

Design of a Portable Control Platform for Rod-Driven Continuum Parallel Robots

Josué Licona Mármol

*A graduation project submitted in partial fulfillment
of the requirements for the degree of*

Bachelor of Science in Mechanical Engineering

Supervisor
Jonathan Camargo Leyva, Ph.D.



Department of Mechanical Engineering
Universidad de los Andes
Bogotá D.C., Colombia
July 9, 2024

To my mother, Gildes Ester.

Success is the sum of small efforts, repeated day in, and day out.

— Robert Collier

Abstract

This project aimed to design and implement a portable, modular, and scalable platform in conjunction with a control system tailored for rod-driven continuum parallel robots. The primary objective was to develop a versatile mounting solution that would facilitate efficient robot movement across diverse applications. The project involved the design and construction of the mounting platform, as well as the development of the control interface. A rod-driven continuum parallel robot was designed and implemented, with a focus on miniaturization and enhancement of the structure. Furthermore, the project aimed to implement a customized language specification for programming the linear actuators, enabling tailored control and operational flexibility. Additionally, an application was developed to interface with the control system, utilizing the created protocol to ensure seamless communication and precise control.

Acknowledgements

I would like to express my sincerest gratitude to God, whose guidance has been a constant source of support throughout my life. I am particularly grateful to my family, especially my mother, whose unwavering encouragement has been a significant source of strength.

I am indebted to the professors who have not only imparted their knowledge but also served as exemplary mentors, shaping my understanding and passion for mechanical engineering. I would also like to express my gratitude to my classmates for their camaraderie and support.

I would also like to extend my gratitude to Daniel Álvarez for his invaluable contribution to the actuator concept and to Santiago Usme for his assistance with the manufacturing and assembly challenges.

Contents

List of Figures	viii
List of Tables	x
1 Introduction	1
1.1 Objectives	5
1.1.1 General Objective	5
1.1.2 Specific Objectives	5
2 Target Robot Structure	6
2.1 Reference Model	6
2.2 Rod Material	7
2.3 Design and Dimensions	9
2.4 Ring Constraints	11
3 Linear Actuator	13
3.1 Motor	16
3.2 First model	18
3.3 Final model	19
3.4 Specifications	22
4 Platform Assembly	25

4.1	Electronics Case	26
4.1.1	Circuit Schematic	28
5	Control System Interface	30
5.1	Protocol Specifications	31
5.2	Command Line Application	34
5.2.1	Architecture	34
5.2.2	Usage	35
5.3	PID Control and Tuning	38
5.3.1	Test Data	38
5.3.2	System Identification	39
5.3.3	PID Tuning	40
6	Performance and Results	42
	Conclusions	46
	Future Work and Recommendations	48
	Bibliography	50
	Appendices	52
A	Forces in Actuator Arm	53
B	PID Tuning Code	53
C	Test Code	54
D	Command Line Interface Source Code	54
E	Firmware Source Code	56
E.1	Arduino Main Entry Point	56
E.2	Persistence Memory Management Class Header	57
E.3	Persistence Memory Management Class	57
E.4	Commands Definitions	58

E.5	Actuators Instances	62
E.6	Actuator Class Header	64
E.7	Actuator Class	65

List of Figures

1.1	Traditional parallel robots	2
1.2	Continuum robots with extrinsic actuation	4
2.1	Reference robot structure	7
2.2	Constraints and range of motion	7
2.3	Maximum deflection and point load	9
2.4	Robot body constraints and parts	10
2.5	Robot body lateral views	10
2.6	Ring constraint	11
2.7	Ring constraint sizes	12
2.8	Closed-form pair constraint	12
3.1	Linear motion systems	13
3.2	3D printer extruder	14
3.3	Extruder without heater	15
3.4	Actuators arrange	15
3.5	Gear motor N20 with encoder	17
3.6	Dimensions of gear motor N20 with encoder	17
3.7	First model isometric view	19
3.8	First model dimensions	19
3.9	Final model isometric view	20

3.10	Final model dimensions	20
3.11	Actuator assembly exploded-view	21
3.12	Forces in actuator arm	22
3.13	Friction force in rod	23
4.1	Platform assembly exploded-view	26
4.2	Electronics assembly exploded-view	27
4.3	Ports and connections	28
4.4	Main circuit schematic	29
4.5	Motor circuit schematic	29
5.1	Relationships of control system components	34
5.2	System flowchart	35
5.3	Command line usage	36
5.4	Command line connection	36
5.5	Command line manual input	37
5.6	Command line file parsing	37
5.7	Input and output signals	38
5.8	Transfer function identification in MATLAB	39
5.9	Transfer function model in MATLAB	39
5.10	Blocks diagram of control system	40
5.11	System response of simulation in Simulink	41
6.1	Arm segments	42
6.2	One segment arm positions	43
6.3	Electronics case and connections	43
6.4	Complete prototype	44
6.5	Linear actuator issues	45

List of Tables

2.1	Fiberglass and AISI 302 steel properties	8
2.2	Comparison between Wu & Shi (2022) model and custom test model	9
3.1	Gear motor specifications	18
3.2	Actuator assembly parts list	22
3.3	Actuator specification	24
4.1	Platform assembly parts list	25
4.2	Electronics assembly parts list	27
5.1	G-Code definitions	31
5.2	M-Code definitions	32
5.3	M-Codes usage	32
5.4	G-Codes usage	33
5.5	Code parameters	33
5.6	PID tunings	40
6.1	Platform specifications	45

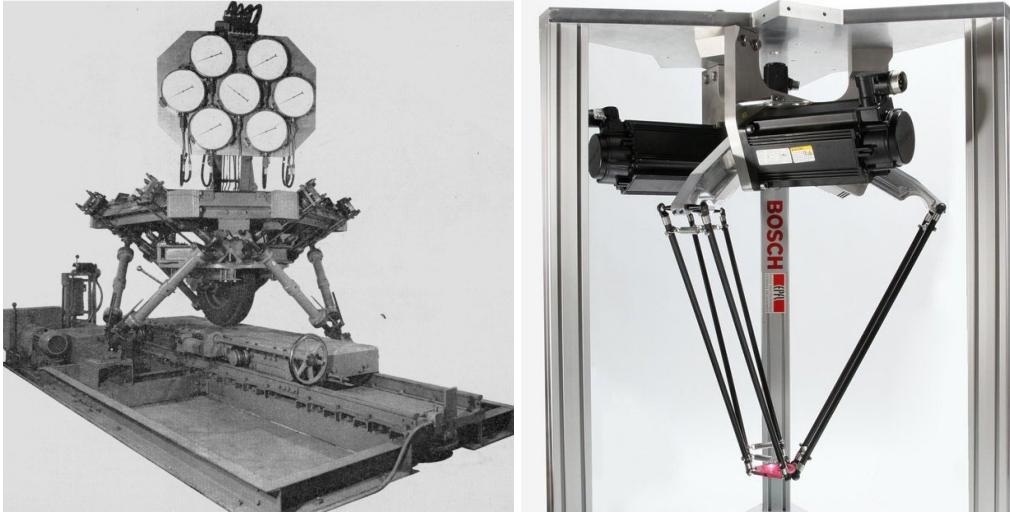
Chapter 1

Introduction

Robots can be broadly classified into two categories: serial and parallel. Serial robots are distinguished by a series of linked joints, whereas parallel robots are characterized by multiple axes that move in parallel, typically working in unison to support a single platform.

Parallel robots represent a significant branch in the field of robotics due to their precision and versatility in various applications, including pick-and-place manipulation, simulators, manufacturing, and tooling, among others. As described in [1], these applications encompass both industry and medicine, capitalizing on the high rigidity, precision, and speed of these robots to compensate for the performance limitations of serial robots.

The origin of parallel robots can be traced back to the 1960s, with the development of the Gough-Stewart [2] parallel mechanism (Figure 1.1a), which has become one of the most iconic in the field of parallel robotics. Subsequently, in the 1980s, Reymond Clavel devised a robust parallel structure comprising three translational degrees of freedom and one rotational degree of freedom, as depicted in Figure 2. The Delta Robot has become one of the most significant examples in the



(a). Gough-Stewart Platform

Source: Adapted from [2]

(b). Clavel's Delta Robot

Source: Adapted from [3]

Figure 1.1: Traditional parallel robots

field of parallel robotics.

Another distinguishing feature of these robots is the presence of multiple closed kinematic chains that connect and move the mobile platforms. In contrast, to serial robots, which operate with an open kinematic chain and move in a linear sequence. Nevertheless, according to Briot and Kahrs [4], the fundamental questions about this class of robots have been addressed. As a result, there has been a shift in interest towards exploring new alternatives in parallel robots in recent years.

The current research in robotics transcends conventional definitions, pushing the boundaries of this field and fostering new possibilities and innovations. Briot and Kahrs [4] illustrate how parallel manipulators have been integrated into various novel robot types, such as continuum robots, flying robots, cable-driven robots, underactuated robots, multi-fingered hands, and microscale parallel robots. However, this emerging class of robots presents significant scientific challenges in the design, modeling, and control of these systems. Among these categories, aerial robots, parallel robots, and continuum or soft robots are partic-

ularly noteworthy for their ability to venture into new areas due to their lightness and flexibility.

Building upon these insights, Russo et al. [5] offer a comprehensive review that addresses recent advancements, current limitations, and ongoing challenges in the design, modeling, and control of continuum robots. Continuum robots are classified according to their design, with distinction made between extrinsic and intrinsic actuation methods. Extrinsic actuation (Figure 1.2) involves transmitting motion from the robot's base along its structure. This is categorized into three main families depending on the transmission elements used: tendon-driven, concentric tube (Figures 1.2a, 1.2b), and rod-driven robots (Figures 1.2c, 1.2d). Moreover, robots with intrinsic actuation utilize actuators within their structural framework to produce motion, thereby enabling actuation to occur within the robot's body.

The current challenges in the field of parallel continuum robot design and modeling include improving through the miniaturization of actuators, the integration of these robots with rigid ones, the exploration of smart materials, the accurate modeling of environments, and the implementation of proprioception with new sensors [5]. Additionally, modeling efforts are focused on representing interaction environments and improving real-time implementations, as well as standardizing simulation environments. Control challenges include ensuring precise manipulation and adapting to dynamic environments using advanced sensor technology and adaptive strategies

This project aims to address the efficiency challenges of continuum robots by miniaturizing their parallel linear actuators. Specifically, the project focuses on the implementation of adaptive control strategies for various applications using an extrinsic actuation platform for rod-driven continuum robots. This research aims to improve affordability and ease of use in order to facilitate broader adoption and

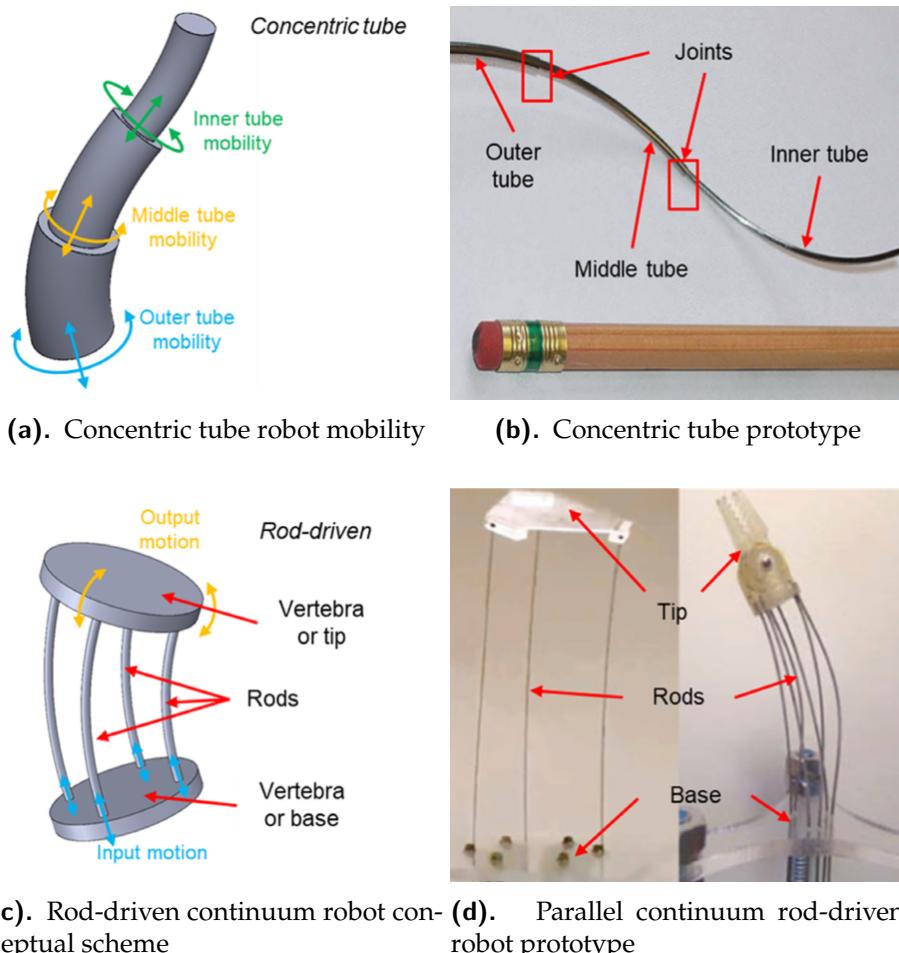


Figure 1.2: Continuum robots with extrinsic actuation

Source: Adapted from [5]

usability in practical contexts.

1.1 Objectives

1.1.1 General Objective

Design and develop a portable, modular, and scalable platform with a control system for the actuation of rod-driven continuum parallel robots.

1.1.2 Specific Objectives

- Adapt an existing design of a rod-driven continuum parallel robot, with the goal of reducing its size while maintaining its functionality and performance.
- Design linear actuators that provide precise control and enable dexterous movements, ensuring high accuracy and reliability.
- Fabricate all necessary components and assemble a fully functional physical prototype of the platform, adhering to the design specifications.
- Implement a customized language specification for programming the linear actuators, allowing for tailored control and flexibility in operations.
- Develop an application to interface with the control system, utilizing the created protocol to facilitate seamless communication and control.
- Conduct comprehensive experimental testing and validation of the system, ensuring it meets all performance criteria and operational standards.

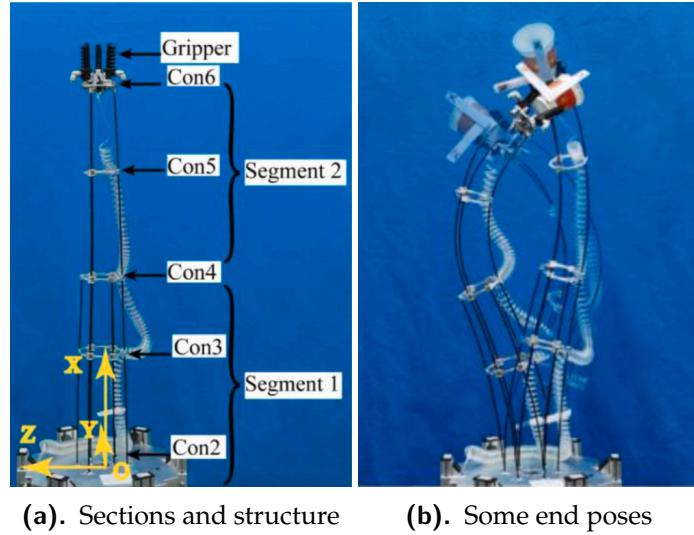
Chapter 2

Target Robot Structure

Rod-driven continuum parallel robots (RDCPR) represent a relatively novel class of robots characterized by their distinctive rod-driven actuation mechanism. It is of paramount importance to define and limit the dimensions and structural characteristics of these robots, in order to facilitate the development of a platform which will enable the robots to be moved and operated in a variety of applications. The following sections will conduct a detailed exploration of the configuration and essential components that define the structure and operational capabilities of the RDCPR.

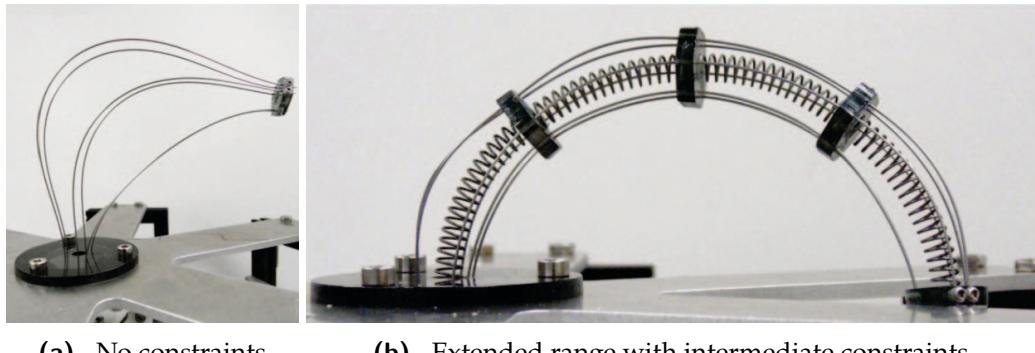
2.1 Reference Model

The reference model, depicted in Figure 2.1, was developed by Wu and Shi [6]. The design was selected for its comprehensive representation of an RDCPR, showcasing versatility and a wide range of motion. The robot features two independently manipulable segments, allowing for a total of five degrees of freedom (DoFs), which makes it extensible.

**Figure 2.1:** Reference robot structure

Source: Adapted from [6]

The incorporation of intermediate constraints, as Figure 2.2 illustrates, expands the operational range of the robot.

**Figure 2.2:** Constraints and range of motion

Source: Adapted from [7]

2.2 Rod Material

A rod is defined by the inequality $L \gg r$, where L represents its length and r its radius. In the reference model, rods are constructed from fiberglass, whereas in Figure 2.2, they are made from steel. These materials are commonly utilized due

to their high stiffness and capacity to withstand significant bending without plastic deformation or fracture. Steel alloys, such as AISI 302, which are commonly employed in springs, exemplify these properties. Table 2.1 provides the material properties of AISI 302 steel and fiberglass.

Table 2.1: Fiberglass and AISI 302 steel properties

Symbol	Property	Fiberglass	AISI 302
ρ	Density [g/cm^3]	2.6	8.0
E	Young's Modulus [GPa]	85	187.5
G	Shearing Modulus [GPa]	36	70.3

While the Cosserat rod model [5] is commonly used for analyzing these robots, understanding the maximum deflection that a rod can withstand can be determined by applying the Euler-Bernoulli beam equation 2.1, where I is given by equation 2.2. For the purposes of practicality, a configuration of AISI 302 steel rods was selected due to their commercial availability, with dimensions specified in Table 2.2. It should be noted that these rods are commonly sold in the form of wire, therefore they had to be straightened prior to use.

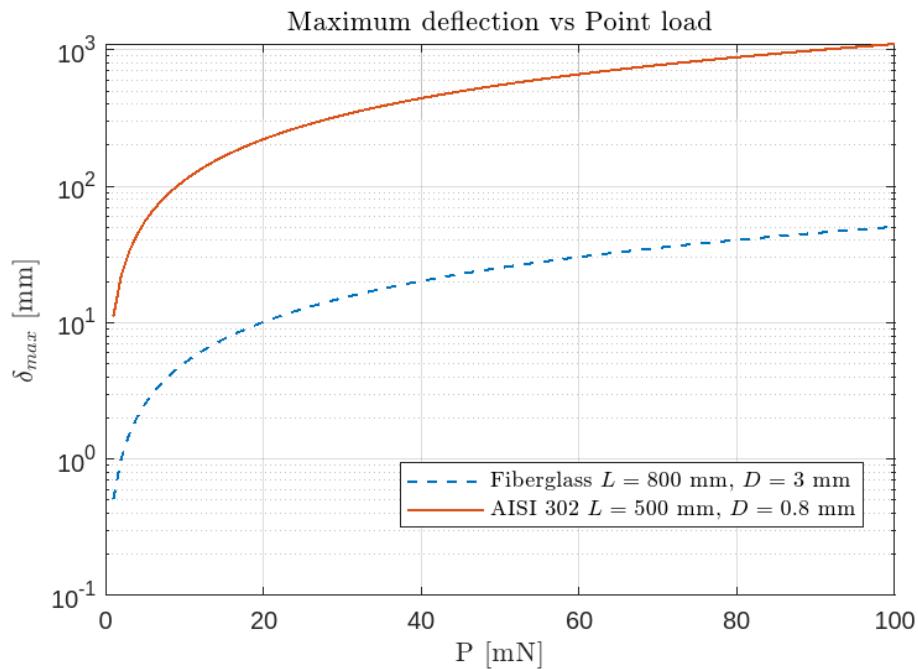
$$\delta_{max} = \frac{PL^3}{3EI} \quad (2.1)$$

$$I = \frac{\pi D^4}{64} \quad (2.2)$$

Despite the higher Young's modulus of steel and the necessity for shorter rods with significantly smaller radii, we achieved greater deflections under the same load compared to the reference model. As shown in Figure 2.3, this implies that the same deflection can be achieved with less force, which is advantageous for actuators with limited force capabilities.

Table 2.2: Comparison between Wu & Shi (2022) model and custom test model

Parameter	Wu & Shi model [6]	Custom test model
Maximum length	~ 860 mm	~ 450 mm
Minimum length	~ 400 mm	~ 250 mm
Diameters of constraints	[90, 80, 70, 60] mm	[55, 50, 45] mm
Base constraint diameter	100 mm	70 mm
Number of sections	2	2
Number of rods	6	6
Rod cross-section diameter	3 mm	0.8 mm
Rod material	Fiberglass	AISI 302

**Figure 2.3:** Maximum deflection and point load

2.3 Design and Dimensions

To discuss the design and dimensions of the robot's body, the various components are depicted in Figure 2.4. These include rods for the first and second segments, constraints, and joints between constraints and rods. The joints can be either closed-form or cylindrical-form pairs designed to restrict movement. Each intermediate segment is required to have at least one closed-form pair joint with a rod for structural stability, while the remaining joints should be cylindrical to facilitate bending of the arm. At the end of each segment, all joints within the

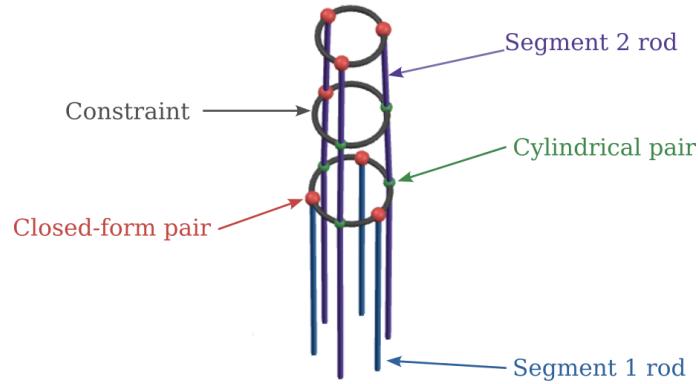
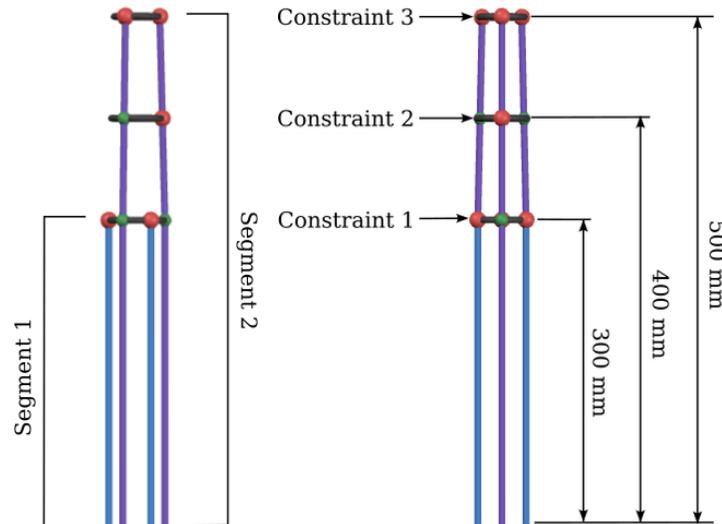


Figure 2.4: Robot body constraints and parts

constraints must be closed-form to prevent disassembly of the segment. A minimum of three rods per segment is necessary to maintain structural integrity.

Figure 2.5 illustrates the height of each segment and their respective constraints are illustrated. Additionally, the arm exhibits a tapered envelope similar to the reference model. This design choice, influenced by Wu and Shi [6], enhances structural stiffness. The tapered shape, resembling a truncated cone, is achieved by gradually reducing the cross-sectional area along the length of the arm.



(a). Left view, and (b). Frontal view, and constraints
robot body segments heights

Figure 2.5: Robot body lateral views

2.4 Ring Constraints

The proposed constraint design, depicted in Figure 2.6, takes the form of a ring or hoop and is specifically tailored for additive manufacturing. The shape of the cross-section is of great importance, since it allows for enhancement of the lateral load-bearing capabilities, which in turn mitigate deformation under load. In contrast to traditional rectangular profiles, which are prone to deformation into ellipses, the selected rhomboidal cross-section provides structural stability. This geometric configuration, which resembles a hollowed-out egg-shaped cylinder in its cross-section, offers enhanced resistance to lateral forces, thereby ensuring robust performance in rod-driven continuous parallel robots.

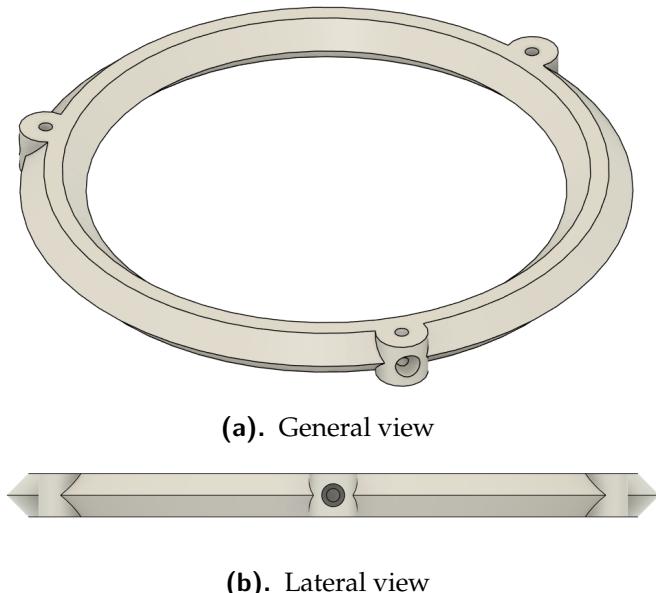


Figure 2.6: Ring constraint

The dimensions of the constraints utilized in the rod-driven continuous parallel robots exhibit variance, as illustrated in Figure 2.7. These dimensions align with the specifications outlined in Table 2.2. Each hole of the constraints is tailored to accommodate a specific rod, as indicated by the labels beside the holes. Additionally, the image depicts the type of joint with the constraint, whether closed-

form or cylindrical.

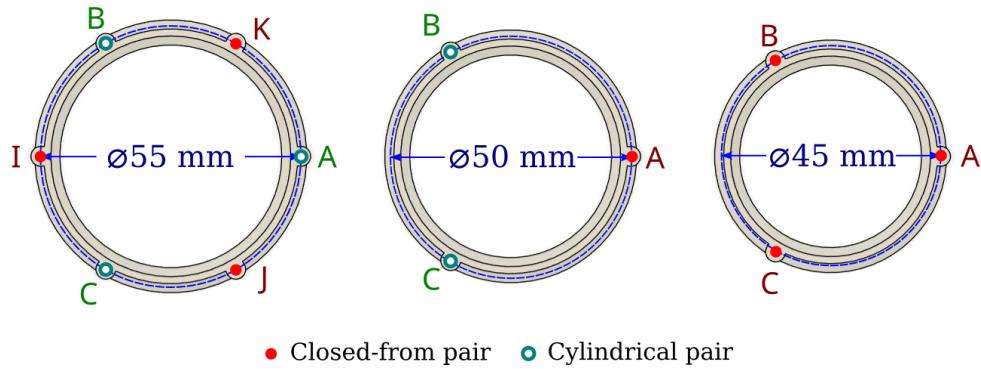
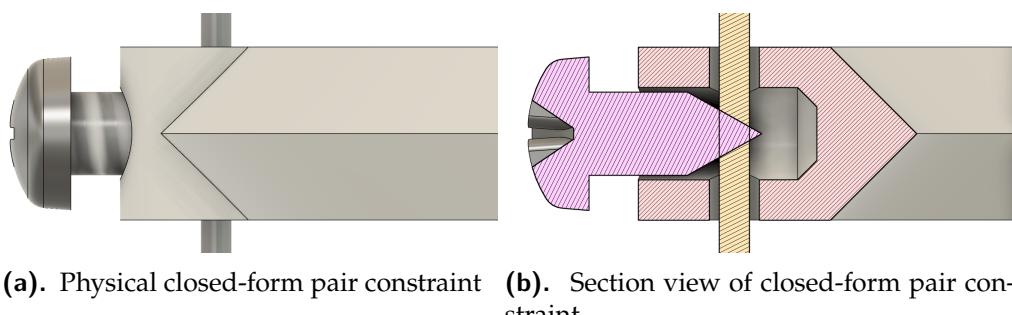


Figure 2.7: Ring constraint sizes

In *blue*, the diameters of the circumferences enclosing the rods are shown. The rods are labeled *A, B, C, I, J, K*. The *green* open circle indicates a cylindrical pair constraint between the ring and the rod, while the *red* indicates a closed-form pair constraint involving both.

In terms of physical construction, a closed-form pair joint is created using a set screw that secures the rod in place. This is illustrated in Figure 2.8. This type of joint provides a firm grip on the rod, preventing any movement. In contrast, a cylindrical-form pair joint is similar in construction but omits the set screw, allowing for both translational and rotational movement along the axis of the hole.



(a). Physical closed-form pair constraint (b). Section view of closed-form pair constraint

Figure 2.8: Closed-form pair constraint

Chapter 3

Linear Actuator

Actuators play a pivotal role in the operation of parallel robots, converting the rotational movement of a motor into linear motion, thus enabling the robot to perform a multitude of tasks. In the context of parallel robots, linear actuators are of particular significance, as they are responsible for the movements needed to control the robot's motion.

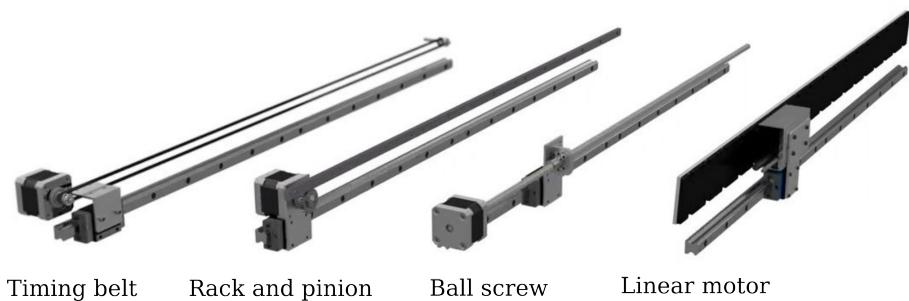


Figure 3.1: Linear motion systems

In parallel robots, linear actuators are used to manipulate the position and orientation of the end-effector. The traditional linear actuators include timing belts, rack and pinion systems, ball screws, and linear motors, as shown in Figure 3.1. However, these actuators often present challenges such as high cost, large size, and fixed or non-modifiable maximum stroke lengths.

The fixed or non-modifiable travel distance of conventional linear actuators represents a major limitation, particularly when the objective is to develop a flexible platform capable of accommodating a wide range of rod-driven robot configurations. Flexibility is a crucial requirement for our platform, as it must be capable of adapting to various setups without requiring significant modifications.

Fortunately, parallel continuum robots offer a broader range of possibilities. In contrast to traditional robots that utilize rigid components, continuum robots employ flexible elements such as rods, which can be shortened or extended to achieve the desired motion. This principle bears resemblance is similar to the operation of 3D printer extruders (see Figure 3.2).

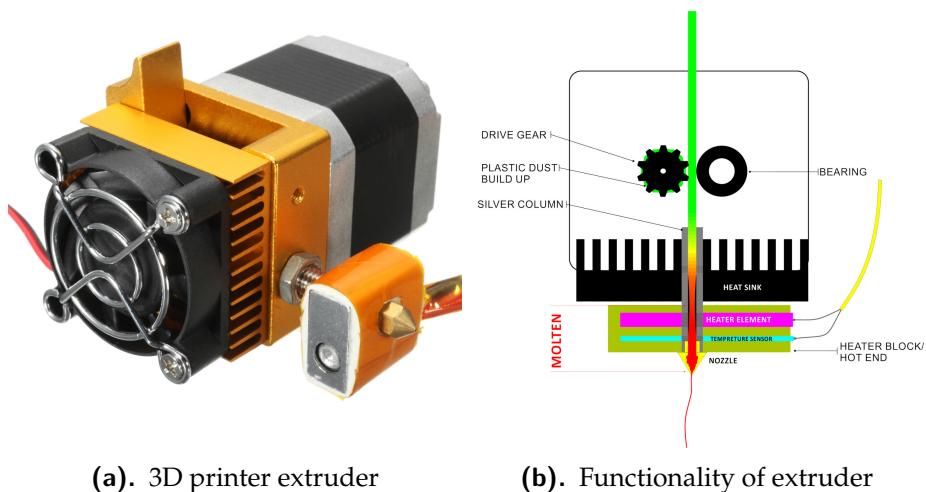


Figure 3.2: 3D printer extruder

The operation of a 3D printer extruder is based in the use of a drive gear and a bearing to push and pull the filament towards the heater. In our application, we are not a priori interested in the heating aspect, but rather in the mechanism that drives the filament. This mechanism can be adapted to move rods in our robot, as shown in Figure 3.3.



Figure 3.3: Extruder without heater

The necessity of a custom design is based on a number of factors. The filaments utilized in standard extruders are thicker than the rods suitable for our robots (> 1.75 mm), as empirical evidence suggests that rods thinner than 1.5 mm are optimal. Thicker rods necessitate greater force to bend, which these actuators are not designed to handle, making this thickness a practical upper limit for our application. Furthermore, the width of the actuators must be considered, along with their weight and speed.

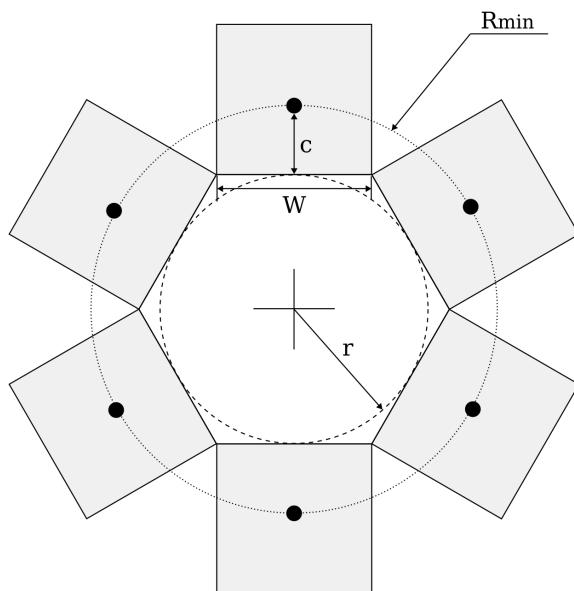


Figure 3.4: Actuators arrange

It is required that each rod be equipped with its own actuator. Consequently, the actuators must be of a size that is compatible with the specified diameter for the base of the robot. The top view of an actuator arrangement is depicted in Figure 3.4. For an arrangement of six actuators with width W and distance to the center c , the relationship with the minimum base radius R_{min} is shown in equations 3.2 and 3.1. An original extruder has a width of $W = 5.2$ cm with $c = 0.7$ cm, which requires a minimum radius of $R_{min} = 5.2$ cm according to equation 3.4. This value is larger than the presented in the model by Wu and Shi [6], and the aim is to design a smaller one. Accordingly, a minimum diameter of $R_{min} = 3.5$ cm with a maximum $c = 0.5$ cm was established, resulting in a maximum width of $W \approx 3.5$ cm, as per equation 3.3.

$$r = R_{min} - c \quad (3.1)$$

$$W = \frac{2}{\sqrt{3}}r \quad (3.2)$$

$$W = \frac{2}{\sqrt{3}}(R_{min} - c) \quad (3.3)$$

$$R_{min} = \frac{\sqrt{3}}{2}W + c \quad (3.4)$$

3.1 Motor

The motor is the central component in actuators, and in extruders, stepper motors are commonly utilized. However, stepper motors were deemed unsuitable for our application due to their size and speed limitations. In this instance, the DC N20 geared motors were selected as they are small and fast enough to meet the requirements. To ensure accurate control of the number of revolutions, it is necessary to employ a motor with an encoder (see Figure 3.5). The dimensions of

the selected motor with encoder are specified in Figure 3.6, and the specifications for the chosen variant are detailed in Table 3.1.



Figure 3.5: Gear motor N20 with encoder

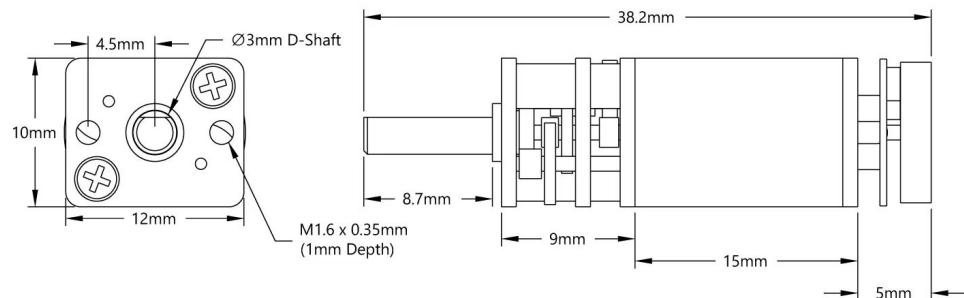


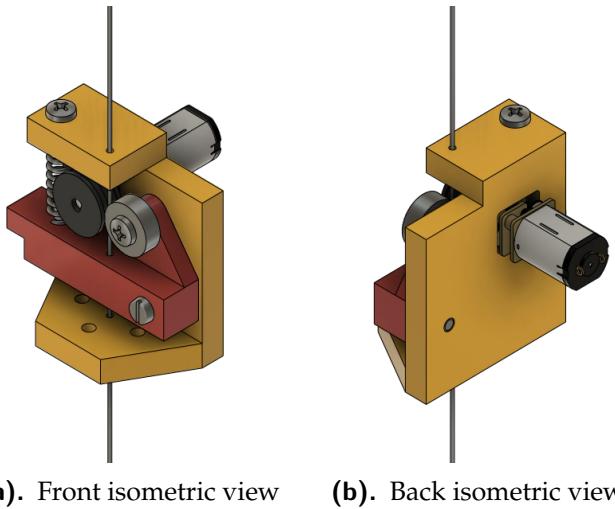
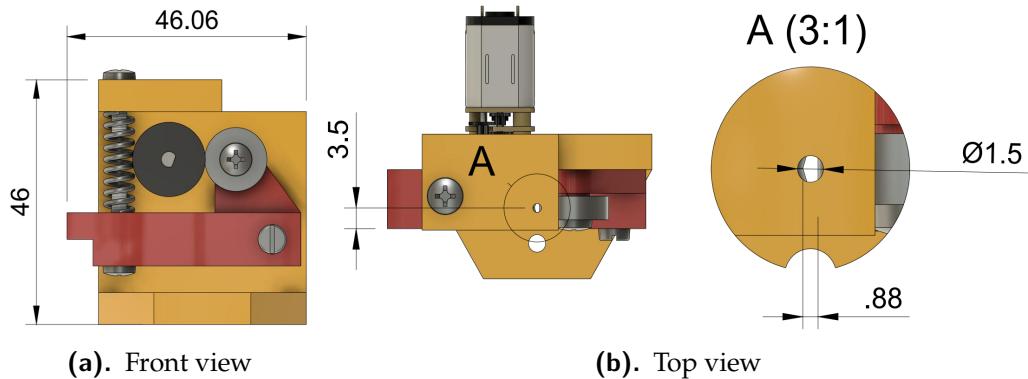
Figure 3.6: Dimensions of gear motor N20 with encoder

Table 3.1: Gear motor specifications

Property	Value
Output shaft style	D-Shaft
Voltage range	6 – 12V
Speed (no load @ 6VDC)	70 rpm
Rated torque	0.65 kg·cm
Stall torque	4 kg·cm
Gear ratio	210:1
Weight	15g
Encoder: cycles per revolution (motor shaft)	3
Encoder sensor type	Magnetic (Hall Effect)
Hall response frequency	100 kHz

3.2 First model

The initial concept of the linear actuator is illustrated in Figure 3.7. This elementary and uncomplicated model was designed to comprehend the fundamental concept of this actuator. The objective was to facilitate the manufacturing process through the use of FDM (fused deposition modeling), specifically 3D printing. The width of this model is 46.06 mm, exceeding the previously stipulated 35 mm. This is shown in Figure 3.8. When attempting to reduce the width of the actuator, limitations were encountered, due to the diameters of the pulley, bearing, spring, and the length of the arm. Additionally, the bearing did not provide sufficient friction, necessitating improvements in this area. Enhancements were also needed for the motor grip and support to ensure ease of printing and overall functionality.

**Figure 3.7:** First model isometric view**Figure 3.8:** First model dimensions

3.3 Final model

Following a series of iterations, the final model was developed and is depicted in Figure 3.9. The final design is more rounded in order to eliminate potential stress concentrators. Moreover, it is also more streamlined and material-efficient, incorporating holes in non-critical areas to reduce material needed in 3D printing.

The viability of manufacturing through this method was a significant factor in the design process. Given our access to 3D printers, this method was the most convenient and expedient solution. The design is conductive to printing when

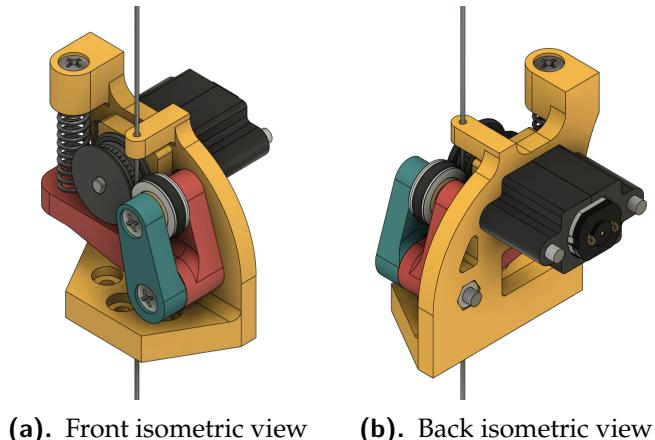


Figure 3.9: Final model isometric view

oriented with the front view facing up on the base. This configuration allows for the nearly complete stacking of the print layers, eliminating the need for support structures. Furthermore, the printing time for the piece is relatively brief, rendering it an optimal choice for rapid prototyping and production.

Although the width remains unchanged, the arm, which previously collided with other actuators in an array, has been repositioned. The arm is now slightly shorter and rounded, as shown in Figure 3.10. This configuration allows the arm to intrude into the space of another actuator without making contact at any point. The design maintains an array of six actuators within the established circle diameter of 7 cm.

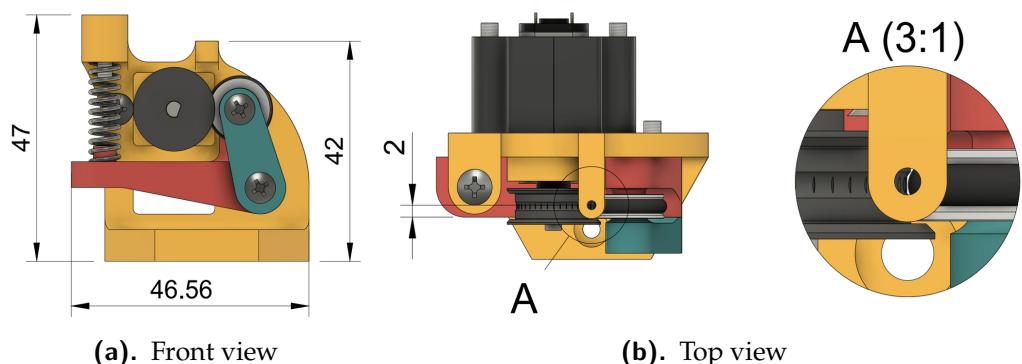


Figure 3.10: Final model dimensions

The issue of friction was also addressed by the addition of an O-ring seal in a

V-shaped bearing and the creation a grooved pattern on the motor pulley. The distance between the pulleys was reduced in order to improve grip. Additionally, a pair of new components was designed to secure the motor, and an additional support was included over the arm and bearing for enhanced rigidity and safety. Critical components such as the spring holder were reinforced, through the addition of material, resulting in a thicker structure.

Figure 3.11 presents an exploded view of the final actuator, with each component labeled with arrows and item numbers in balloons. The corresponding part names for the assembled components are provided in Table 3.2, which also indicates the quantity and source of each part. The sources are classified as 3D printed, commercial, or manufactured. It is noteworthy that the only "manufactured" part is the spring, which was custom-made.

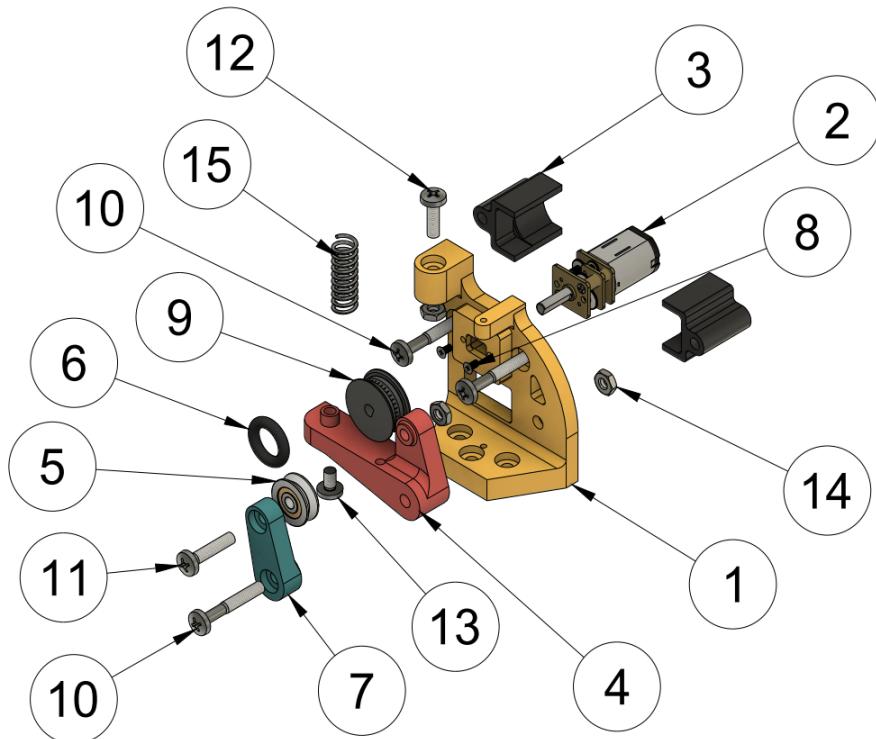
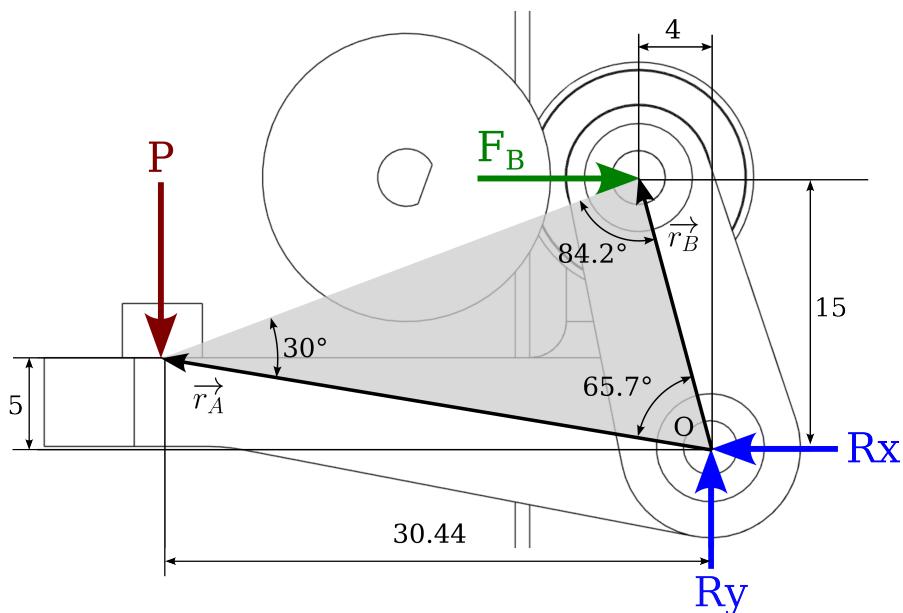


Figure 3.11: Actuator assembly exploded-view

Table 3.2: Actuator assembly parts list

Item	Qty	Part Name / Description	Source
1	1	Base	3D Printed
2	1	Motor GA12 N20 (must be with encoder)	Commercial
3	2	Motor Support	3D Printed
4	1	Arm	3D printed
5	1	V623ZZ Groove Pulley	Commercial
6	1	O-Ring - W2.62 x DI7.59 x DE2.83	Commercial
7	1	Arm Support	3D Printed
8	2	Phillips Countersunk Screw - DIN 965H M1.6x3	Commercial
9	1	Pulley	3D Printed
10	3	Binding Head Screw JIS B 1111 - M3x20	Commercial
11	1	Binding Head Screw JIS B 1111 - M3x14	Commercial
12	1	Binding Head Screw JIS B 1111 - M3x10	Commercial
13	1	Binding Head Screw JIS B 1111 - M3x5	Commercial
14	3	Hexagon Thin Nut DIN 439-2 - M3x0.5	Commercial
15	1	Spring - D5.5x20mm	Manufactured

3.4 Specifications

**Figure 3.12:** Forces in actuator arm

The pressure or grip on the rod by the actuator is provided by the bearing and the actuator arm. The static model of the actuator arm is shown in Figure 3.12.

The sum of forces is found in Equation 3.5, and the sum of moments at point O is given in Equation 3.6. The force P is exerted by the spring, as described by Equation 3.7, where k is the spring constant and Δy is the compression. At the pin O , the components of the reaction force, R_x and R_y , are present. From Equation 3.6, the reaction force F_B at the bearing can be determined, as shown in Equation 3.8.

$$\sum \vec{F} = 0 \quad \therefore \quad \vec{P} + \vec{R}_y + \vec{F}_B + \vec{R}_x = 0 \quad (3.5)$$

$$\sum \vec{M}_O = 0 \quad \therefore \quad \vec{r}_A \times \vec{P} + \vec{r}_B \times \vec{F}_B = 0 \quad (3.6)$$

$$P = -k\Delta y \quad (3.7)$$

$$F_B = \frac{r_{A,x}}{r_{B,y}} P = \frac{30.44}{15} P = -2.0293 k \Delta y \quad (3.8)$$

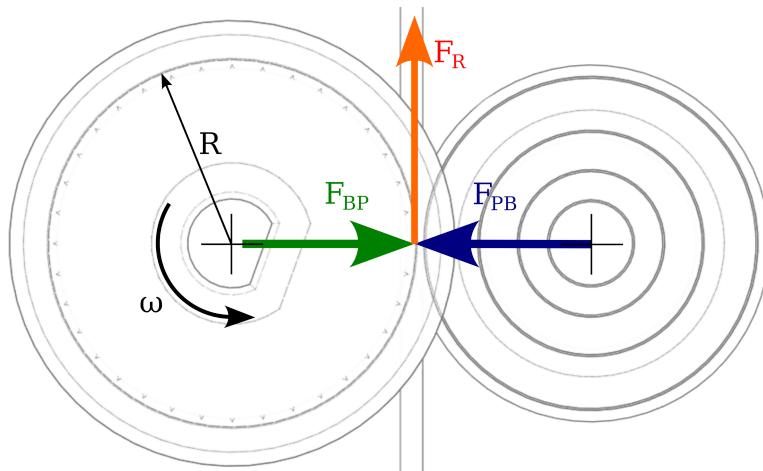


Figure 3.13: Friction force in rod

The force of friction on the rod can be calculated from the force F_B , as illustrated in Figure 3.13 and demonstrated by Equation 3.10. However, these formulas were not subjected to a comprehensive analysis because the static friction factor, represented by the symbol μ_s , were not measured. Instead, geometric con-

straints were of greater importance. Therefore, the spring was selected by empirically testing those with the appropriate stiffness. The drive pulley, with radius R , rotates at speed of ω . Considering the motor's RPM, the actuator's speed specifications are provided in Table 3.3.

$$F_{BP} = F_B \quad (3.9)$$

$$F_R = \mu_s F_{BP} = -2.0293\mu_s k \Delta y \quad (3.10)$$

Table 3.3: Actuator specification

Property	Value
Pulley ratio (R)	5.3 mm
Maximum angular velocity (ω_{max})	7.3304 rad/s
Maximum linear velocity (v_{max})	38.85 mm/s

Chapter 4

Platform Assembly

The assembly of the structure involves the integration of the array of all the actuators, the platform on which they are mounted, and the electronics responsible for controlling the linear actuators. The components of the platform are illustrated in Figure 4.1, and the corresponding parts list is provided in Table 4.1. The structure features a hexagonal in a shape, with includes only three lateral structural supports, made of acrylic, and the upper and lower bases, also made of acrylic. The aforementioned components are connected using 3D printed L-shaped joints. A void is present beneath the actuators, with dimensions to approximately 25 cm, to accommodate the rods.

Table 4.1: Platform assembly parts list

Item	Qty	Part Name / Description	Source
1	1	Actuators Base	3D Printed
2	2	Support Base	Acrylic Laser Cut
3	6	Support Union	Acrylic Laser Cut
4	1	Support Front	Acrylic Laser Cut
5	2	Support Lateral	Acrylic Laser Cut
6	1	Electronics Assembly	
7	4	Binding Head Screw JIS B 1111 - M3x40	Commercial
8	24	Binding Head Screw JIS B 1111 - M3x10	Commercial
9	6	Actuator Assembly	

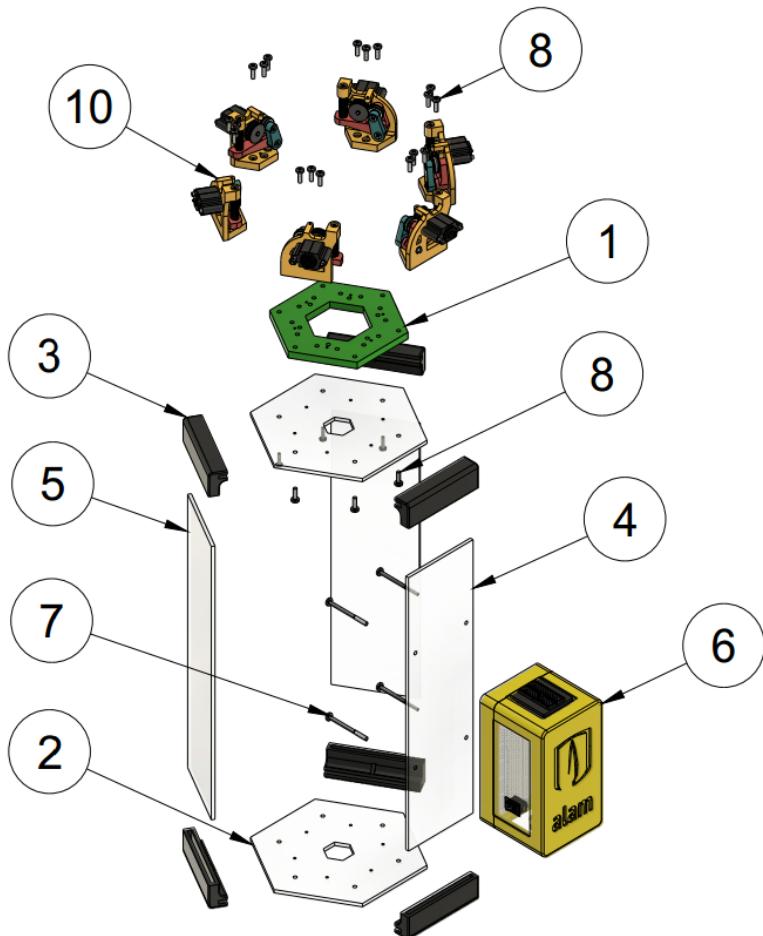


Figure 4.1: Platform assembly exploded-view

4.1 Electronics Case

The electronics case contains all the electronic components that are necessary for the control of the motors. The module is an independent component that can be removed and replaced without disturbing the platform structure. It is conveniently positioned next to the platform for easy access. The case contains the microcontroller unit (MCU), the breadboard on which the motor drivers and various connections are made, as well as the input and output peripherals of the module. These can be seen in Figure 4.2 and are listed in Table 4.2. The housing is 3D printed and consists of two main parts: one containing the MCU and the other containing the breadboard. The peripherals are affixed to the walls, which are constructed from laser-cut acrylic sheets.

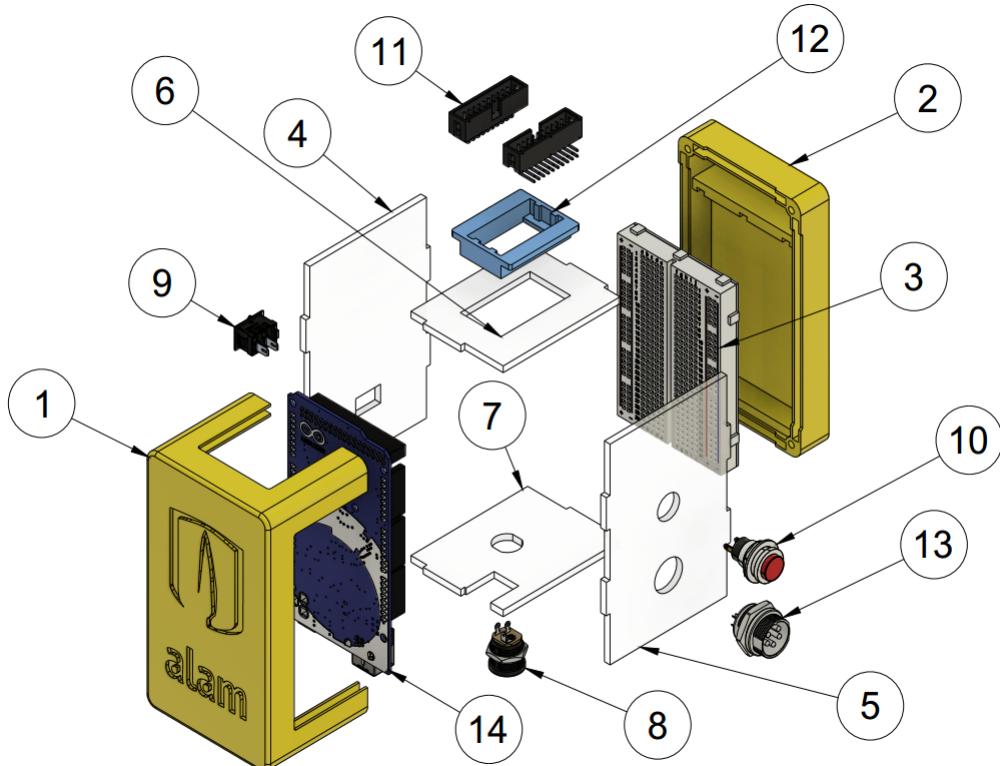


Figure 4.2: Electronics assembly exploded-view

Table 4.2: Electronics assembly parts list

Item	Qty	Part Name / Description	Source
1	1	Arduino Case	3D Printed
2	1	Breadboard Base	3D Printed
3	1	Half-Size Breadboard	Commercial
4	1	Left Wall	Acrylic Laser Cut
5	1	Right Wall	Acrylic Laser Cut
6	1	Top Wall	Acrylic Laser Cut
7	1	Bottom Wall	Acrylic Laser Cut
8	1	DC Power Input Jack - DS-223B	Commercial
9	1	Power Switch - KCD1-11-2P	Commercial
10	1	Red Push Button - DS 212	Commercial
11	2	IDC 3020-20-0200-00 - 20P 2.54MM	Commercial
12	1	IDC Support	3D Printed
13	1	MX M12 5-Pin Male MIC Connector Plug	Commercial
14	1	Arduino MEGA 2650	Commercial

Figure 4.3 illustrates the ports and connections of the case. The lower section comprises a serial port for programming the MCU and sending commands, as

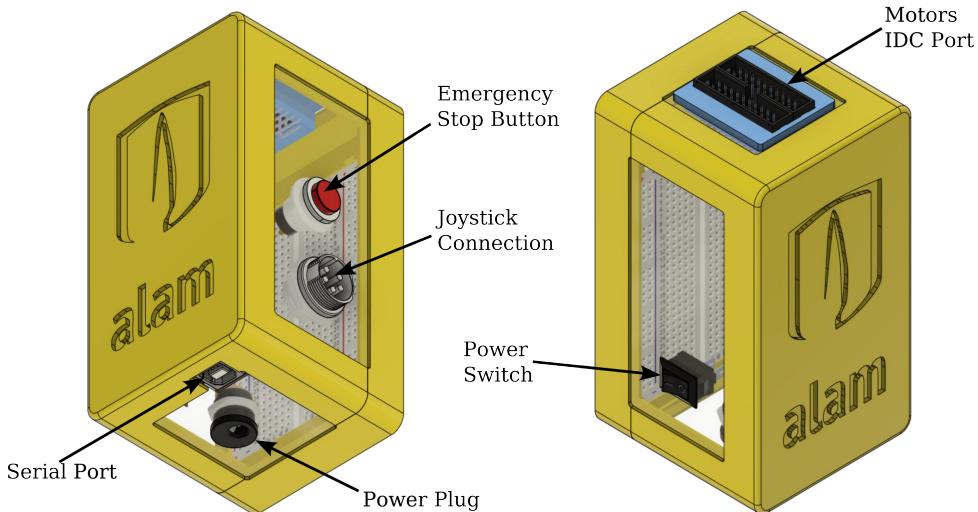


Figure 4.3: Ports and connections

well as a power plug. The right side features an emergency button to prevent the robot from moving, and a 5-pin connector input for connecting a joystick. The left side houses the power button, while the upper section includes a pair of 20-pin IDC inputs for connecting the unit to the motors.

4.1.1 Circuit Schematic

The schematic of the circuit connections is illustrated in Figure 4.4. As shown in Figure 4.5, each actuator motor necessitates six connections. The pins designated as M1 and M2 are responsible for supplying power to the motors, while the pins designated as VCC and GND provide power to the magnetic encoder. The pins A and B are utilized for quadrature output.

In order to accurately count the encoder pulses in both directions, it is necessary to connect one pin to an interrupt pin and the other to a digital pin. The motor driver, designated as DRV8833, is employed to regulate the polarity and speed of the motors. This device necessitates two PWM connections for each motor. In view of the necessity for a cost-effective MCU with an adequate number of pins to connect six motors, the Arduino MEGA 2650 was selected for this project.

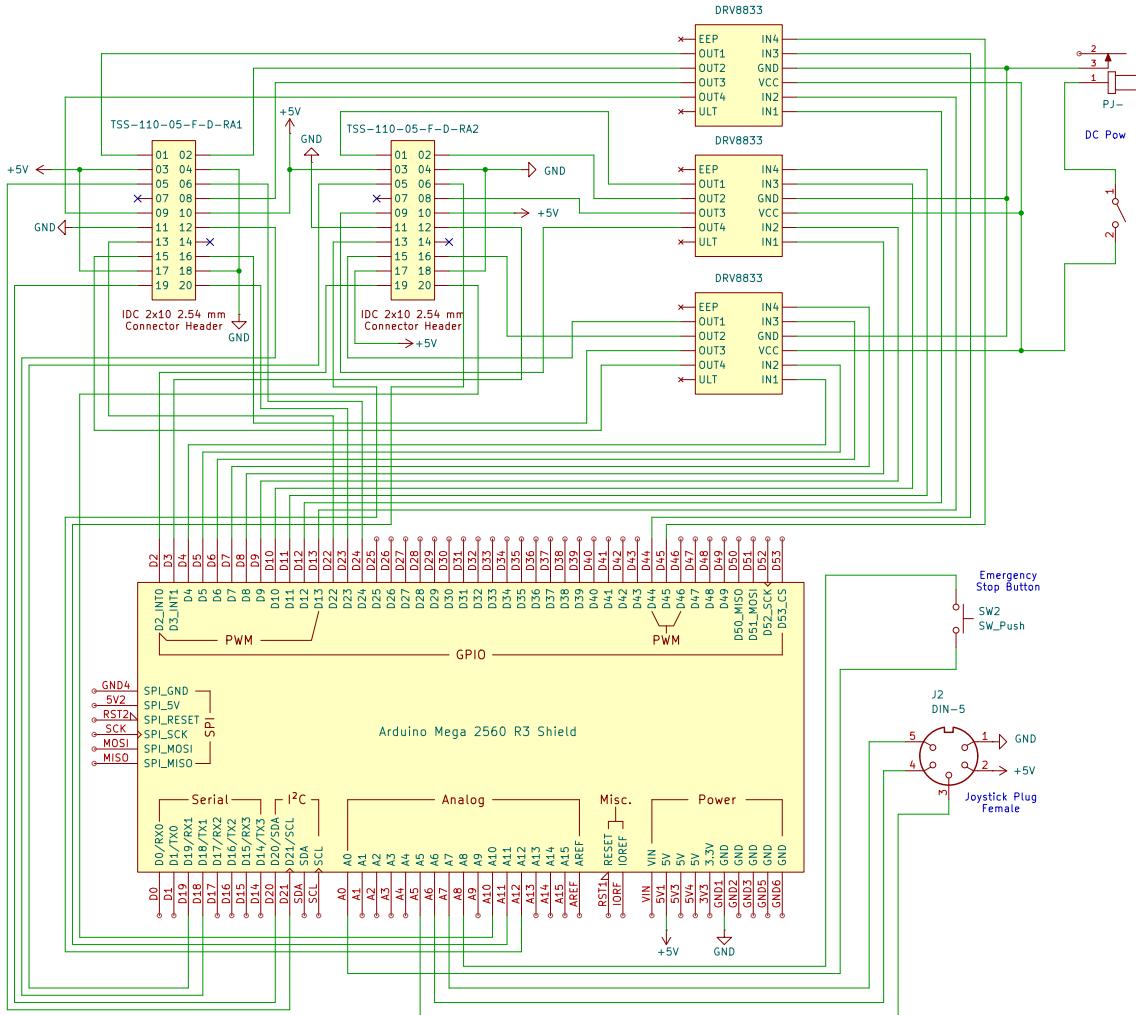


Figure 4.4: Main circuit schematic

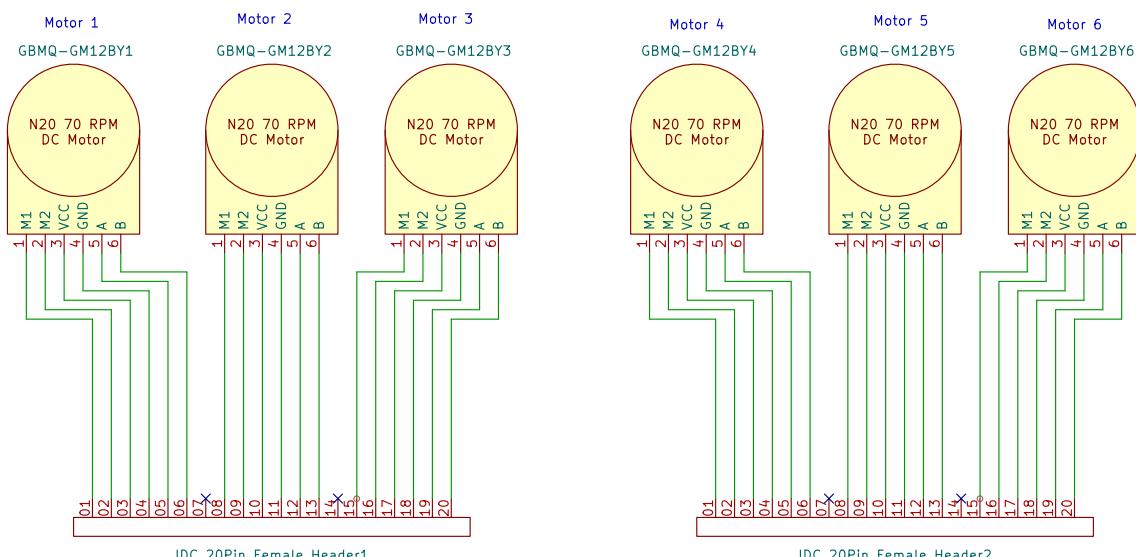


Figure 4.5: Motor circuit schematic

Chapter 5

Control System Interface

One of the project's objectives was to devise a protocol for controlling the robot. Although the robot has an input for a joystick, this is insufficient for controlling the robot in a three-dimensional space. Furthermore, the platform is designed to accommodate any user requirement, necessitating the development of a protocol or language for programming the robot.

The protocol will be incorporated into the controller firmware. The commands will be transmitted in real-time by a device, currently a computer, which serves as the interface for receiving user inputs. For this reason, a command-line application has also been developed to send commands via the established port. The source code for this project, Alam¹, can be found on GitHub².

The development of the robot's kinematic model was beyond the scope of this project due to the added complexity. This remains an area for future exploration. Consequently, the control system is direct, specifying the distance or position of each actuator individually and manually. However, the intention is for this pro-

¹This is the name of the project. It is derived from the Spanish word "*alambre*", which translates to rod or wire.

²<https://www.github.com/liconaj/GraduationProject-Alam>

tocol to be compatible with such a model in the future.

5.1 Protocol Specifications

The G-Code is a standard language that is widely utilized in the manufacturing and computer numerical control (CNC) machining industries to control machine tools, including mills and lathes. It is composed of alphanumeric commands that specify coordinates, speeds, and machine functions. These commands facilitate precise control and automation in manufacturing processes. In addition to G-Code, M-Code commands are employed to control additional machine functions, such as tool changes and device operations.

Table 5.1: G-Code definitions

G-Code	Definition
G1	Move one or more actuators to specified positions at a given optional speed.
G4	Pause or dwell. Halts the machine for the specified duration.
G28	Move actuators to origin position.
G28.1	Move actuators to target position.
G90	Set absolute positioning mode. All subsequent movements are interpreted as absolute positions. This is the default mode.
G91	Set relative positioning mode. All subsequent movements are interpreted as displacements from the current position.
G92	Set origin position.
G92.1	Set target position. This may be a position of interest.

The adoption of G-Code as the foundation for our robot control protocol offers efficiency and clarity, rendering it an ideal choice for rapid command transmission to microcontrollers such as the Arduino. Its hierarchical structure and diverse commands provide the flexibility needed for motion control and complex

operations, aligning well with the platform's adaptability goals. The detailed definitions of the proposed G-Code and M-Code commands are presented in Tables 5.1 and 5.2, respectively. These tables serve as a clear reference for the implementation and usage of the commands.

Table 5.2: M-Code definitions

M-Code	Definition
M00	Stop program
M85	Maximum command wait time.
M114	Report current position of the actuators.
M301	Set PID parameters.
M303	Initiate autotuning process for the PID parameters.
M355	Report current status of robot parameters.
M500	Save parameters to EEPROM.
M501	Load parameters from EEPROM.
M502	Reset parameters saved in EEPROM to defaults.
M503	Display the saved configuration in EEPROM.

The application of these commands and their respective parameters is elucidated in Tables 5.4 and 5.3. Each table includes illustrative examples that demonstrate the practical implementation of the aforementioned commands. Additionally, Table 5.5 provides a comprehensive explication of the parameters associated with each command.

Table 5.3: M-Codes usage

Note that only commands with parameters are explained. All others can be used by simply entering the code.

Code	Usage	Example
M85	M85 P<millis,5000> S<secs,5>	> M85 S3
M301	M301 [P<kp,0>] [I<ki,0>] [D<kd,0>]	> M301 P1000 I0.5
M500	M500 <optn>	> M500 T P
M502	M502 <optn>	> M502 I J K

Table 5.4: G-Codes usage

Code	Usage	Example
G1	G1 <id><pos> [F<speed,255>]	> G1 A100.0
G4	G4 P<millis> S<secs>	> G4 P3000
G20	G20	> G20
G21	G21	> G21
G28	G28 [<id,A B C I J K>]	> G28 A B C
G28	G28.1 [<id,A B C I J K>]	> G28.1 J K
G90	G90	> G90
G91	G91	> G91
G92	G92 [<id,A B C I J K><pos,0>]	> G92 A-100.0 I5.0
G92.1	G92.1 [<id, A B C I J K><pos,0>]	> G92.1 B250.0

Table 5.5: Code parameters

Parameter	Meaning
<id>	Actuator identification letter: A, B, C, I, J or K.
<pos>	Position number.
<kp>	Proportional constant.
<ki>	Integral constant.
<kd>	Derivative constant.
<speed>	Maximum speed of actuator as integer number from 50 to 255.
<millis>	Milliseconds as integer number.
<secs>	Seconds as integer number.
<optn>	Parameter of the robot. T for PID tuning constants; A, B, C, I, J or K for position of given actuator; P for all actuator positions
<...,VALUE>	Default value.
[...]	Optional parameter. If omitted, default value will be used.

5.2 Command Line Application

The robot's firmware was developed in Arduino/C++ using the Arduino library and the *SerialCommands* library³ for command definitions. Communication occurs via a serial data bus, as suggested by the name of the library. The command-line application, written in Python, employs the *pyserial* library⁴ to establish a connection with the Arduino.

5.2.1 Architecture

Figure 5.1 depicts the dependency relationships between the firmware source components and the packages used. Furthermore, a flowchart diagram illustrating the interaction between the command-line application and the robot is included in Figure 5.2.

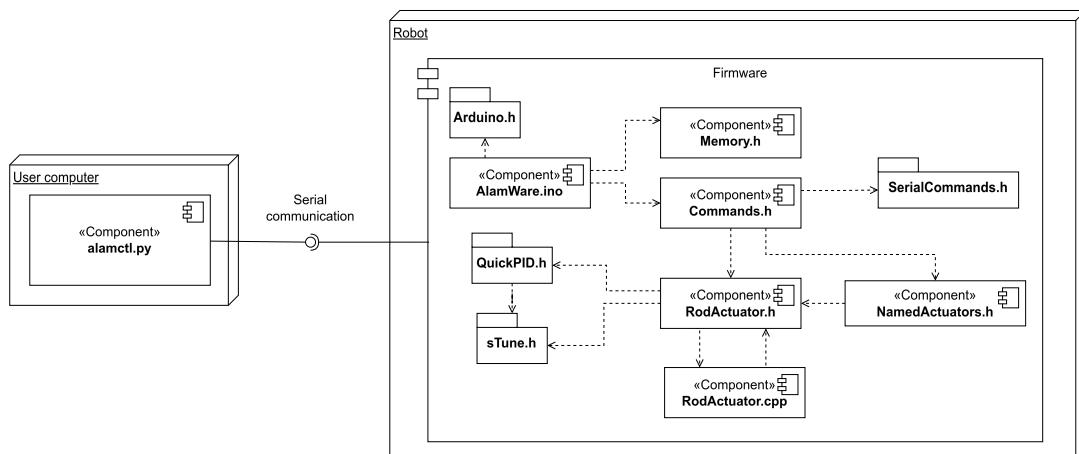


Figure 5.1: Relationships of control system components

The serial port data buffer is 128 bytes, which allows for a maximum of 128 characters per command. Only a single command can be sent at a time, separated by a newline. Following the execution of a command, the robot sends a message indicating completion. The robot is programmed with a configurable maximum

³<https://github.com/ppedro74/Arduino-SerialCommands>

⁴<https://pypi.org/project/pyserial/>

timeout, which is employed to handle commands that take an excessive amount of time. It will not interrupt or modify its current task until a new instruction is received. The robot's firmware does not support comments, and thus, these are removed by the application's parser to send only essential instructions to the robot.

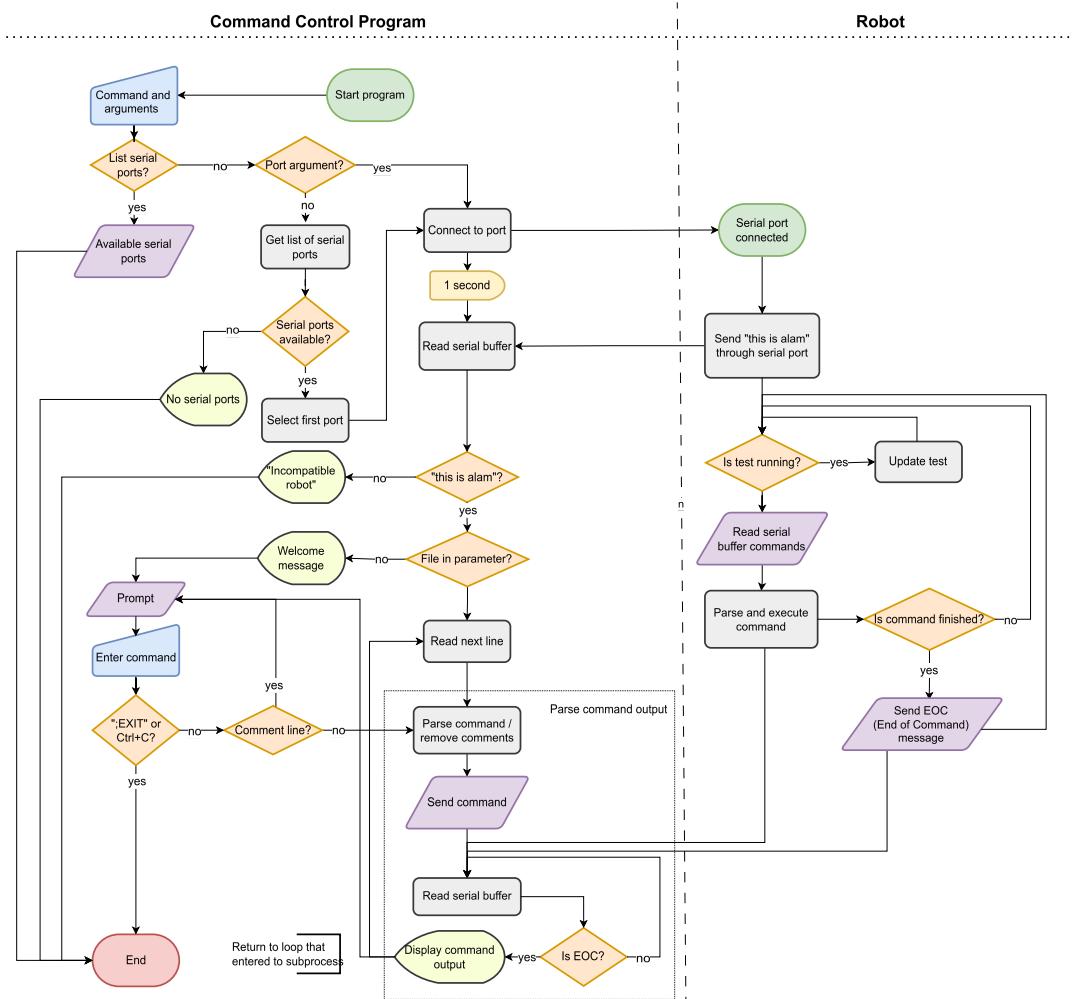
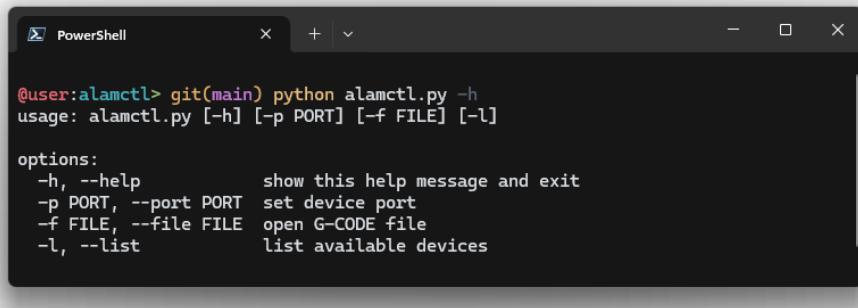


Figure 5.2: System flowchart

5.2.2 Usage

In order to control the robot, it is necessary to utilize the `alamctl.py` script. The code for this script can be found in Appendix D. Figure 5.3 shows how to use the application via the help parameter output.

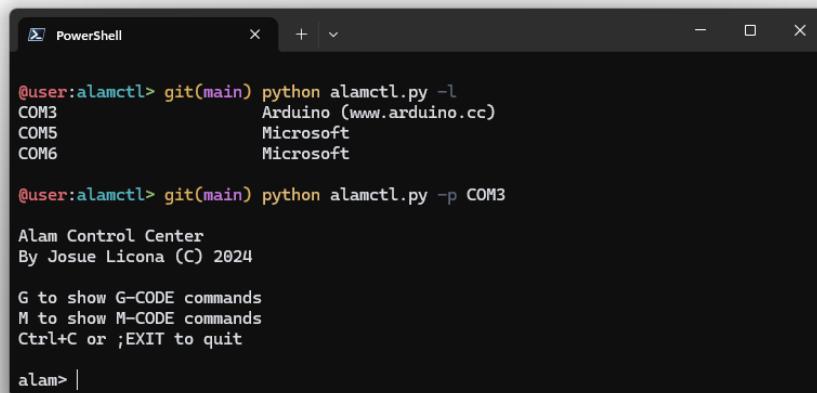


```
@user:alamctl> git(main) python alamctl.py -h
usage: alamctl.py [-h] [-p PORT] [-f FILE] [-l]

options:
  -h, --help            show this help message and exit
  -p PORT, --port PORT set device port
  -f FILE, --file FILE  open G-CODE file
  -l, --list             list available devices
```

Figure 5.3: Command line usage

It should be noted that all parameters are optional. If no parameters are passed, as explained in the flowchart in Figure 5.2, the application will attempt to establish a connection to one of the available ports. However, the recommended method is to initially search for available ports and then attempt to connect to the device whose description most closely aligns with that of the controller, as shown in Figure 5.4.



```
@user:alamctl> git(main) python alamctl.py -l
COM3          Arduino (www.arduino.cc)
COM5          Microsoft
COM6          Microsoft

@user:alamctl> git(main) python alamctl.py -p COM3
Alam Control Center
By Josue Licona (C) 2024

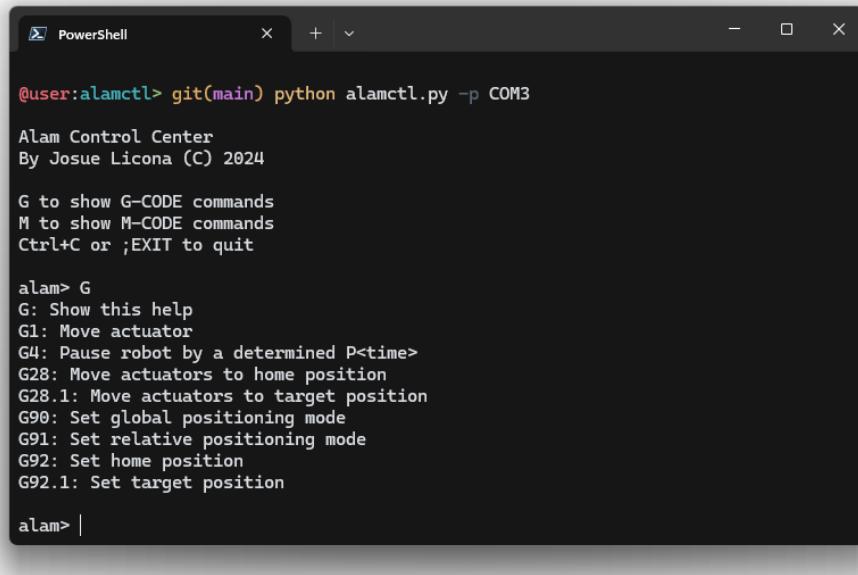
G to show G-CODE commands
M to show M-CODE commands
Ctrl+C or ;EXIT to quit

alam> |
```

Figure 5.4: Command line connection

Once a successful connection with the robot has been established, the application displays a welcome message and a brief instructional guide to facilitate the user's interaction with the robot (see Figure 5.5). However, it is also possible to

pass a G-Code script as a parameter for the program to send commands automatically (Figure 5.6). This is the most efficient and powerful method for users to control a robot.



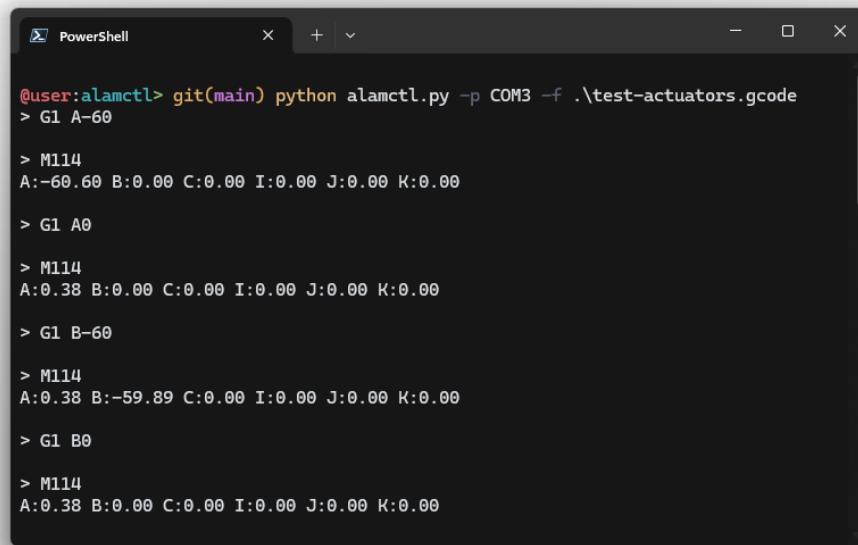
```
@user:alamctl> git(main) python alamctl.py -p COM3
Alam Control Center
By Josue Licona (C) 2024

G to show G-CODE commands
M to show M-CODE commands
Ctrl+C or ;EXIT to quit

alam> G
G: Show this help
G1: Move actuator
G4: Pause robot by a determined P<time>
G28: Move actuators to home position
G28.1: Move actuators to target position
G90: Set global positioning mode
G91: Set relative positioning mode
G92: Set home position
G92.1: Set target position

alam> |
```

Figure 5.5: Command line manual input



```
@user:alamctl> git(main) python alamctl.py -p COM3 -f .\test-actuators.gcode
> G1 A-60
> M114
A:-60.60 B:0.00 C:0.00 I:0.00 J:0.00 K:0.00
> G1 A0
> M114
A:0.38 B:0.00 C:0.00 I:0.00 J:0.00 K:0.00
> G1 B-60
> M114
A:0.38 B:-59.89 C:0.00 I:0.00 J:0.00 K:0.00
> G1 B0
> M114
A:0.38 B:0.00 C:0.00 I:0.00 J:0.00 K:0.00
```

Figure 5.6: Command line file parsing

5.3 PID Control and Tuning

Given that the actuators are equipped with DC motors, the controllable variable is the speed. However, the position can be determined thanks to the magnetic encoder, which counts the number of motor revolutions. Based on this, a PID control is implemented, wherein the position is utilized as the input variable, thereby enabling the controller to manipulate the speed in order to reach the desired position. To implement this control, the *QuickPID*⁵ library was employed, which can be integrated with the *sTune* library to determine the actuator tunings automatically. An attempt was made to use this latter library to find the tunings; however, its methods are designed for systems with an S-shaped response to a constant input, which does not apply to our system. Therefore, the outcomes yielded by this library were frequently unsatisfactory. For this reason, a more sophisticated tool, MATLAB and Simulink, was employed.

5.3.1 Test Data

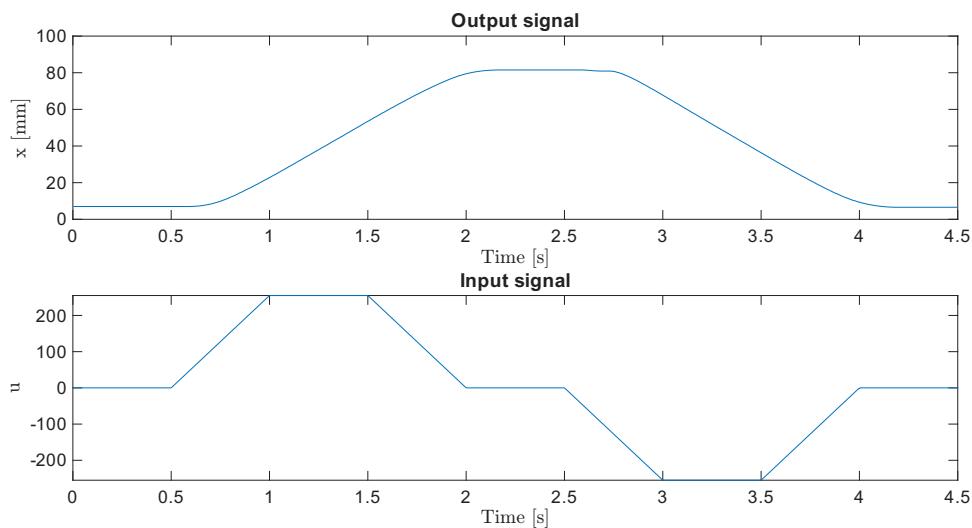


Figure 5.7: Input and output signals

The initial step involved the implementation of a test to obtain data from the actuators, with the objective to gaining a comprehensive understanding of the

⁵<https://github.com/Dlloyddev/QuickPID>

system. The durations of this test is 4.5 seconds, and it can be initiated via the command M300. It sends a PWM signal to the actuators that varies over time. The output of the system is the linear position of the actuator at each time instant, as seen in Figure 5.7.

5.3.2 System Identification

The system's transfer function is identified based on the obtained data using MATLAB's *System Identification Toolbox*. The process carried out is shown in Figure 5.8. The model fit to the real data is depicted in Figure 5.9. The MATLAB code utilized is provided in Appendix B.

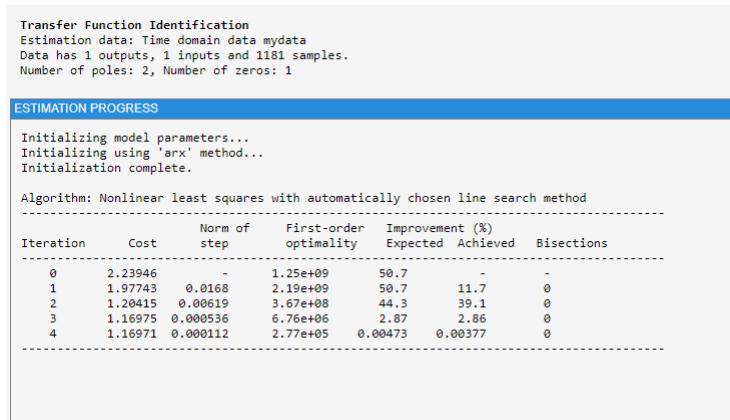


Figure 5.8: Transfer function identification in MATLAB

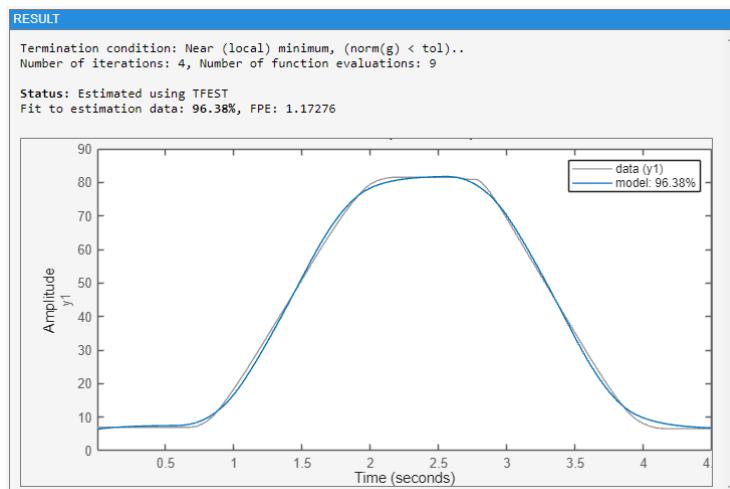


Figure 5.9: Transfer function model in MATLAB

The transfer equation of the linear actuator that was identified is shown in Equation 5.1, where the variable of the function is z^{-1} .

$$tf = \frac{(4.808 - 4.596z^{-1}) \times 10^{-4}}{1 - 1.982z^{-1} + 0.9824z^{-2}} \quad (5.1)$$

5.3.3 PID Tuning

The tunings of the PID controller were determined using the MATLAB utility *pidTuner*. The PI controller, which is a PID controller with $K_d = 0$, exhibited the most optimal response. In this instance, the control is discrete, and the resulting constants are presented in Table 5.6. These variables are utilized to implement the controller with the *QuickPID* library in Arduino.

$$C = K_p + K_i * \frac{T_s}{z - 1} \quad (5.2)$$

Table 5.6: PID tunings

Property	Value
Proportional constant, K_p	21.4
Integrative constant, K_i	1.97
Sample time, T_s	0.00381s

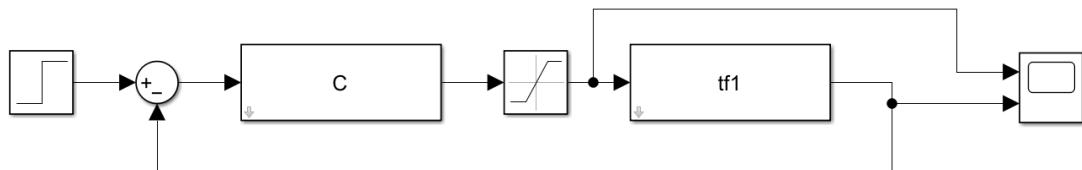


Figure 5.10: Blocks diagram of control system

A general block diagram of the control system is shown in Figure 5.10, where C represents the transfer function of the PID controller (equation 5.2) and $tf1$ rep-

resents the transfer function of the actuator. The system's response to a 100 mm input is shown in Figure 5.11. The blue curve indicates that the system responds promptly, reaching the desired value in approximately two seconds, theoretically. However, it exhibits overshoot and takes nearly four seconds to settle into a steady state. In many cases, it is not necessary for the robot to reach exact intermediate positions when it is in motion. Consequently, the firmware was programmed to permit the input of new commands once the desired position has been reached, thus compromising accuracy for speed.

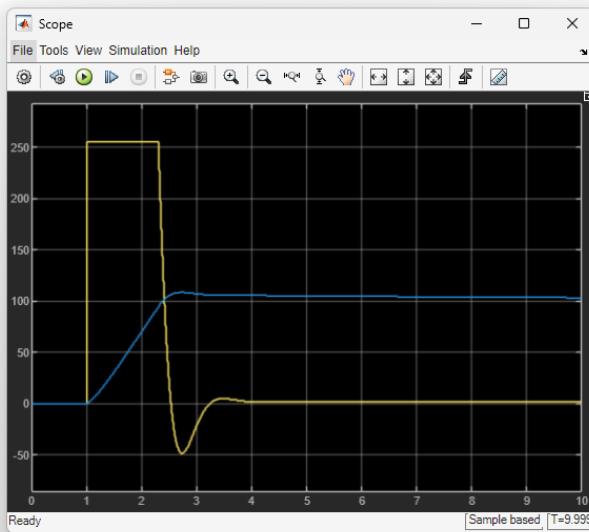


Figure 5.11: System response of simulation in Simulink

Moreover, the control signal, depicted in yellow, exhibits saturation, underscoring the necessity to implement an anti-windup control. Fortunately, the *QuickPID* library offers a function for this, rendering implementation straightforward. This feature was duly applied.

Chapter 6

Performance and Results

The construction of the Alam robot platform was successfully completed using steel rod segments, as depicted in Figure 6.1. The constraints are securely fastened to the rods, thereby ensuring stability and structural integrity.

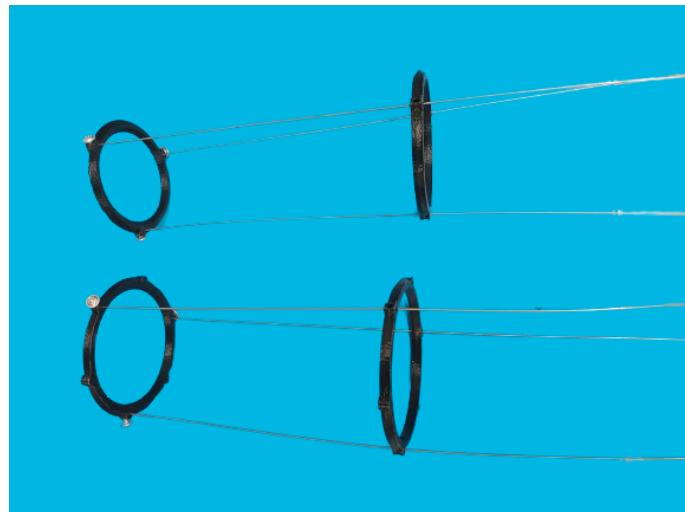


Figure 6.1: Arm segments

Figure 6.2 illustrates the robot in three different poses using a single segment. The first pose shows the arm to the left, the second a straight configuration, and the third with the arm to the right. The actuators exhibited sufficient friction and

force to maintain each position effectively.

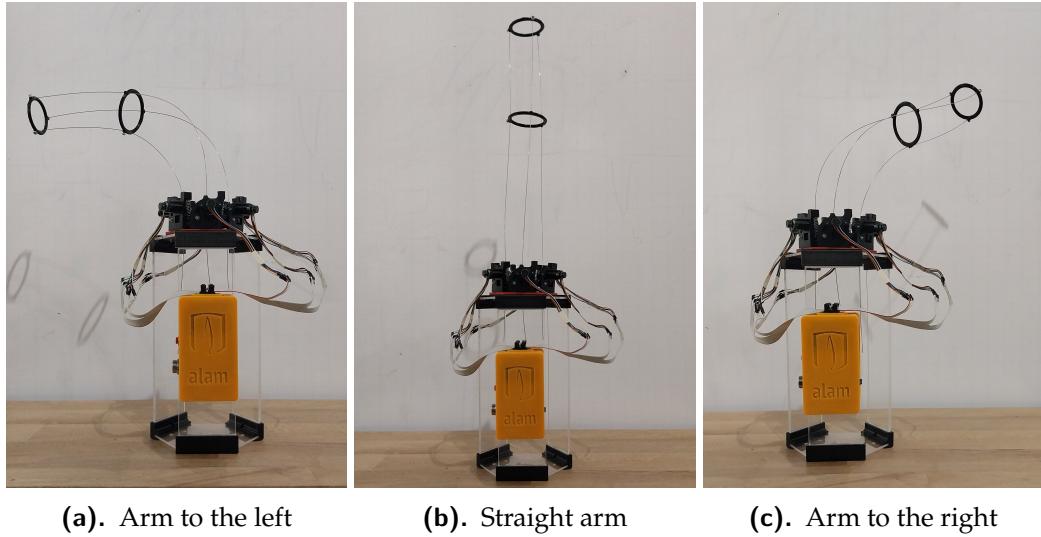


Figure 6.2: One segment arm positions

The firmware and control application demonstrated satisfactory functionality during testing. However, the firmware remains incomplete, particularly with regard to its ability to handle commands with erroneous parameters. The majority of the code functionalities were tested, but certain aspects related to the EEPROM memory and robot status display were not yet fully implemented. Furthermore, the functionality of the joystick control remains unimplemented.



Figure 6.3: Electronics case and connections

The electronics case, shown in Figure 6.3, was initially prototypical, utilizing a breadboard and jumper wires for connections. This configuration was found to be cumbersome and prone to disconnection, which made the assembly and debugging processes more complex.

Figure 6.4 compares the prototype with two segments against the initial design. The prototype closely resembles the design, thereby demonstrating the successful realization of the intended structure.

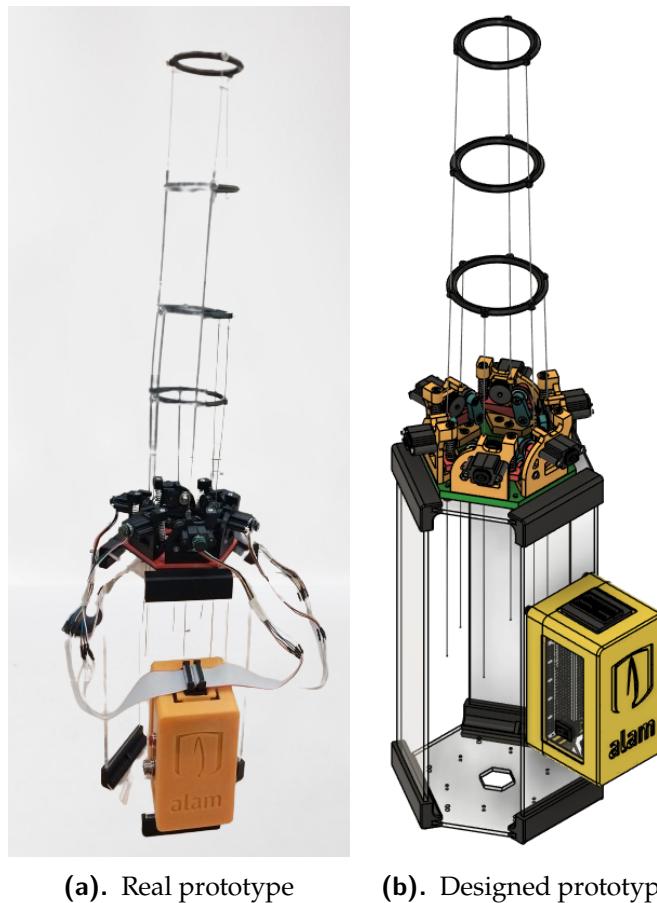
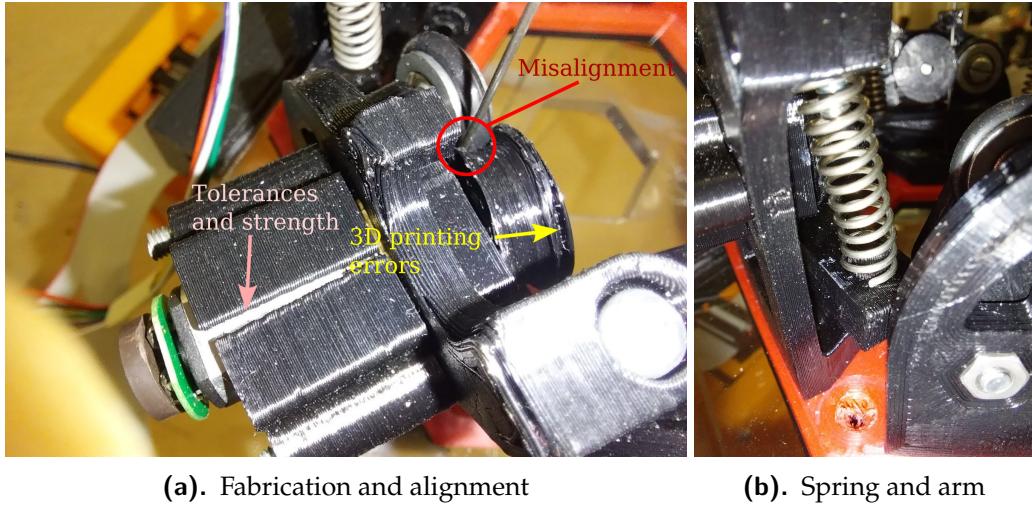


Figure 6.4: Complete prototype

Issues encountered with the linear actuators are shown in Figure 6.5. These include discrepancies in tolerance, alignment challenges between rod and bearing, printing errors, and base deformation due to spring force, which has occasionally resulted in detachment of the arm from its support.



(a). Fabrication and alignment

(b). Spring and arm

Figure 6.5: Linear actuator issues

Finally, Table 6.1 presents general specifications of the platform, including height, maximum arm reach, body dimensions, and actuator tolerances.

Table 6.1: Platform specifications

Property	Value
Height	30 cm
Maximum height (with arm)	70 cm
Weight	1 kg
Linear actuators tolerance	± 1 mm

Conclusions

The development of the Alam robot platform has demonstrated the successful integration of mechanical design, control theory, and software development throughout this project. The platform, constructed from steel rod segments and 3D-printed components, provides a robust framework for robotic experimentation and control. Notwithstanding the difficulties encountered in the completion and the electronic assembly, the platform succeeded in attaining its principal objective of offering a flexible and programmable robotic system.

The testing and experimentation phase revealed both the platform's achievements and areas for improvement. The implementation of PID control algorithms demonstrated effective position control; however, issues with overshoot and controller saturation necessitated further refinement. The difficulties encountered with linear actuators highlighted the necessity for precision manufacturing and iterative design enhancements. These insights will inform future iterations aimed at optimizing performance and reliability.

The scalability of the platform was a fundamental aspect of the overarching objective. The design and architecture provide a robust foundation for the expansion of functionality in both software and hardware. The modular nature of the system allows for straightforward incorporation of supplementary sensors, actuators, and peripherals, thereby facilitating enhancements and customization based

on specific requirements. The firmware was designed in such a way that the new commands and features can be added with a minimum of reconfiguration. Moreover, the use of standardized communication protocols guarantees compatibility with a multitude of external devices and systems. This inherent flexibility positions the platform for future developments and applications, thereby rendering it a versatile and scalable solution for various robotic projects.

In summary, the Alam robot platform serves as a foundational tool for the exploration of advanced robotics concepts and applications. By addressing challenges in mechanical design, control algorithms, and software integration, this project sets a precedent for future developments in the field of robotics.

Future Work and Recommendations

- Develop a PCB for the electronics to mitigate potential issues and eliminate current problems. Currently, connecting IDC 2 with the power off results in the motor drivers drawing power from the Arduino, which poses a risk of damage to the MCU and the computer. The underlying cause of this issue remains elusive.
- When establishing serial communication with the Arduino, the LED linked to pin D13 blinks twice. This indicates that the power to the motors must be off or that another PWM pin must be utilized when connecting the module.
- It is recommended that the joystick functionality be implemented for direct control. The objective of this feature was to facilitate the simultaneous adjustment of all actuators, which would in turn assist with the insertion and removal of rods.
- Complete the implementation of missing command functionalities in the control language.
- Enhance the design of actuators to streamline assembly. The current placement of screws securing actuators to the base is inaccessible and cumbersome.
- Address challenges with spring installation and adjustability. Utilize math-

ematical models or adopt extruder-like systems to regulate spring tension effectively.

- Refine model tolerances to improve alignment of the rod with bearings and drive pulleys. Explore alternative materials or manufacturing methods to rectify identified defects in the actuator design for 3D printing.
- Enhance portability by transitioning the system from mains power to battery operation.
- Develop a static or kinetic model of the robot using analytical methods or machine learning algorithms, leveraging the platform and robot for practical applications.

Bibliography

- [1] Z. Shao, D. Zhang, and S. Caro, "New frontiers in parallel robots," *Machines*, vol. 11, no. 3, 2023, issn: 2075-1702. doi: 10.3390/machines11030386.
- [2] D. Stewart, "A platform with six degrees of freedom," *Proceedings of the Institution of Mechanical Engineers*, vol. 180, no. 1, pp. 371–386, 1965. doi: 10.1243/PIME PROC_1965_180_029_02.
- [3] E. News. "The delta robot – swiss-made and fastest in the world!" (), [Online]. Available: <https://actu.epfl.ch/news/the-delta-robot-swiss-made-and-fastest-in-the-world/> (visited on 06/14/2024).
- [4] S. Briot and J. Burgner-Kahrs, "Editorial: New frontiers in parallel robotics," *Frontiers in Robotics and AI*, vol. 10, 2023, issn: 2296-9144. doi: 10.3389/frobt.2023.1282798.
- [5] M. Russo, S. M. H. Sadati, X. Dong, et al., "Continuum robots: An overview," *Advanced Intelligent Systems*, vol. 5, no. 5, p. 2200367, 2023. doi: <https://doi.org/10.1002/aisy.202200367>.
- [6] G. Wu and G. Shi, "Design, modeling, and workspace analysis of an extensible rod-driven parallel continuum robot," *Mechanism and Machine Theory*, vol. 172, p. 104798, 2022, issn: 0094-114X. doi: <https://doi.org/10.1016/j.mechmachtheory.2022.104798>.
- [7] A. L. Orekhov, V. A. Aloi, and D. C. Rucker, "Modeling parallel continuum robots with general intermediate constraints," in *2017 IEEE International Conference on Mechatronic Sciences, Engineering and Technology (IMSET)*, 2017, pp. 1–6. doi: 10.1109/IMSET50000.2017.8250001.

ference on Robotics and Automation (ICRA), 2017, pp. 6142–6149. doi: 10.1109/ICRA.2017.7989728.

Appendices

A Forces in Actuator Arm

```

syms Fb P rAx rAy rBx rBy

vecFb = [Fb 0 0];
vecP = [0 P 0];
rA = [rAx rAy 0];
rB = [rBx rBy 0];

eq = cross(rA, vecP) + cross(rB, vecFb);
solve(eq, Fb)

```

B PID Tuning Code

```

% Get data
dataTable = readtable("datos.csv");
t = dataTable.t/1000; %convert time from milliseconds to seconds
PWM = dataTable.PWM;
A = dataTable.A;
startTime = t(1);
sampleTime = mean(diff(t));
mydata = iddata(A,PWM,sampleTime, "Tstart", startTime);

% Find transfer function of system
npoles = 2; nzeros = 1;
Options = tfestOptions;
Options.Display = 'on';
Options.EnforceStability = true;
tf1 = tfest(mydata, npoles, nzeros, Options, 'Ts', sampleTime, ...
    'Feedthrough', true);

% PID Tune
opts = pidtuneOptions("PhaseMargin", 90);
[C, info] = pidtune(tf1, 'PI', 5, opts);

```

C Test Code

```
; ALAM ROBOT TEST
;BY LICONAJ (c) 2024

G4 P5000 ; WAIT 3 SECONDS
;MOVE ACTUATOR A
G1 A-60
G4 P500
G1 AO
G4 P500
;MOVE ACTUATOR B
G1 B-60
G4 P500
G1 BO
G4 P500
;MOVE ACTUATOR C
G1 C-60
G1 CO
;MOVE ACTUATORS A AND B
G4 P1000
G1 A-60 B-60
G4 P500
G1 AO BO
;MOVE ACTUATORS B AND C
G4 P1000
G1 B-60 C-60
G4 P500
G1 BO CO
;MOVE ACTUATORS C AND A
G4 P1000
G1 C-60 A-60
G4 P500
G1 CO AO
;MOVE ALL ACTUATORS
G4 P1000
G1 A-60 B-60 C-60
G4 P1000
;MOVE ACTUATORS A B C TO HOME
G28 A B C
```

D Command Line Interface Source Code

```
#!/usr/bin/python3
#alamctl.py

import serial
import sys
import argparse
import time
import serial.tools.list_ports

EOC = "!EOC"
EOL = "\r\n"

def get_ports():
    ports = []
    patterns = ["COM", "ttyACM", "ttyUSB"]
```

```

for pattern in patterns:
    ports += serial.tools.list_ports.grep(pattern)
return ports

def list_ports():
    ports = get_ports()
    for port in ports:
        print("{0:<20} \t {1:<30}".format(port.device, port.manufacturer))

parser = argparse.ArgumentParser()
parser.add_argument("-p", "--port", help="set device port")
parser.add_argument("-f", "--file", help="open G-CODE file")
parser.add_argument("-l", "--list", help="list available devices", action="store_true")
args = parser.parse_args()

if args.list:
    list_ports()
    sys.exit()

if args.port:
    port = args.port
else:
    ports = get_ports()
    print(f"{len(ports)} device(s) found")
    if not ports:
        print("No serial ports available")
        sys.exit()
    else:
        port = ports[0].device
        print(f"Connecting to {port}...")

try:
    mcu = serial.Serial(port=port, baudrate=250000, timeout=.1)
    time.sleep(1)
    response = mcu.readline().decode()
    if not response.startswith("this is alam"):
        print("Incompatible robot")
        sys.exit()
except serial.SerialException:
    print("Serial port unavailable")
    sys.exit()

def send_command(value: str):
    mcu.write(bytes(value, "utf-8"))

def parse_command(cmd: str) -> str:
    cmd = cmd.strip().split(";")[0]
    cmd += EOL
    # print(f"Parsing command {repr(cmd)}")
    send_command(cmd)
    output = ""
    value = ""
    while not EOC in value:
        value = mcu.readline().decode()
        if not EOC in value:
            output += value
    return output

if __name__ == "__main__":
    if args.file is None:
        print("\nAlam Control Center")
        print("By Josue Licona (C) 2024\n")

        print("G to show G-CODE commands")
        print("M to show M-CODE commands")
        print("Ctrl+C or ;EXIT to quit\n")

        try:
            while True:

```

```

        cmd = input("alam> ").upper()
        if ";EXIT" in cmd:
            break
        elif not cmd or cmd.startswith(";"):
            continue
        print(parse_command(cmd))
    except KeyboardInterrupt:
        print("\n\nPROGRAM ENDED BY USER\n")

else:
    time.sleep(1)
    try:
        with open(args.file, "r", encoding="utf-8") as fh:
            lines = fh.readlines()
            fh.close()
    except FileNotFoundError:
        print(f"File '{args.file}' not found")
        sys.exit()

    for cmd in lines:
        cmd = cmd.strip().upper()
        if not cmd or cmd.startswith(";"):
            continue
        print(f"> {cmd.strip()}")
        print(parse_command(cmd))

```

E Firmware Source Code

E.1 Arduino Main Entry Point

```

// AlamWare.ino

#include "Memory.h"
#include "Commands.h"

// SerialCommands Setup
char serialCommandBuffer[128];
SerialCommands serialCommands(&Serial, serialCommandBuffer, sizeof(serialCommandBuffer));

void addSerialCommands() {
    serialCommands.SetDefaultHandler(cmdUnrecognized);

    serialCommands.AddCommand(&GHelp);
    serialCommands.AddCommand(&G1);
    serialCommands.AddCommand(&G4);
    serialCommands.AddCommand(&G92);
    serialCommands.AddCommand(&G92_1);
    serialCommands.AddCommand(&G28);
    serialCommands.AddCommand(&G28_1);
    serialCommands.AddCommand(&G90);
    serialCommands.AddCommand(&G91);

    serialCommands.AddCommand(&MHelp);
    serialCommands.AddCommand(&M00);
    serialCommands.AddCommand(&M85);
    serialCommands.AddCommand(&M114);
    serialCommands.AddCommand(&M303);
    serialCommands.AddCommand(&M301);
    serialCommands.AddCommand(&M300);
}

void setup() {

```

```

    Serial.begin(250000);
    Serial.print("this is alam");
    addSerialCommands();
}

void loop() {
    if (testing) {
        updateTest();
    } else {
        serialCommands.ReadSerial();

        updateNamedActuators();
        nextCommand();
    }
}

```

E.2 Persistence Memory Management Class Header

```

// Memory.h

#ifndef MEMORY_H
#define MEMORY_H

#include <EEPROM.h>

class Memory {
private:
    float kp;
    float kd;
    float ki;

    uint8_t kpAddress = 10;
    uint8_t kiAddress = 12;
    uint8_t kdAddress = 14;

    int lastPositions[6];
    uint8_t pAddresses[6];

public:
    Memory();
    ~Memory();

    int GetLastPosition(uint8_t actuatorId);
    void SavePosition(uint8_t actuatorId, int position);
    void GetTunings(float* kp, float* ki, float* kd);
    void SaveTunings(float kp, float ki, float kd);
};

#endif

```

E.3 Persistence Memory Management Class

```

// Memory.cpp
#include "Memory.h"

Memory::Memory() {
    for (int i = 0; i < 6; i++) {
        pAddresses[i] = 20 + sizeof(float) * i;
    }
}

```

```

Memory::~Memory() {
}

void Memory::SavePosition(uint8_t actuatorId, int position) {
    EEPROM.put(pAddresses[actuatorId], position);
}

int Memory::GetLastPosition(uint8_t actuatorId) {
    return lastPositions[actuatorId];
}

void Memory::SaveTunings(float kp, float ki, float kd) {
    EEPROM.put(kpAddress, kp);
    EEPROM.put(kiAddress, ki);
    EEPROM.put(kdAddress, kd);
}

void Memory::GetTunings(float* kp, float* ki, float* kd) {
    EEPROM.get(kpAddress, kp);
    EEPROM.get(kiAddress, ki);
    EEPROM.get(kdAddress, kd);
}

```

E.4 Commands Definitions

```

// Commands.h
#ifndef COMMANDS_H
#define COMMANDS_H

#include "SerialCommands.h"
#include "NamedActuators.h"
#include "RodActuator.h"

bool sendingCommand = false;
bool testing = false;
unsigned long startTimeTest;

void updateTest();
void nextCommand();

// G-Code Definitions
void cmdUnrecognized(SerialCommands *, const char *);
void cmdGetGCodes(SerialCommands *);
void cmdDelay(SerialCommands *);
void cmdMoveActuator(SerialCommands *);
void cmdSetGlobalMode(SerialCommands *);
void cmdSetRelativeMode(SerialCommands *);
void cmdSetHome(SerialCommands *);
void cmdSetTarget(SerialCommands *);
void cmdMoveToHome(SerialCommands *);
void cmdMoveToTarget(SerialCommands *);

// M-Code Definitions
void cmdGetMCodes(SerialCommands *);
void cmdStop(SerialCommands *);
void cmdsetTimeout(SerialCommands *);
void cmdGetActuatorPositions(SerialCommands *);
void cmdTestActuators(SerialCommands *);
void cmdStartAutoTuning(SerialCommands *);
void cmdSetPIDConstants(SerialCommands *);

SerialCommand GHelp("G", &cmdGetGCodes);
SerialCommand G4("G4", &cmdDelay);
SerialCommand G1("G1", &cmdMoveActuator);
SerialCommand G28("G28", &cmdMoveToHome);
SerialCommand G28_1("G28.1", &cmdMoveToTarget);
SerialCommand G90("G90", &cmdSetGlobalMode);
SerialCommand G91("G91", &cmdSetRelativeMode);
SerialCommand G92("G92", &cmdSetHome);
SerialCommand G92_1("G92.1", &cmdSetTarget);

SerialCommand MHelp("M", &cmdGetMCodes);
SerialCommand M00("M00", &cmdStop);
SerialCommand M85("M85", &cmdSetTimeout);

```

```

SerialCommand M114("M114", &cmdGetActuatorPositions);
SerialCommand M300("M300", &cmdTestActuators);
SerialCommand M303("M303", &cmdStartAutoTuning);
SerialCommand M301("M301", &cmdSetPIDConstants);

void endCmd(SerialCommands *sender) {
    sender->GetSerial()->println("!EOC");
}

void nextCommand() {
    if (!sendingCommand)
        return;
    for (NamedActuator actuator : namedActuators) {
        if (!actuator.object->Next())
            return;
    }
    Serial.println("!EOC");
    sendingCommand = false;
}

float getvel(int t) {
    float v;
    float m = 255.0 / 500.0;
    if (t < 500 || t > 4000) {
        v = 0;
    } else if (t < 1000) { // entre 500 y 1000 ms rampa
        v = m * (t - 500);
    } else if (t < 1500) { // mantener velocidad máxima por 500 ms
        v = 255;
    } else if (t < 2000) { // disminuir velocidad
        v = -m * (t - 1500) + 255;
    } else if (t < 2500) { // mantener velocidad máxima en reversa por 500 ms
        v = 0;
    } else if (t < 3000) {
        v = -m * (t - 2500);
    } else if (t < 3500) {
        v = -255;
    } else if (t < 4000) {
        v = m * (t - 3500) - 255;
    }
    return v;
}

void updateTest() {
    unsigned long t = millis() - startTimeTest;
    Serial.print(t);
    float v = getvel(t);
    Serial.print(";");
    Serial.print(v);
    for (NamedActuator actuator : namedActuators) {
        Serial.print(";");
        actuator.object->TestVelocity(v);
        actuator.object->Update();
        Serial.print(actuator.object->GetPosition());
    }
    Serial.println("");
    if (t >= 4500) {
        testing = false;
        for (NamedActuator actuator : namedActuators) {
            actuator.object->Stop();
        }
        Serial.println("!EOC");
    }
}

void cmdUnrecognized(SerialCommands *sender, const char *cmd) {
    sendingCommand = true;
    sender->GetSerial()->print("Unrecognized command [");
    sender->GetSerial()->print(cmd);
    sender->GetSerial()->println("]");
}

void cmdGetGCodes(SerialCommands *sender) {
    sendingCommand = true;
    sender->GetSerial()->println("G: Show this help");
    sender->GetSerial()->println("G1: Move actuator");
    sender->GetSerial()->println("G4: Pause robot by a determined P<time>");
    sender->GetSerial()->println("G28: Move actuators to home position");
    sender->GetSerial()->println("G28.1: Move actuators to target position");
    sender->GetSerial()->println("G90: Set global positioning mode");
    sender->GetSerial()->println("G91: Set relative positioning mode");
    sender->GetSerial()->println("G92: Set home position");
    sender->GetSerial()->println("G92.1: Set target position");
}

void cmdGetMCodes(SerialCommands *sender) {
}

```

```

sendingCommand = true;

sender->GetSerial()->println("M: Show this help");
sender->GetSerial()->println("M00: Stop actuators");
sender->GetSerial()->println("M85: Set command timeout with P<milliseconds> or S<seconds>");
sender->GetSerial()->println("M114: Report actual position of actuators");
sender->GetSerial()->println("M300: Test actuators");
sender->GetSerial()->println("M301: Set PID constants with P<value> I<value> D<value>");
sender->GetSerial()->println("M303: Starts autotuning");
}

void cmdDelay(SerialCommands *sender) {
    sendingCommand = true;

    // sender->GetSerial()->println("Starting delay");
    char *arg = sender->Next();
    if (arg != NULL) {
        char parameter = arg[0];
        float value = atof(arg + 1);
        if (parameter == 'P') {
            delay(value);
        }
    }
}

void cmdMoveActuator(SerialCommands *sender) {
    sendingCommand = true;
    uint8_t speed = 255;
    char *arg;
    while ((arg = sender->Next()) != NULL) {
        char id = arg[0];
        float value = atof(arg + 1);
        if (id == 'F') {
            speed = value;
            continue;
        }
        RodActuator *actuator = getActuatorByName(id);
        actuator->Move(value);
        actuator->SetMaxSpeed(speed);
    }
}

void cmdStop(SerialCommands *sender) {
    sendingCommand = true;
    for (NamedActuator actuator : namedActuators) {
        actuator.object->Stop();
    }
}

void cmdSetTimeout(SerialCommands *sender) {
    sendingCommand = true;
    float maximumTimeout = 60e3;
    float timeout = 5000;
    char *arg = sender->Next();
    if (arg != NULL) {
        char parameter = arg[0];
        float value = atof(arg + 1);
        if (parameter == 'P') {
            timeout = value;
        } else if (parameter == 'S') {
            timeout = value * 1000;
        }
        if (timeout > maximumTimeout)
            timeout = maximumTimeout;
        for (NamedActuator actuator : namedActuators) {
            actuator.object->SetTimeout(timeout);
        }
    } else {
        timeout = namedActuators[0].object->GetTimeout();
    }
    sender->GetSerial()->print("Timeout:");
    sender->GetSerial()->println(timeout);
}

void cmdGetActuatorPositions(SerialCommands *sender) {
    sendingCommand = true;

    int decimalPlaces = 5;

    String positions = "";
    for (int i = 0; i < N_ACTUATORS; ++i) {
        String name = (String)namedActuators[i].name;
        float position = namedActuators[i].object->GetPosition();
        positions += name + ":" + String(position);
        if (i < N_ACTUATORS - 1) {
            {
                positions += " ";
            }
        }
    }
}

```

```

        }
        sender->GetSerial()->println(positions);
    }

void cmdStartAutoTuning(SerialCommands *sender) {
    sendingCommand = true;

    char *arg;
    while ((arg = sender->Next()) != NULL) {
        char id = arg[0];
        Serial.print("Starting auto tuning in actuator ");
        Serial.println(id);
        RodActuator *actuator = getActuatorByName(id);
        actuator->AutoTune();
    }
}

void cmdTestActuators(SerialCommands *sender) {
    testing = true;
    startTimeTest = millis();
    sender->GetSerial()->print("t");
    sender->GetSerial()->print(",PWM");
    for (NamedActuator actuator : namedActuators) {
        sender->GetSerial()->print(";");
        sender->GetSerial()->print(actuator.name);
    }
    sender->GetSerial()->println("");
}

void cmdSetPIDConstants(SerialCommands *sender) {
    sendingCommand = true;

    float kp = 0;
    float ki = 0;
    float kd = 0;

    char *arg;
    while ((arg = sender->Next()) != NULL) {
        char id = arg[0];
        float value = atof(arg + 1);
        if (id == 'P')
            kp = value;
        else if (id == 'I')
            ki = value;
        else if (id == 'D')
            kd = value;
    }

    for (NamedActuator actuator : namedActuators) {
        actuator.object->SetPIDConstants(kp, ki, kd);
    }

    sender->GetSerial()->print("Kp:");
    sender->GetSerial()->print(kp);
    sender->GetSerial()->print(" Kd:");
    sender->GetSerial()->print(kd);
    sender->GetSerial()->print(" Ki:");
    sender->GetSerial()->println(ki);
}

void cmdSetGlobalMode(SerialCommands *sender) {
    for (NamedActuator actuator : namedActuators) {
        actuator.object->SetMoveMode(actuator.object->Global);
    }
}

void cmdSetRelativeMode(SerialCommands *sender) {
    for (NamedActuator actuator : namedActuators) {
        actuator.object->SetMoveMode(actuator.object->Relative);
    }
}

void cmdSetHome(SerialCommands *sender) {
    sendingCommand = true;
    char *arg;
    while ((arg = sender->Next()) != NULL) {
        char id = arg[0];
        float value = atof(arg + 1);
        RodActuator *actuator = getActuatorByName(id);
        actuator->SetHomePosition(value);
    }
}

void cmdSetTarget(SerialCommands *sender) {
    sendingCommand = true;
    char *arg;
    while ((arg = sender->Next()) != NULL) {
        char id = arg[0];
        float value = atof(arg + 1);
    }
}

```

```

    RodActuator *actuator = getActuatorByName(id);
    actuator->SetTargetPosition(value);
}

void cmdMoveToHome(SerialCommands *sender) {
    sendingCommand = true;
    char *arg;
    arg = sender->Next();
    if (arg == NULL) {
        for (NamedActuator actuator : namedActuators) {
            actuator.object->MoveToHome();
        }
    } else {
        char id = arg[0];
        float value = atof(arg + 1);
        RodActuator *actuator = getActuatorByName(id);
        actuator->MoveToHome();

        while ((arg = sender->Next()) != NULL) {
            id = arg[0];
            float value = atof(arg + 1);
            actuator = getActuatorByName(id);
            actuator->MoveToHome();
        }
    }
}

void cmdMoveToTarget(SerialCommands *sender) {
    sendingCommand = true;
    char *arg;
    arg = sender->Next();
    if (arg == NULL) {
        for (NamedActuator actuator : namedActuators) {
            actuator.object->MoveToTarget();
        }
    } else {
        char id = arg[0];
        float value = atof(arg + 1);
        RodActuator *actuator = getActuatorByName(id);
        actuator->MoveToTarget();

        while ((arg = sender->Next()) != NULL) {
            id = arg[0];
            float value = atof(arg + 1);
            actuator = getActuatorByName(id);
            actuator->MoveToTarget();
        }
    }
}

#endif

```

E.5 Actuators Instances

```

// NamedActuators.h
#ifndef NAMEDACTUATORS_H
#define NAMEDACTUATORS_H

#include "RodActuator.h"

// 1.1
#define A_ENA 21
#define A_ENB 24
#define A_IN1 13
#define A_IN2 12
// 1.3
#define B_ENA 20
#define B_ENB 23
#define B_IN1 6
#define B_IN2 7
// 2.2
#define C_ENA 3
#define C_ENB 66
#define C_IN1 10
#define C_IN2 11
// 2.1
#define I_ENA 19
#define I_ENB 65
#define I_IN1 9
#define I_IN2 8
// 2.3
#define J_ENA 2
#define J_ENB 64
#define J_IN1 5
#define J_IN2 4

```

```

// 1.2
#define K_ENA 18
#define K_ENB 22
#define K_IN1 44
#define K_IN2 45

enum Actuators {
    A, B, C, I, J, K,
    N_ACTUATORS
};

struct NamedActuator {
    char name;
    RodActuator *object;
};

// Posiciones
volatile long angPosition[N_ACTUATORS] = {0, 0, 0, 0, 0, 0};

// Interrupt Handlers
// static void A_interruptHandler() {
//     if (digitalRead(A_ENA) != digitalRead(A_ENB))
//         angPosition[A]++;
//     else
//         angPosition[A]--;
// }
static void B_interruptHandler() {
    if (digitalRead(B_ENA) != digitalRead(B_ENB))
        angPosition[B]++;
    else
        angPosition[B]--;
}
static void C_interruptHandler() {
    if (digitalRead(C_ENA) != digitalRead(C_ENB))
        angPosition[C]++;
    else
        angPosition[C]--;
}
static void I_interruptHandler() {
    if (digitalRead(I_ENA) != digitalRead(I_ENB))
        angPosition[I]++;
    else
        angPosition[I]--;
}
static void J_interruptHandler() {
    if (digitalRead(J_ENA) != digitalRead(J_ENB))
        angPosition[J]++;
    else
        angPosition[J]--;
}
static void K_interruptHandler() {
    if (digitalRead(K_ENA) != digitalRead(K_ENB))
        angPosition[K]++;
    else
        angPosition[K]--;
}

NamedActuator namedActuators[N_ACTUATORS] = {
    {'A', new RodActuator(A_ENA, A_ENB, A_IN1, A_IN2, angPosition[A], A_interruptHandler)},
    {'B', new RodActuator(B_ENA, B_ENB, B_IN1, B_IN2, angPosition[B], B_interruptHandler)},
    {'C', new RodActuator(C_ENA, C_ENB, C_IN1, C_IN2, angPosition[C], C_interruptHandler)},
    {'I', new RodActuator(I_ENA, I_ENB, I_IN1, I_IN2, angPosition[I], I_interruptHandler)},
    {'J', new RodActuator(J_ENA, J_ENB, J_IN1, J_IN2, angPosition[J], J_interruptHandler)},
    {'K', new RodActuator(K_ENA, K_ENB, K_IN1, K_IN2, angPosition[K], K_interruptHandler)},
};

RodActuator *getActuatorByName(char name) {
    for (int i = 0; i < N_ACTUATORS; ++i) {
        if (namedActuators[i].name == name)
            return namedActuators[i].object;
    }
    return nullptr;
}

void updateNamedActuators() {
    for (NamedActuator actuator : namedActuators)
        actuator.object->Update();
}

#endif

```

E.6 Actuator Class Header

```
//RodActuator.h
#ifndef RODACTUATOR_H
#define RODACTUATOR_H

#include <Arduino.h>
#include <QuickPID.h>
#include <sTune.h>

class RodActuator {
public:
    RodActuator(uint8_t ENA, uint8_t ENB, uint8_t IN1, uint8_t IN2,
                volatile long &angPosition, void (*interruptHandler)());
    enum MoveMode {
        Global,
        Relative
    };
    void Stop();
    void Update();
    void SetPIDConstants(float Kp, float Ki, float Kd);
    void Move(float desiredPosition);
    float GetPosition();
    int GetTimeout();
    void TestVelocity(float velocity);
    void SetLeftLimit(float leftLimit);
    void SetRightLimit(float rightLimit);
    void SetMaxSpeed(uint8_t maxSpeed);
    void SetMoveMode(MoveMode);
    void SetTimeout(float timeout);
    void MoveToHome();
    void MoveToTarget();
    void SetHomePosition(float newHomePosition);
    void SetTargetPosition(float newSecondHomePosition);
    float GetHomePosition();
    float GetTargetPosition();
    void AutoTune();
    bool Next();

private:
    float homePosition = 0;
    float targetPosition = 0;

    bool isStable(float threshold);
    volatile long &angPosition;
    uint8_t IN1; // Pin forward velocity
    uint8_t IN2; // Pin backwards velocity
    uint8_t ENA; // Pin encoder A
    uint8_t ENB; // Pin encoder B

    float ticksRev = 7.0; // ticks in angPosition that makes a rev
    float gearRatio = 210.0; // reduction gear of gearRatio:1
    float R = 5.3;
    float positionFactor;

    uint8_t speed = 255;
    unsigned long timeout = 5000;
    unsigned long timer = millis();
    float position = 0;
    float lastPosition = 0;
    float desiredPosition = 0;
    float lastAngvelocity = 0;
    float angVelocity = 0; // motor angular position

    MoveMode moveMode = Global;

    bool next = true;
    bool move = false;
    float Kp, Kd, Ki;
    float outputMin = -255;
    float outputMax = 255;
    float outputSpan;
    bool startup = false;
    QuickPID PID; // Controlador PID

    bool test = false;
    bool tune = false;
    sTune tuner = sTune(&position, &angVelocity, tuner.NoOvershoot_PI, tuner.directIP, tuner.printALL);

    float _leftLimit = -2000;
    float _rightLimit = 2000;

    void updateVelocity();
    void updatePosition();
};

#endif
```

E.7 Actuator Class

```

//RodActuator.cpp
#include "RodActuator.h"
#include <Arduino.h>
#define DEBUG false

RodActuator::RodActuator(uint8_t ENA, uint8_t ENB, uint8_t IN1, uint8_t IN2,
    volatile long &angPosition, void (*interruptHandler)()):
    ENA(ENA), ENB(ENB), IN1(IN1), IN2(IN2), angPosition(angPosition) {
    pinMode(ENA, INPUT);
    pinMode(ENB, INPUT);
    pinMode(IN1, OUTPUT);
    pinMode(IN2, OUTPUT);

    attachInterrupt(digitalPinToInterruption(ENA), interruptHandler, CHANGE);

    // Calculate position factor
    // 1 Revolución en el encoder son 7 cambios
    // La relación de engranajes de 235:1
    // El valor de _position está en revoluciones
    // 1/(7*235) = 6.0790273556231e-4
    positionFactor = 2.0 * R * PI / (ticksRev * gearRatio);

    // Inicializar control PID
    PID = QuickPID(&position, &angVelocity, &desiredPosition);
    PID.SetOutputLimits(outputMin, outputMax);
    PID.SetSampleTimeUs(2030);
    PID.SetDerivativeMode(QuickPID::dMode::dOnError);
    PID.SetProportionalMode(QuickPID::pMode::pOnError);
    PID.SetAntiWindupMode(QuickPID::iAwMode::iAwCondition);
    PID.SetMode(QuickPID::Control::automatic);
    PID.SetTunings(22.1, 67, 3.8632);
}

void RodActuator::Update() {
    updatePosition();

    if (tune) {
        switch (tuner.Run()) {
            case tuner.sample:
                updateVelocity();
                break;

            case tuner.tunings:
                angVelocity = 0;
                updateVelocity();
                tuner.GetAutoTunings(&Kp, &Ki, &Kd);
                SetPIDConstants(Kp, Ki, Kd);
                tune = false;
                next = true;
                break;

            default:
                break;
        }
    }

    else if (move) {
        if (startup && position > desiredPosition - 10) {
            startup = false;
            angVelocity = 0.01 * angVelocity;
            updateVelocity();
            PID.SetMode(QuickPID::Control::manual);
            PID.SetMode(QuickPID::Control::automatic);
        }
        PID.Compute();
        updateVelocity();
    }

    else if (test) {
        updateVelocity();
    }

    lastPosition = position;
    lastAngVelocity = angVelocity;

    if (isStable(1.5)) {
        next = true;
    }

    if (!next && move) {
        unsigned long currentTime = millis();
        if (currentTime - timer > timeout) {
            next = true;
        }
    }
}

```

```

}

void RodActuator::Stop() {
    angVelocity = 0;
    next = true;
    move = false;
    test = false;
    updateVelocity();
    updatePosition();
}

void RodActuator::Move(float desiredPosition) {
    next = false;
    timer = millis();
    move = true;
    startup = true;
    if (moveMode == MoveMode::Global) {
        RodActuator::desiredPosition = desiredPosition;
    } else if (moveMode == MoveMode::Relative) {
        RodActuator::desiredPosition += desiredPosition;
    }
}

float RodActuator::GetHomePosition() {
    return homePosition;
}

float RodActuator::GetTargetPosition() {
    return targetPosition;
}

void RodActuator::SetHomePosition(float newHome) {
    RodActuator::homePosition = newHome;
}

void RodActuator::SetTargetPosition(float newSecondHome) {
    RodActuator::targetPosition = newSecondHome;
}

void RodActuator::MoveToHome() {
    float desiredPosition = homePosition;
    if (moveMode == MoveMode::Relative) {
        desiredPosition = homePosition - desiredPosition;
    }
    Move(desiredPosition);
}

void RodActuator::MoveToTarget() {
    float desiredPosition = targetPosition;
    if (moveMode == MoveMode::Relative) {
        desiredPosition = targetPosition - desiredPosition;
    }
    Move(desiredPosition);
}

float RodActuator::GetPosition() {
    return position;
}

void RodActuator::SetPIDConstants(float Kp, float Ki, float Kd) {
    Kp = Kp;
    Ki = Ki;
    Kd = Kd;
    PID.SetTunings(Kp, Ki, Kd);
}

bool RodActuator::Next() {
    return next;
}

void RodActuator::SetMaxSpeed(uint8_t maxSpeed) {
    if (maxSpeed < 50) {
        maxSpeed = 50;
    }
    RodActuator::speed = maxSpeed;
}

void RodActuator::TestVelocity(float velocity) {
    RodActuator::angVelocity = velocity;
    test = true;
}

void RodActuator::SetTimeout(float timeout) {
    RodActuator::timeout = timeout;
}

int RodActuator::GetTimeout() {
    return timeout;
}

void RodActuator::AutoTune() {
    tune = true;
    next = false;
}

```

```

tuner.Configure(1000, 200, 100, 0, 3, 1, 500);
tuner.SetEmergencyStop(1000);
}

void RodActuator::SetMoveMode(MoveMode moveMode) {
    RodActuator::moveMode = moveMode;
}

// =====
// Private methods
// =====
bool RodActuator::isStable(float threshold = 1) {
    return abs(position - desiredPosition) <= threshold;
}

void RodActuator::updateVelocity() {
    uint8_t vel1 = 0;
    uint8_t vel2 = 0;
    float minInput = 0;
    uint8_t minPWM = 0;
    if (angVelocity > minInput) {
        // vel1 = abs((float)_angVelocity / (float)_outputMax) * (float)_speed;
        vel1 = map(abs(angVelocity), minInput, outputMax, minPWM, RodActuator::speed);
        if (vel1 < 40)
            vel1 = 0;
    } else if (angVelocity < -minInput) {
        // vel2 = abs((float)_angVelocity / (float)_outputMax) * (float)_speed;
        vel2 = map(abs(angVelocity), minInput, outputMax, minPWM, RodActuator::speed);
        if (vel2 < 40)
            vel2 = 0;
    }
    // Serial.println(vel1);
    if (DEBUG && (move && !next)) {
        Serial.print("Velocity: ");
        Serial.print(angVelocity);
        Serial.print(" ");
        Serial.print(vel1);
        Serial.print(" ");
        Serial.print(vel2);
        Serial.print(" Position: ");
        Serial.print(position);
        Serial.println();
    }
    analogWrite(IN1, vel1);
    analogWrite(IN2, vel2);
}

void RodActuator::updatePosition() {
    position = angPosition * positionFactor;
}

```
