_1, ccr);
30      srv->pwm_timer->Instance->EGR = TIM_EGR_UG; //reset timer
31
32  }
```

So already in the init function the PWM is generated for a default value of 0 degrees on the servo which can be changed later at any time with the servo_move function which modifies the CCR value.

5.3 trapezoidal_profile.h/c

For the minimum time trapezoidal control, different approaches were used, the results that were ultimately the most valid are two: the first method works entirely in interrupt mode and theoretically is the one that ensures better response times leaving the CPU almost inactive.
The second method uses a sort of polling mode inside a function called only with pwm updates (this in interrupt mode), updating a series of vectors and the slave timer. The second method was developed because of a problem of competition between interrupts of the first method in the case in which one wanted to move the three stepper motors together. In this case, one could act on the frequency of the steppers so as not to have too many interrupts from one motor that overwhelms the others, manage the priority of the interrupts in some way, use HAL_Delays to start the "slow" interrupts first or, as was done, reduce the quantity of interrupts. This method solves for the most part the problem of being able to start multiple motors at the same time, although you still have to play a bit with the HAL_Delay when you have to stop at one point and then start again, for this reason the debugger was very useful to understand if the motor accelerated, decelerated, remained constant, if you had to increase/decrease the delay or if there was some other problem.

Listing 6: trapezoidal_profile.c

```
1
2  void HAL_TIM_PWM_PulseFinishedCallback(TIM_HandleTypeDef *htim) {
3
4      if (htim->Instance == TIM1) {//joint1
5
6          count_rising_edge[0] += (htim->Instance->PSC + 1);
7          //arr[0] = -(__HAL_TIM_GET_AUTORELOAD(&htim1));  //debug graph
8          trapezoidal_func(0, htim, &htim4); //0 = the joint1 timer
9
10     }
11
12     ...
13 }
```

This function is called with a frequency given by the PWM signal of the master timer scaled by the PSC, for this reason the artificial counter used in this implementation is incremented by this amount and then the function is called:

Listing 7: trapezoidal_profile.c

```
1  void trapezoidal_func(int k, TIM_HandleTypeDef *htim, TIM_HandleTypeDef *hslave) {
2
3      int cnt = __HAL_TIM_GET_COUNTER(hslave);  //retrieve the N_steps done by the ...
           slave
4      if (cnt ≥ hslave->Instance->ARR) { //verify if arr has been reached
```

```
5           if (hslave->Instance == TIM3) { //for the timers 2 the second pwm must be ...
                stopped too
6               HAL_TIM_PWM_Stop_IT(htim, TIM_CHANNEL_2);
7           }
8
9           HAL_TIM_PWM_Stop_IT(htim, TIM_CHANNEL_1); //stop the pwm
10
11          count_rising_edge[k] = 0;   //reset the counter
12          acc_count[k] = 0;           //reset the acceleration steps counter
13          //dec_count[k] = 0; //debug
14
15          htim->Instance->EGR |= TIM_EGR_UG; //reset the timer master
16          hslave->Instance->EGR |= TIM_EGR_UG; //reset the timer slave
17
18      } else {
19
20          if (arr_des_a[k] > arr_start[k]) { //subroutine to set the starting ...
                minimum acceleration
21              __HAL_TIM_SET_AUTORELOAD(htim, arr_start[k]);
22              hslave->Instance->PSC = arr_start[k];
23              __HAL_TIM_SET_COMPARE(htim, TIM_CHANNEL_1,
24                      __HAL_TIM_GET_AUTORELOAD(htim) / 2);
25
26              hslave->Instance->EGR |= TIM_EGR_UG;
27              hslave->Instance->CNT = count_rising_edge[k];
28
29          }
30
31          if (arr_des_a[k] < (arr_current[k] - acc_rate_a[k])
32                  && cnt ≤ (int) (n_steps_a[k] * 1 / 2)) { //acceleration phase
33
34              arr_current[k] -= acc_rate_a[k];
35              if (arr_current[k] ≤ arr_max[k]) //max velocity saturation
36                  arr_current[k] = arr_max[k];
37
38              __HAL_TIM_SET_AUTORELOAD(htim, arr_current[k]);
39              hslave->Instance->PSC = arr_current[k];
40              __HAL_TIM_SET_COMPARE(htim, TIM_CHANNEL_1,
41                      __HAL_TIM_GET_AUTORELOAD(htim) / 2);
42
43              hslave->Instance->EGR |= TIM_EGR_UG;
44              hslave->Instance->CNT = count_rising_edge[k];
45
46              acc_count[k] += (htim->Instance->PSC + 1); //increase the acc_count
47
48          }
49
50          else {
51              if (arr_des_a[k] ≥ (arr_current[k] - acc_rate_a[k])
52                      && arr_current[k] > arr_des_a[k]
53                      && cnt ≤ (int) (n_steps_a[k] * 1 / 2)) { //constant phase
54
55                  __HAL_TIM_SET_AUTORELOAD(htim, arr_des_a[k]);
56                  hslave->Instance->PSC = arr_des_a[k];
57                  __HAL_TIM_SET_COMPARE(htim, TIM_CHANNEL_1,
58                          __HAL_TIM_GET_AUTORELOAD(htim) / 2);
59
60                  hslave->Instance->EGR |= TIM_EGR_UG;
61                  hslave->Instance->CNT = count_rising_edge[k];
62
63              }
64
65              else {
66
67                  if (cnt ≥ (n_steps_a[k] - acc_count[k])) { //deceleration phase
```

```
68                          arr_current[k] += acc_rate_a[k];
69
70                          if (arr_current[k] ≥ (arr_start[k] - acc_rate_a[k])) //min ...
                                velocity saturation
71                              arr_current[k] = arr_start[k];
72
73                          __HAL_TIM_SET_AUTORELOAD(htim, arr_current[k]);
74                          hslave->Instance->PSC = arr_current[k];
75                          __HAL_TIM_SET_COMPARE(htim, TIM_CHANNEL_1,
76                                  __HAL_TIM_GET_AUTORELOAD(htim) / 2);
77
78                          hslave->Instance->EGR |= TIM_EGR_UG;
79                          hslave->Instance->CNT = count_rising_edge[k];
80                      }
81                  }
82              }
83          }
84  }
```

Then the speed control was performed by increasing or decreasing the ARR according to the various configurations, value controls and various resets, allowing to obtain a speed that varies according to the designated trajectory.
The slope of the speed is due to how much the 'accel_rate' variable of the specific engine is set which has an influence on the granularity of the trend.
With this type of control it is therefore possible to obtain a completely constant trend, in the case in which the desired input speed is lower than the 'start' speed of the motor, in which case the motor starts and ends at the minimum speed; Bang bang profile in the case in which the desired speed is not yet reached but the deceleration phase intervenes; trapezoidal profile in which the desired speed is reached and it is kept constant for as long as the trajectory lasts (it will be proportional to the amplitude of the movement).

You can see the effectiveness of the algorithm using the debugger as follows (Fig.11-12-13):

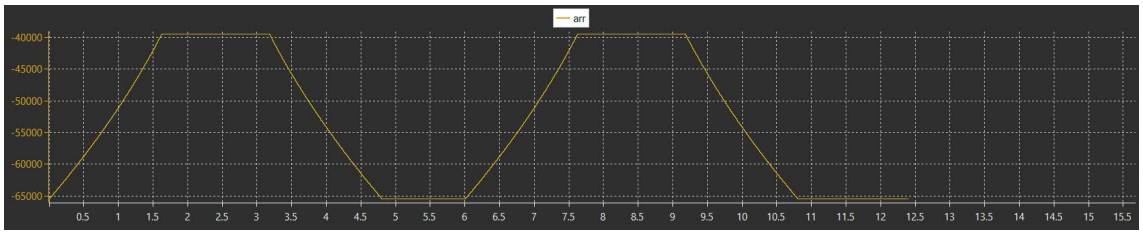


Figure 11: Trapezoidal_profile

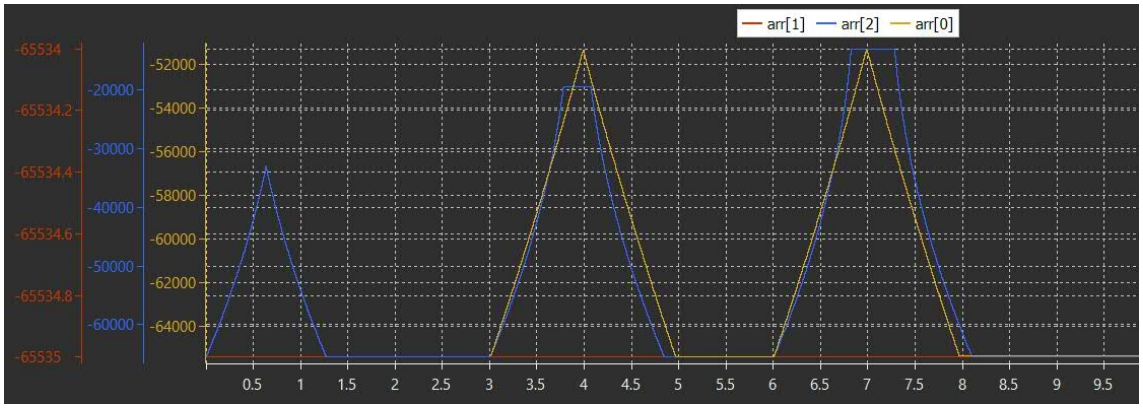


Figure 12: Trajectory_example

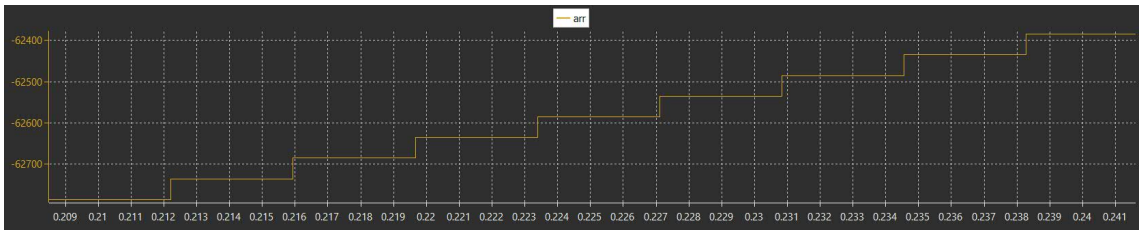


Figure 13: Granularity of the slope

5.4 end_eff_gpio.h/c

A library has also been implemented to manage an end-effector such as a gripper, with 2 GPIOs and 4 function sets.

Listing 8: end_eff_gpio.c

```
1   void gripper_config(gripper_obj *endeff, int func, int time) {
2
3   if (func == 1) { //hold- 00
4
5       HAL_GPIO_WritePin(ENDEFF1_en_GPIO_Port, ENDEFF1_en_Pin, GPIO_PIN_SET);
6       HAL_GPIO_WritePin(ENDEFF2_en_GPIO_Port, ENDEFF2_en_Pin, GPIO_PIN_SET);
7   }
8   if (func == 2) { //grip-01
9
10      HAL_GPIO_WritePin(ENDEFF1_en_GPIO_Port, ENDEFF1_en_Pin, GPIO_PIN_RESET);
11      GPIO_PinState state = HAL_GPIO_ReadPin(ENDEFF1_en_GPIO_Port,
12              ENDEFF1_en_Pin);
13      if (state == GPIO_PIN_RESET) {
14          HAL_GPIO_WritePin(ENDEFF2_en_GPIO_Port, ENDEFF2_en_Pin,
15                  GPIO_PIN_SET);
16      }
17
18  }
19  if (func == 3) { //release-10
20
21      HAL_GPIO_WritePin(ENDEFF2_en_GPIO_Port, ENDEFF2_en_Pin, GPIO_PIN_RESET);
22      GPIO_PinState state = HAL_GPIO_ReadPin(ENDEFF2_en_GPIO_Port,
23              ENDEFF2_en_Pin);
24      if (state == GPIO_PIN_RESET) {
```

```
25              HAL_GPIO_WritePin(ENDEFF1_en_GPIO_Port, ENDEFF1_en_Pin,
26                      GPIO_PIN_SET);
27          }
28
29      }
30
31      if (func == 4) { //timer
32
33          __HAL_TIM_SET_PRESCALER(endeff->htim, 1400 * time);
34          __HAL_TIM_SET_AUTORELOAD(endeff->htim, 59999);
35          endeff->htim->Instance->EGR |= TIM_EGR_UG;
36
37          __HAL_TIM_CLEAR_IT(endeff->htim, TIM_IT_UPDATE);
38          HAL_TIM_Base_Start_IT(endeff->htim);
39
40      }
41      else{
42          Error_Handler();
43      }
44  }
```

When the function is '1' the gripper holds the state, when '2' the gripper closes, when '3' it opens, when '4' the gripper switches to the hold state after a period of time defined in input in seconds.

5.5 stp_callback.c

This function is called only when you want to control the stepper motors without the trapezoidal_approach with a given velocity set in the configuration phase of the timers.

Listing 9: end_eff_gpio.c

```
1       void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
2       if (htim->Instance == TIM5) {
3           HAL_TIM_PWM_Stop_IT(&htim8, TIM_CHANNEL_1);
4           htim8.Instance->EGR |= TIM_EGR_UG;
5
6           ...
```

Simply the slave timer stops the PWM of the master timer when it counts the total steps the motor has to do.

5.6 main.c

```
1           //FIRST MOVEMENT
2       servo_move(&srv2, -90);
3       stepper_move(&stp1, COUNTERCLOCKWISE, 60, 60);
4       stepper_move(&stp4, CLOCKWISE, 35, 40);
5       HAL_Delay(1600);
6
7       //SECOND MOVEMENT
8       stepper_move(&stp4, CLOCKWISE, 7, 16);
9       stepper_move(&stp2, CLOCKWISE, 30, 40);
10      stepper_move(&stp3, COUNTERCLOCKWISE, 30, 40);
11      servo_move(&srv2, 0);
12      HAL_Delay(780);
13
14      //THIRD MOVEMENT
15      stepper_move(&stp4, COUNTERCLOCKWISE, 7, 16);
16      servo_move(&srv2, -90);
17      HAL_Delay(20);
```

```
18      stepper_move(&stp2, COUNTERCLOCKWISE, 30, 40);
19      stepper_move(&stp3, CLOCKWISE, 30, 40);
20      stepper_move(&stp4, COUNTERCLOCKWISE, 35, 40);
```

This is an example of a trajectory implemented in the main. Delay times have been changed based on: trajectory planning, minimum trajectory time for a given movement (you can't make him do another move if the previous one is still in progress), delay time for a given joint. These times were calculated and recalibrated during debugging with the SWD Data Trace Timeline Graph tool of the STM32CubeIDE and Live Expression tool for constant velocity.

6 Results

Having an open-loop control we can consider the position error to be zero only if we have no stepper motor step losses, however, we cannot in any way correct additional errors due to mechanical tolerances of the structure.

Once the hardware and software were configured, we conducted a series of tests to evaluate the reliability and reproducibility of the system. The main objective was to verify the system's ability to reproduce a trajectory without losing steps and without incurring positioning errors.

We perform a first, simple trajectory (Fig: 14a), only suitable for verifying the correct operation of MATLAB's inverse kinematics and the general control software that implements the trapezoidal profile at the joints. In this phase, different velocities are tested for the same trajectory in conjunction with the ranges defined in the Table 2. In this simple trajectory a metal ball is grasped by the end effector and deposited in a container.

For the second test (Fig: 14b), a trajectory was implemented to evaluate the repeatability of the system, specifically the metal ball is grasped from a known position and slid down a chute that will lead it to the same starting position. The repeatability of the test is verified by the cyclic conduction of this trajectory.

(a) **(b)**

Figure 14: Trajectories test

7 Conclusions and Future Developments

The implementation of a trapezoidal velocity profile in the joints has been shown to enable higher speeds, which would otherwise be unfeasible without incurring step losses. Additional benefits include reduced current consumption during movement and a significant decrease in acoustic noise.

The conducted tests confirm the feasibility of open-loop control for the robot when using trapezoidal velocity profiles. However, this approach remains valid only within specific frequency and torque ranges tolerated by the stepper motors. Exceeding these limits inevitably results in step losses, causing a discrepancy between the displacement computed by the software and the actual motion executed by the manipulator.

To address this issue, a closed-loop control system based on position and velocity feedback is required. Implementing such an upgrade necessitates the integration of encoders for each joint or, alternatively, for each motor, considering the transmission ratio. It should be noted that this modification would also require structural adjustments to the mechanical system. Another possible solution involves the use of specialized stepper motors with internal drivers designed to prevent step losses. However, this approach would likely require an upgrade of the microcontroller unit (MCU) due to the current saturation of timers essential for proper encoder management.

Another potential improvement is the integration of limit switches or feedback signals, which would facilitate the manipulator's return to a reference position. Additionally, the choice of materials used in the 3D-printed structure significantly impacts joint precision, particularly in the pinion-gear coupling. Higher-quality prints directly reduce positioning errors, thus improving control accuracy, while more resistant materials would allow for the application of higher torque ranges.

The integration of encoders in the motors could also enable the implementation of more advanced and centralized control strategies. The developed 3D model allows for accurate parameterization of the manipulator's nonlinear dynamics, making it possible to estimate inertia and gravity matrices for motion and force control and could be integrated into a simulation environment such as Simulink for prototyping different control architectures.

From a software perspective, while maintaining the current MCU, an additional improvement could be the adoption of a Real-Time Operating System (RTOS), which would provide multiple advantages, including parallel task execution, more efficient process priority management, and greater system scalability.

References

[1] Dassault Systèmes. *SolidWorks*. Software for 3D CAD design. 2024. URL: https://www.solidworks.com.

[2] *Functional Design Endeffector Project*. URL: https://github.com/Phersax/Functional-Design-Endeffector-Project.

[3] Bruno Siciliano et al. *Robotics: Modelling, Planning and Control*. Jan. 2011. ISBN: 978-1-84628-641-4. DOI: 10.1007/978-1-84628-642-1.

[4] *STM32IDE*. Last access date: 23 Mar 2025. URL: https://www.st.com/en/development-tools/stm32cubeide.html.

[5] *Dracarys Repository*. URL: https://github.com/Phersax/Dracarys_5J.