Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

ModuGrip

Project Engineering

Year 4

Allyn McKenna Patterson

Bachelor of Engineering (Honours) in Software and Electronic Engineering

Atlantic Technological University

2023/2024

Declaration

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering (Honours) in Software and Electronic Engineering at Galway-Mayo Institute of Technology.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

_____Allyn McKenna Patterson_____

Acknowledgements

I would like to thank my family, friends, lecturers, and project supervisor Niall O'Keeffe for their continued support throughout this project. This project wouldn't have been possible without their guidance.

Contents

1 Summary

My goal for this project was to combine the software and hardware skills that I have learned over the past four years to create a demonstratable robotic arm controlled through a custom-made web application.

The ModuGrip is a 3-DOF (three degrees of freedom) robotic arm with a modular end effector system. The modular end effectors allow the user to remotely swap tools during use, enabling the operator to complete multiple jobs with one robotic arm. The ability to swap tools makes the ModuGrip a great asset when working in remote, dangerous, or controlled environments. The web application features three control methods, inverse kinematics calculations, script creation UI, and a virtual representation of the ModuGrip.

This project required careful planning in the early stages and parallel development of hardware and software. There were times that I could not progress until the web application and physical device were equally developed. I initially adopted an agile kanban methodology but as the project got more complex, I realised that planning my work was taking more time than it was saving. At that point I decided to stop the formal planning process and incrementally tackle the next obvious task.

A broad range of technologies were used to complete this project. The web application frontend is written in React using the NextJS framework. The backend is written in Java Spring Boot with a MongoDB database. The ModuGrip was designed using AutoDesk's Fusion 360 and 3D printed with a Sovol SV06 Plus. The ModuGrip's ESP32 communicates with the web application over MQTT using Ubidots as the broker.

I am proud of what I have been able to accomplish in the eight months of work. I have been able to create a polished web application with a good user experience and design a functional robotic arm from scratch. This project was a huge learning experience and an exercise in discipline and patience.

2 Poster

Oliscoil Teicneolaíochta an Atlantaigh
Atlantic Technological University

ModuGrip

Allyn McKenna Patterson
BEng (H) in Software & Electronic Engineering

Project Summary

What is the ModuGrip?

The ModuGrip is a 3-DOF (three degrees of freedom) robotic arm with a modular end-effector system.

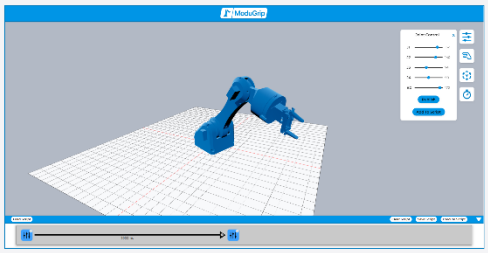
The modular end-effectors allow the user to remotely swap tools during use, enabling the operator to complete multiple jobs without entering the operating environment.

The ability to swap tools makes the ModuGrip a great asset when working in remote, dangerous, or controlled environments.

How it Works:

1. The ModuGrip is operated through my Nextjs web application featuring a virtual model, script generation and multiple control options.
2. The user sets the desired position of the ModuGrip and publishes the values to Ubidots, a cloud-based MQTT broker.
3. The ESP32 is the brains of the ModuGrip. It connects to Ubidots over Wi-Fi and waits for new values to be published.
4. When new values are received, the various actuators move the ModuGrip into position.

Web Application



Control Methods

Different tasks require different methods of control.

The web application features manual control of each joint using sliders, and direct control over the end-effector position using coordinates.

The application runs inverse kinematics calculations to convert coordinates into motor movements.

Script Creation

Script creation is an essential feature in any robot control software.

The ModuGrip web application empowers the user to chain together a sequence of movements and save them for later. The freedom to swap tools mid-script enables the ModuGrip to complete complex, multi-process tasks.

Virtual Model

I wanted to ensure that the operator could remotely control the ModuGrip from anywhere in the world, even when it's out of sight.

The virtual model gives an accurate representation of the desired state of the robot before executing a movement. This is a vital feature for both safety, and user experience.

Technologies Used

Software:

NextJS: An open source React framework.

React Three Fiber: A React renderer for ThreeJS.

Java Spring Boot: An open source Java framework for backend development.

MongoDB: A NoSQL database.

Hardware:

DS3240SG: 45kg-cm servo motor.

DS3225MG: 28kg-cm servo motor.

Parallax #900-00005: 2.74kg-cm servo motor.

RS 180-5278: Unipolar stepper motor.

Tools:

Fusion 360: Autodesk's CAD software.

Sovol SV06 Plus: 3D Printer.

Sovol3D Cura : 3D Slicer application.

AWS EC2: Amazon cloud computing platform.

Vercel: Serverless NextJS deployment platform.

Ubidots: A cloud hosted MQTT broker.

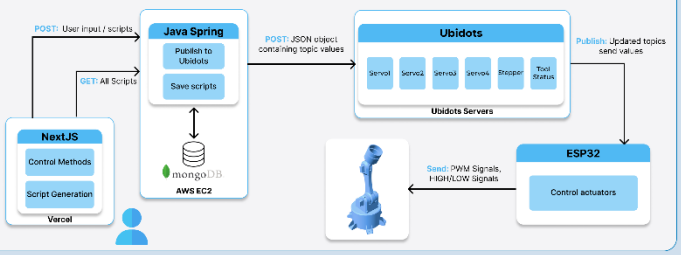
Hardware Design

The ModuGrip was designed using AutoDesk's Fusion 360 software and 3D printed with a Sovol SV06 Plus.

The arm was printed with PLA filament, sanded, and coated in multiple layers of paint and polyurethane clear coat to provide a durable surface.

The entirely original design features five servo motors, one stepper motor, and a mechanical locking system.

Project Architecture



See the ModuGrip in Action

SCAN ME

Figure 2-1 Project Poster

3 Introduction

This report will cover the research and development steps that enabled me to complete this project. I chose to attempt this project because the challenge interested me. Each stage of the project tested me in new ways and kept me thoroughly engaged. My hope is that the knowledge and learning experiences shared in this report will inspire and assist others who are interested in completing projects of similar scope. This report will give a brief insight into my hardware selection decisions, 3D design process, full stack web application development and kinematics calculations.

4 Background

4.1 Robot Terminology

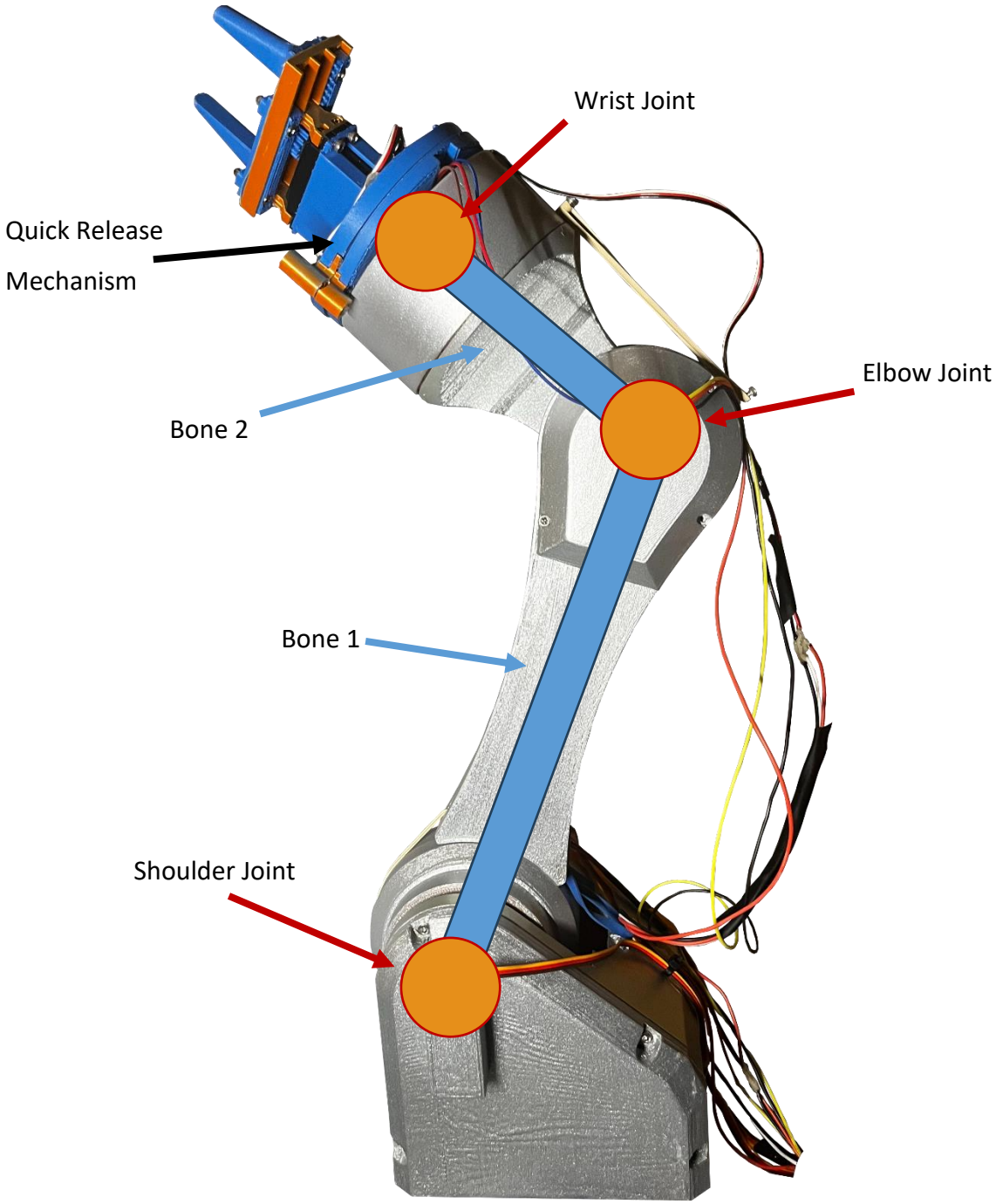


Figure 4-1 ModuGrip Terminology

5 Project Architecture

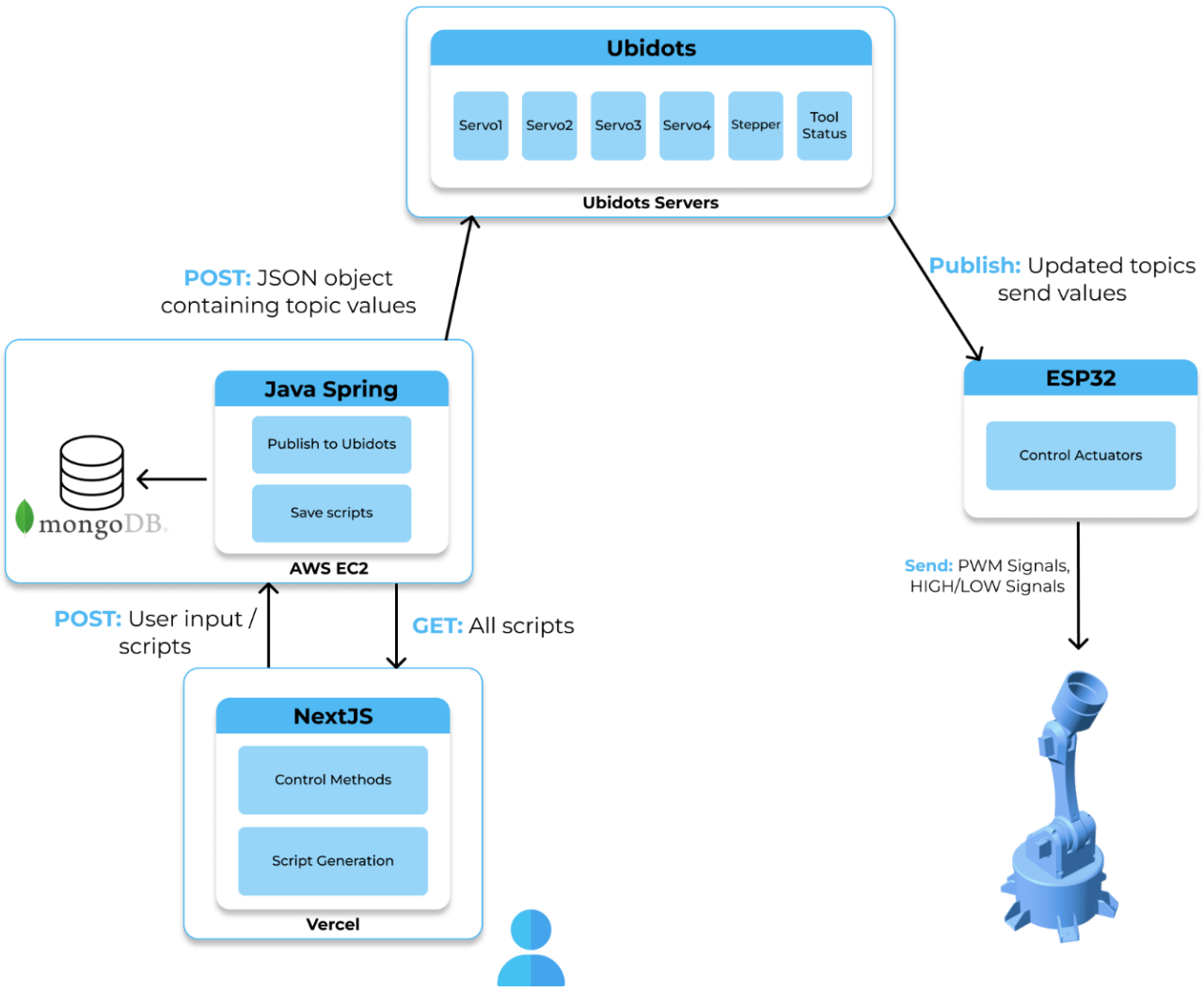


Figure 5-1 Architecture Diagram

6 Project Plan

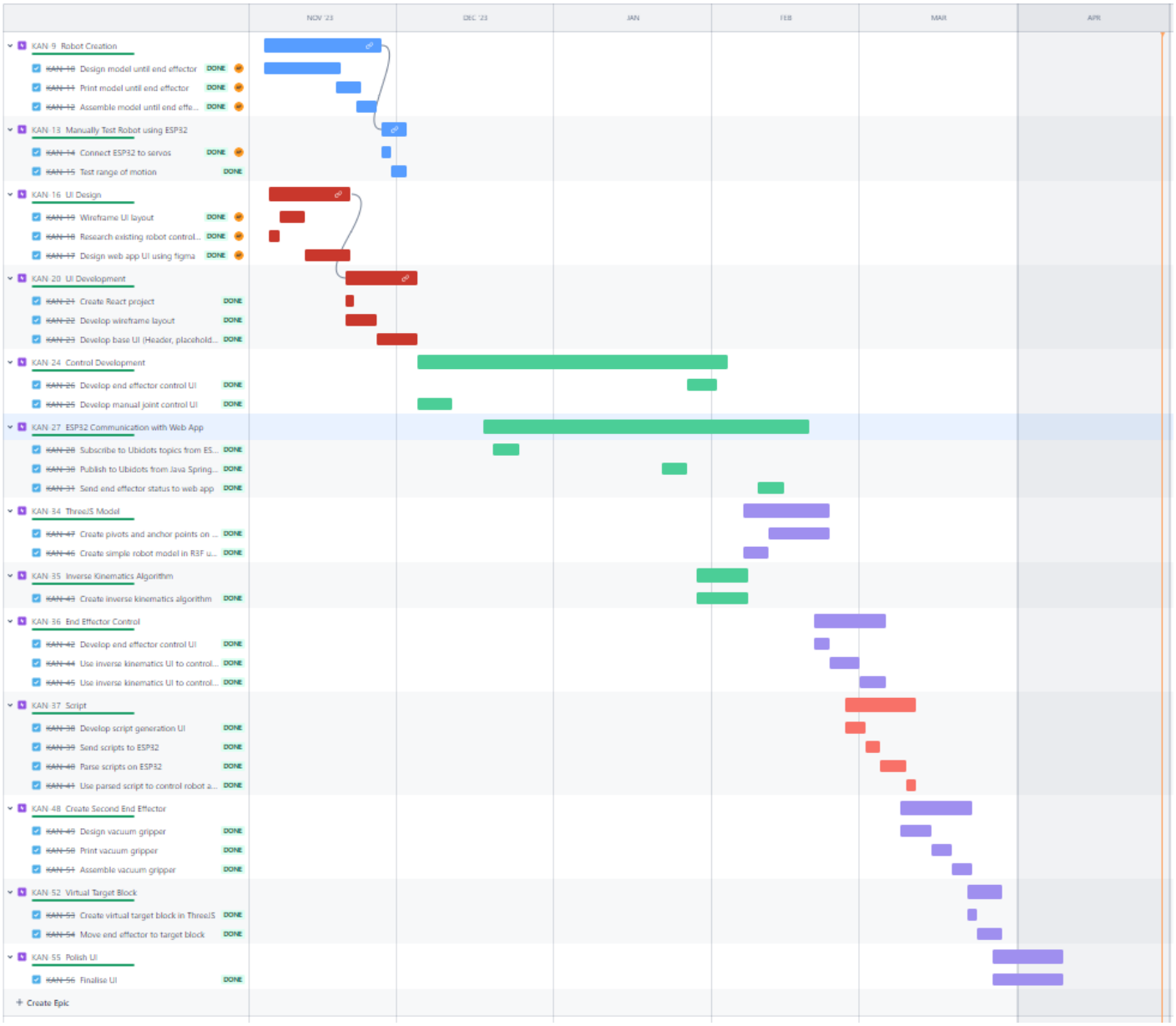


Figure 6-1-1 Project Timeline

7 Hardware Design

7.1 Component Selection

This section will cover the hardware components used in this project, my reasons for selecting these components, and circuit diagrams. The code that controls these components will be covered later in the ESP32 firmware section.

7.1.1 Servo Motors

A servo motor is a rotary actuator that allows for control over angular position via pulse width modulated (PWM) signals. [1] There are three different types of servo motors: positional rotation, continuous rotation, and linear. [2] The ModuGrip features five positional rotation servo motors so they will be the focus of this section. A positional rotation servo motor features physical stops in the inner gear system which limit the output shaft to a range of 0-180 degrees.

Servo motors feature four core components: a DC motor, a gearbox, a potentiometer, and a control circuit. The small DC motor operates at high RPM (rotations per minute) with very low torque, so an arrangement of gears is used to convert the high rotation speed to a much slower, high torque output. [2] The potentiometer monitors the position of the output shaft and acts as feedback for the control circuit, which also interprets the PWM signals that the servo receives.

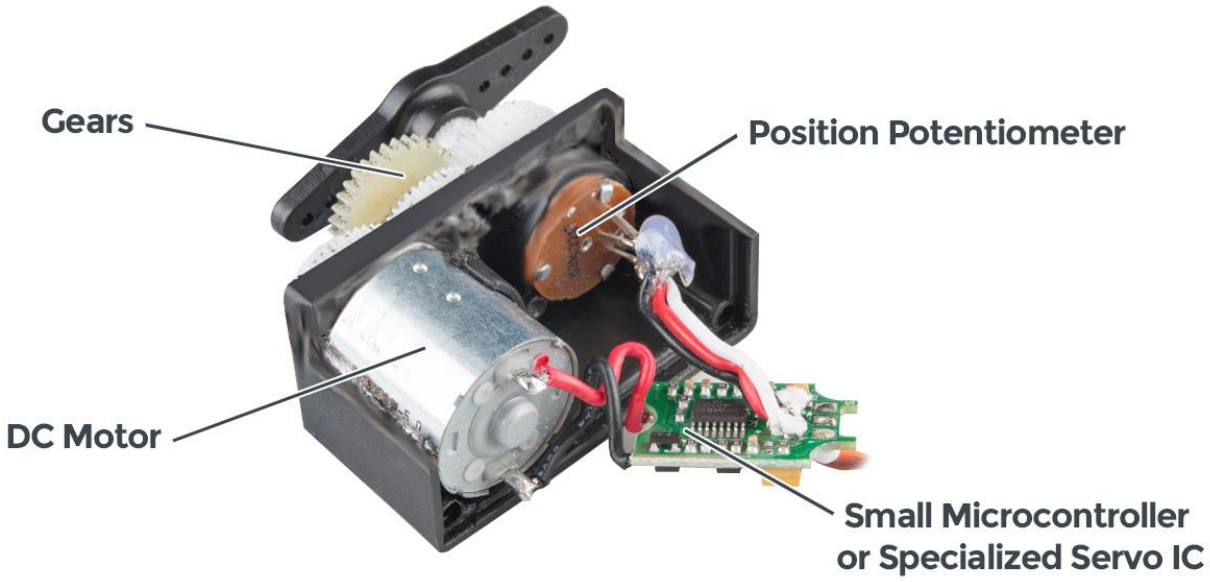


Figure 7-1-1 Servo Design [2]

7.1.1.1 DS3240SG

The DS3240SG is a high torque servo with a stainless-steel gear system. I chose this servo due to its 45kg-cm stall torque, 180° of operation, and appropriate form factor. This servo can draw up to 3.9A at 6.8V when stalled so I needed to be sure that it would not be held at stall current, so I don't exceed the maximum operating temperature of 70°C.

Figure 7-1-2 DS3240SG Product Image [3]

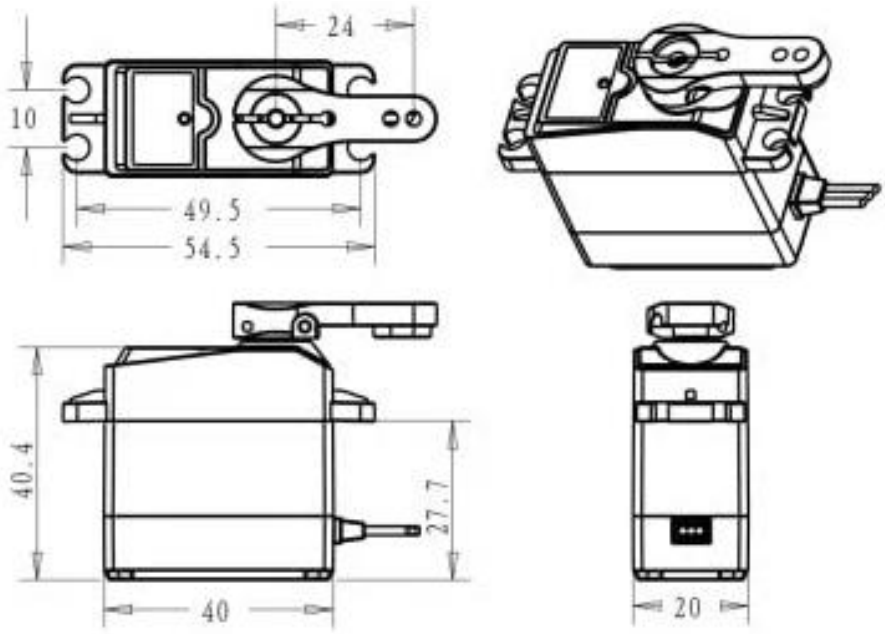


Figure 7-1-3 DS3240SG Dimensions [3]

DS3240SG Datasheet: [3]

Mechanical Specifications:

Item	Specification
Weight	60g
Gear ratio	342
Operating voltage range	4.8-6.8V
Operating temperature range	-25°C-70°C

Electrical Specifications:

Item	5V	6.8V
Idle current (stopped, no load)	4mA	5mA
Operating speed (no load)	0.2sec/60°	0.17sec/60°
Stall torque (locked)	36kg-cm	45kg-cm
Stall current (locked)	3.1A	3.9A

Control Specifications:

Item	Specification
Control System	PWM
Pulse width range	500-2500 µsec
Neutral position	1500 µsec
Range of rotation	0-180°
Dead band width	3 µsec
Rotating direction	Counterclockwise

7.1.1.2 DS3225MG

The DS3225MG is very similar to the DS3240SG. It also operates between 4.8V and 6.8V, with a smaller stall torque of 24.5-28kg-cm and a stall current draw of 2.1-2.9A. The ModuGrip makes use of two DS3225MG servos, one acting as the elbow actuator and the other aligned opposite the DS3240SG to offer support. This servo was a good option for the "elbow" joint as the load acting on this servo would be much less than the "shoulder" joint.

Figure 7-1-4 DS3235MG Product Image

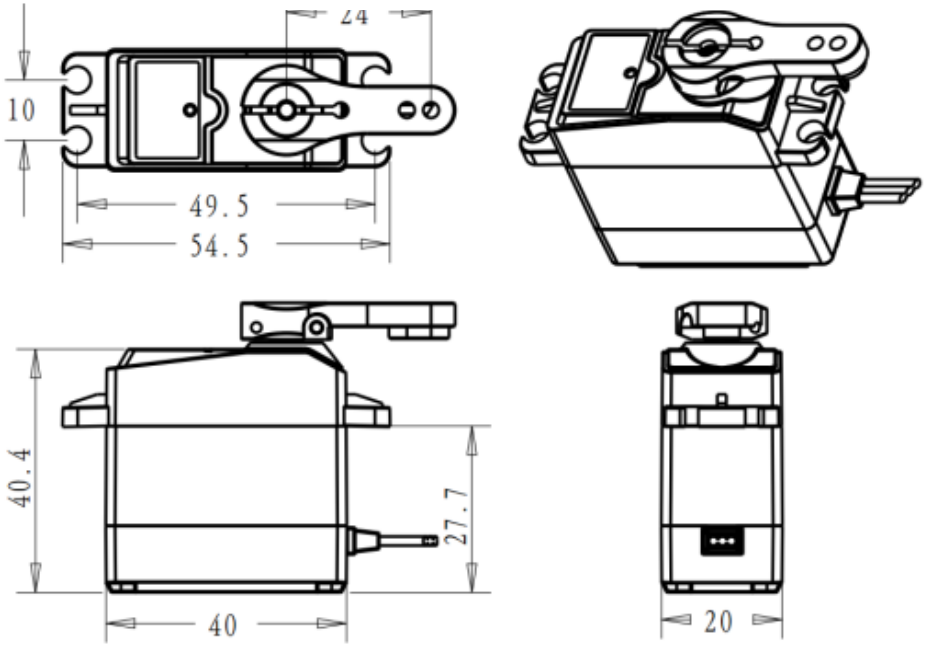


Figure 7-1-5 DS3225MG Dimensions [4]

DS3225MG Datasheet: [4]

Mechanical Specifications:

Item	Specification
Weight	60g
Gear ratio	275
Operating voltage range	4.8-6.8V
Operating temperature range	-25°C-70°C

Electrical Specifications:

Item	5V	6.8V
Idle current (stopped, no load)	4mA	5mA
Operating speed (no load)	0.16sec/60°	0.14sec/60°
Stall torque (locked)	24.5kg-cm	28kg-cm
Stall current (locked)	2.1A	2.9A

Control Specifications:

Item	Specification
Control System	PWM
Pulse width range	500-2500 µsec
Neutral position	1500 µsec
Range of rotation	0-180°
Dead band width	3 µsec
Rotating direction	Counterclockwise

7.1.1.3 Parallax Standard Servo #900-00005

The Parallax #900-00005 is a much lower power servo than the previous two. It operates from 4-6V with a maximum current draw between 90-190mA and only 2.74kg-cm of torque. The ModuGrip features two #900-00005s, one to rotate the quick release mechanism and the other to actuate the gripper tool. The low power that this servo offers is no problem in my application as both applications require little torque.

Figure 7-1-6 Parallax #900-00005 Product Image [12]

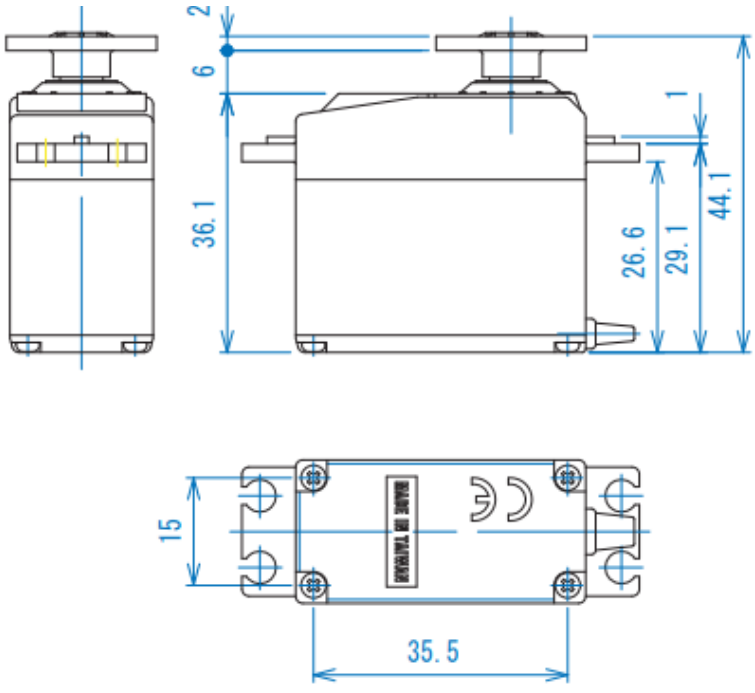


Figure 7-1-7 Parallax #900-00005 Dimensions [13]

Mechanical Specifications: [5]

Item	Specification
Weight	42g
Operating voltage range	4.8-6.8V
Operating temperature range	-10°C-50°C

Electrical Specifications:

Item	6V
Idle current (stopped, no load)	20mA
Operating current (no load)	100mA
Operating speed (no load)	0.19sec/60°
Torque	2.7kg-cm

Control Specifications:

Item	Specification
Control	PWM
Pulse width range	500-2530 µsec
Neutral position	1520 µsec
Range of rotation	0-180°
Rotating direction	Counterclockwise

7.1.2 Stepper Motor

Stepper motors are DC motors that rotate in a series of small angular steps. They have coils that are organised in groups called phases. There are two basic winding arrangements for the coils in a two-phase motor: bipolar and unipolar.

Bipolar stepper motors use a single coil per winding which creates more powerful magnetic fields allowing for higher torque forces to be achieved.

Unipolar steppers use tapped coils, each side can be magnetised independently. The current through each coil runs in a different direction dependent on which side of the coil is magnetised so the polarity of each phase can be reversed without reversing the current of the entire circuit. This results in a simpler driver circuit. [6]

7.1.2.1 RS 180-5278

The RS 180-5278 is a unipolar hybrid stepper motor. The motor features a stepping angle of 1.8° which is equivalent to 200 steps per revolution. I chose to use this motor due to its relatively low cost when compared to bipolar stepper motors of the same specifications.

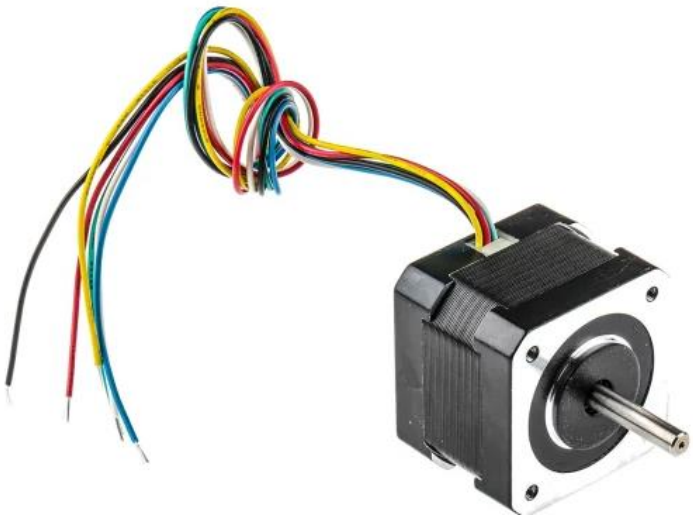


Figure 7-2-1 RS 180-5278 Product Image [14]

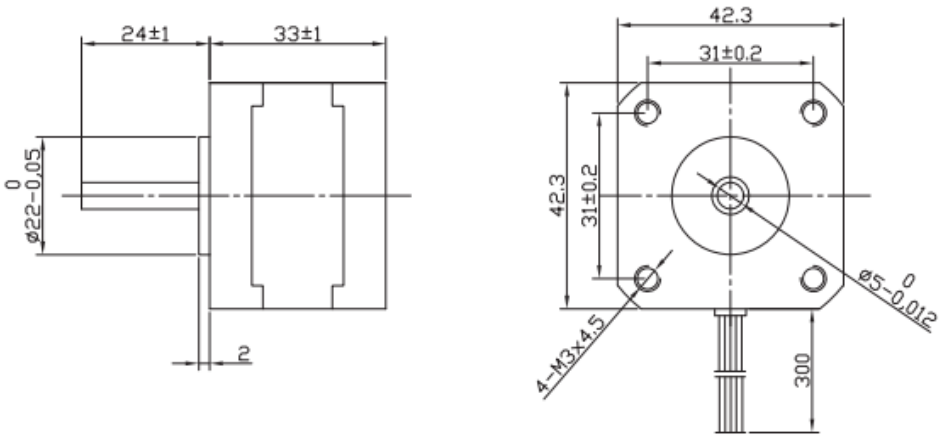


Figure 7-2-2 RS 180-5278 Dimensions [15]

Specifications:

Item	Specification
Rated Voltage	9.6V
Current / Phase	0.4A
Resistance / Phase	24Ω
Inductance / Phase	15mH
Holding Torque	0.158 Nm
Rotor Inertia	35g-cm^2
Weight	0.22kg

Connections:

Wire colour	Function
Black	Phase A
Green	Phase A-
Red	Phase B
Blue	Phase B-
Yellow	Com Phase A
White	Com Phase B

7.1.2.2 Stepper Driver

The ULN2003 is one of the most used motor driver ICs due to its high voltage rating and ability to drive up to 500mA per pin. I am using a ULN2003APG driver board which features motor connection pins, control inputs, and step indicator LEDs.

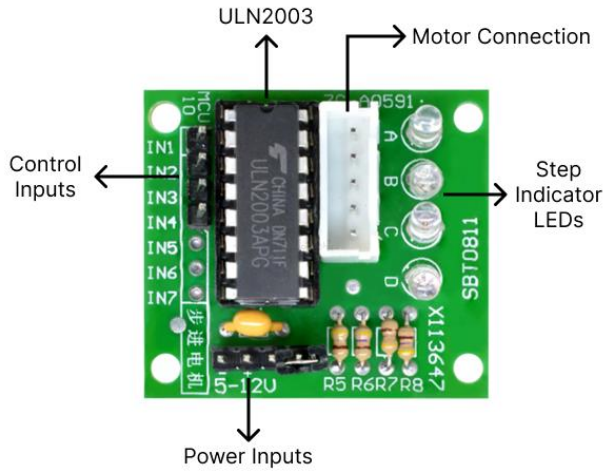


Figure 7-2-3 ULN2003 Board

Stepper Circuit Diagram:

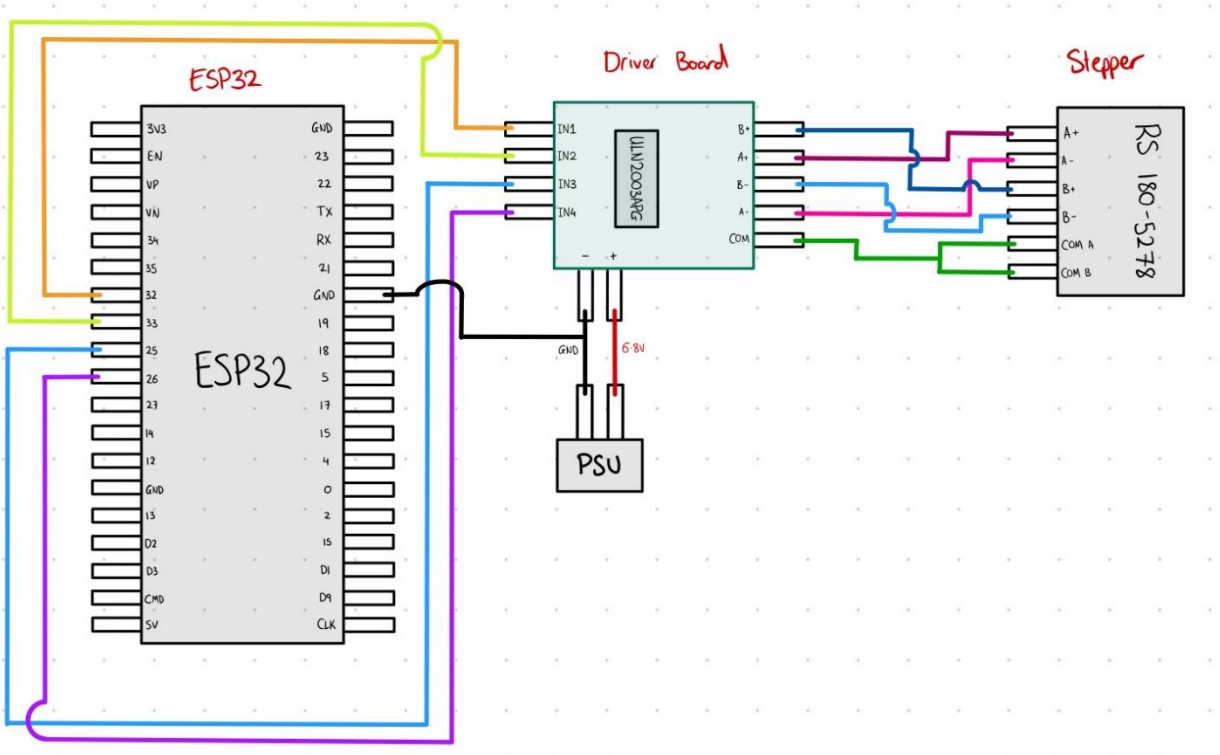


Figure 7-2-4 Stepper Circuit Diagram

7.1.3 ESP32

7.1.3.1 Board Selection

The ESP32 is a series of low power microcontroller development boards made by Espressif. An ESP32 board includes Wi-Fi capabilities and a dual core processor. I chose this board due to its fast wireless connection over Wi-Fi (up to 150Mbps) [7], capable CPU, and ability to produce PWM signals on its peripheral I/O pins.

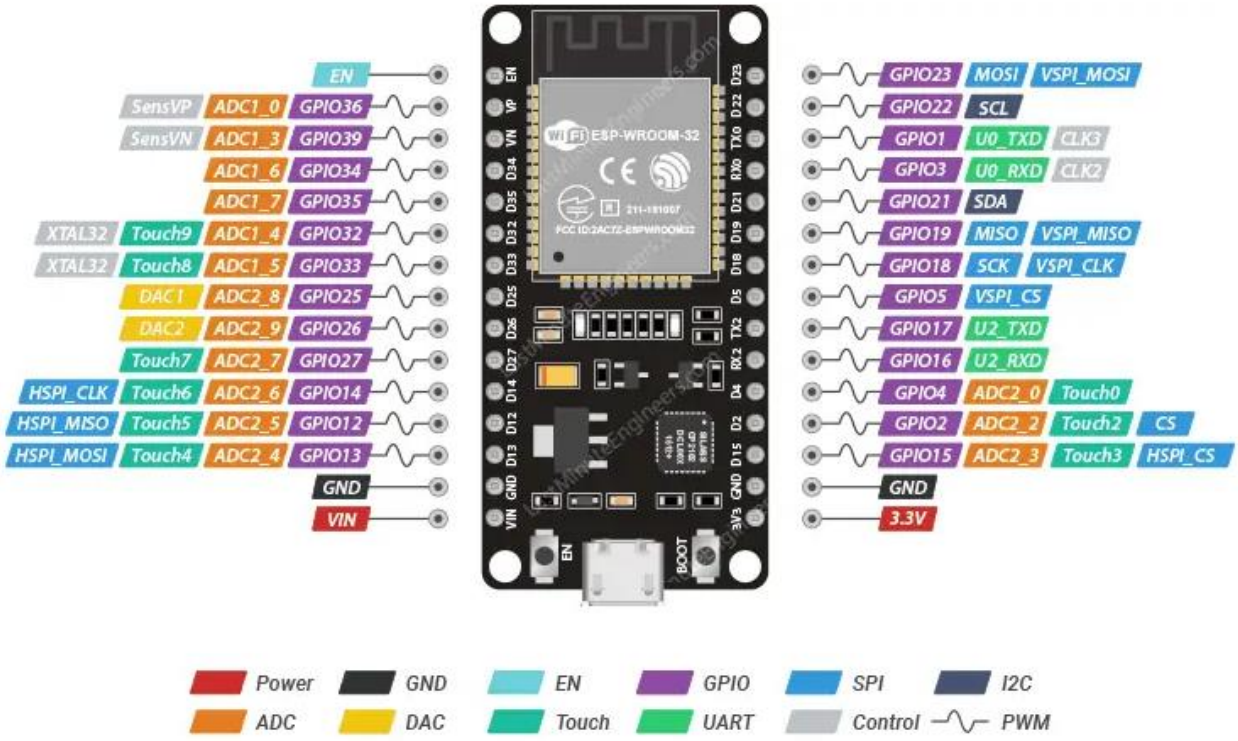


Figure 7-3-1 ESP32 Pinout [16]

7.1.4 Power Supply

I have been using a Thandar TS3022S throughout this project. The TS3022S is a dual output DC power supply capable of 30V at 2A.

I use output one to power the high torque servos and stepper motor at 6.8V and output two to power the lower torque servos and end effector attachments at 5V. I chose to use both outputs to ensure each component receives enough current. My highest torque servo can draw up to 3.9A at 6.8V when stalled so I wanted to ensure that if I cross the 2A limit of this power supply, the other components could still operate. It was also a simple way to provide separate 6.8V and 5V power lines without needing to step down the voltage.

Figure 7-4-1 Thandar TS3022S Product Image [17]

7.1.5 DC Geared Motor

A DC Geared motor is a DC motor combined with a gearbox. The gearbox is attached to the output shaft of the DC motor and by meshing gears of different sizes together, the rotational speed and output torque can be manipulated.

7.1.5.1 MFA 918D301/1

My “screwdriver” end effector makes use of the MFA 918D301/1. This motor operates between 1.5V and 3V with a gear ratio of 30:1. The gearbox is constructed from brass and steel and features a protective plastic cover to prevent dust ingress.

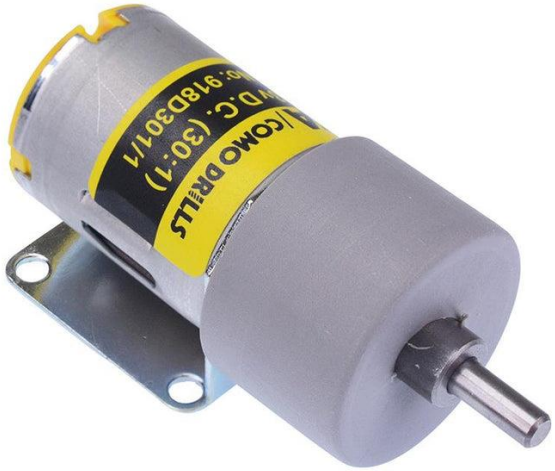


Figure 7-5-1 MFA 918D301/1 Product Image [18]

Specifications:

Item	Specification
Voltage range	1.5V – 3V
No load current	0.24A
No load speed	8800RPM
Geared speed	293RPM
Output shaft length	4mm
Dimensions	67.5 x 24mm
Weight	75g

7.1.6 Electromagnet

An electromagnet is a device that produces a magnetic field when supplied with electrical current. Electromagnets work by wrapping a long length of conductive wire around a ferromagnetic core. The strength of the magnetic field is determined by the number of turns in the coil, the amount of current flowing through the coil, and the core material.

I chose to incorporate an electromagnetic into the magnetic end effector due to its adjustable strength and temporary magnetism.

7.1.6.1.1 KK-P20/15

The KK-P20/15 is a 12V DC solenoid electromagnetic with a 3kg lifting capacity. It contains an iron core and is designed to release any residual magnetism after de-energisation. I chose this electromagnet due to its small form factor, relatively low operating current (0.2A), and powerful magnetic force.

Figure 7-6-1 KK-P20/15 Product Image [8]

Specifications: [8]

Item	Specification
Voltage rating	12V
Current	0.2A
Lift power (2mm thick steel plate)	3kg
Lift power (1mm spacing)	0.5kg
Operating temperature range	-20 to 120°C
Dimensions	20mm x 15mm
Weight	25g

8 Robot Design

8.1 3D Model Design

There were many areas of consideration I had to be aware of when designing the ModuGrip. Range of motion, torque requirements, hardware housing, and serviceability were some of the main areas of focus while I was designing each part.

8.1.1 CAD Software

I completed all my design tasks in AutoDesk's Fusion 360. Fusion 360 is a 3D modelling software capable of modelling, simulation, and documentation. I chose this CAD software due to its cloud-based system, free student access, and abundance of documentation.

Fusion 360 is the first and only CAD software that I have used, therefore I experienced a steep learning curve when I first started designing in November 2023. It follows the typical "sketch, extrude" process of a CAD software. First you draw a sketch on one axis, then you extrude that 2D schematic to turn it into a 3D shape.

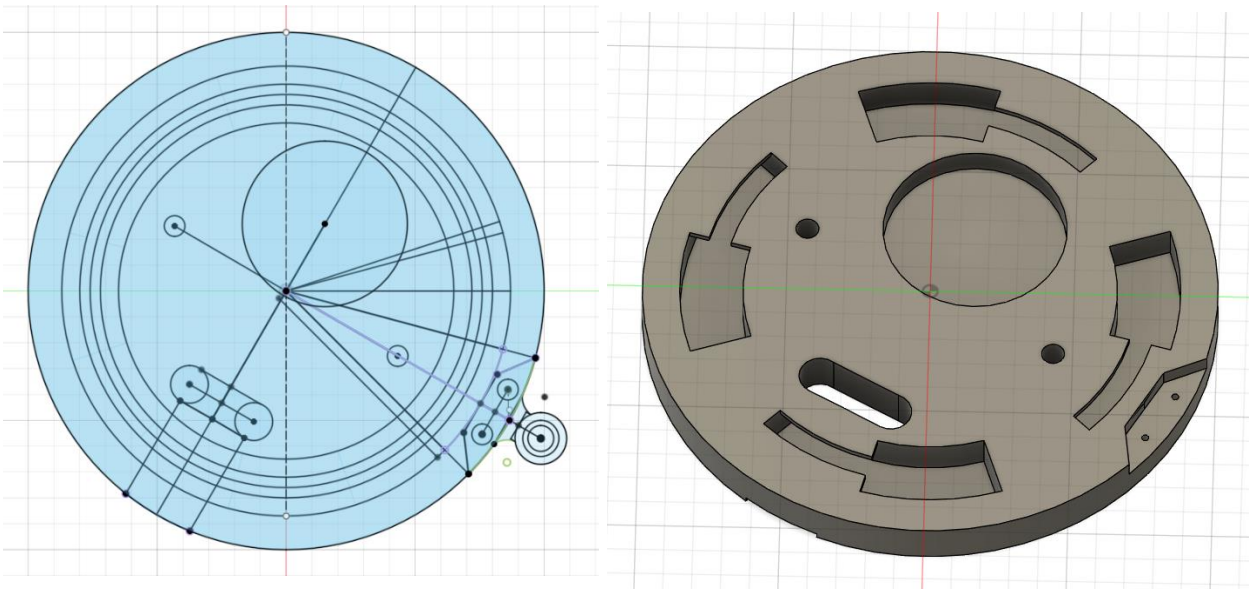


Figure 8-1-1 Female quick release sketch vs extruded.

8.1.2 Challenges

The main challenges I encountered while designing the ModuGrip were weight limits and housing components.

Most of the force acting on the shoulder joint came from the end-effector and servo mounted in the wrist joint. These components on their own weighed less than 700g so one may think they would be no problem for the 45kg-cm servo, however the length of bone 1 and bone 2 greatly increase the load acting on the joint.

One of my solutions to this problem was to attach an elastic band connecting bone 1 to the base. This band increased in tension as the tool lowered and took some of the weight off the servos.

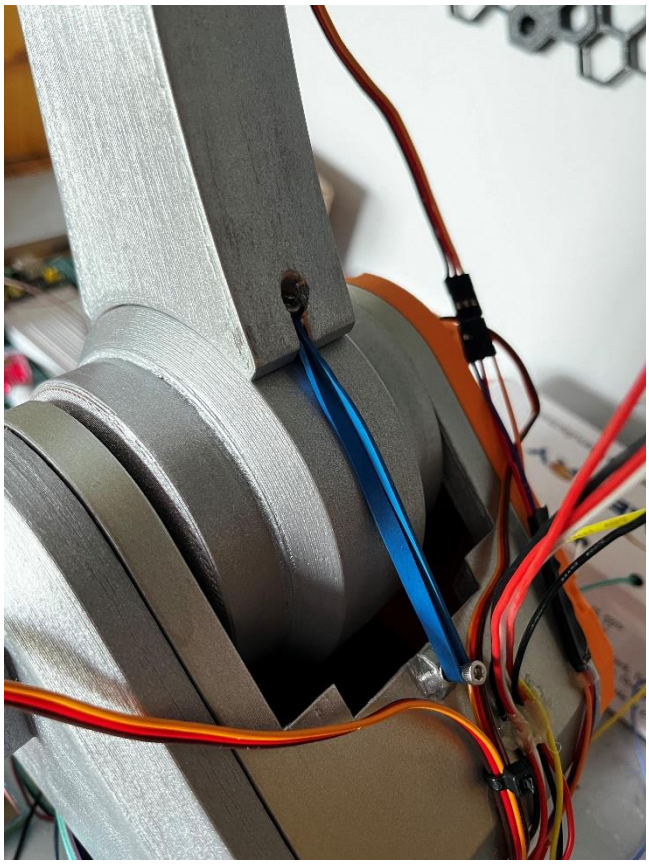


Figure 8-1-2 Elastic band tension support

8.2 Key Design Areas

8.2.1 Joints

The joints in this project use a gear that passes through the bone. The servo shaft slots into the end of the gear to provide rotational force. This simple design ensures the joints are serviceable. If the joint gets damaged, only the gear needs to be reprinted.

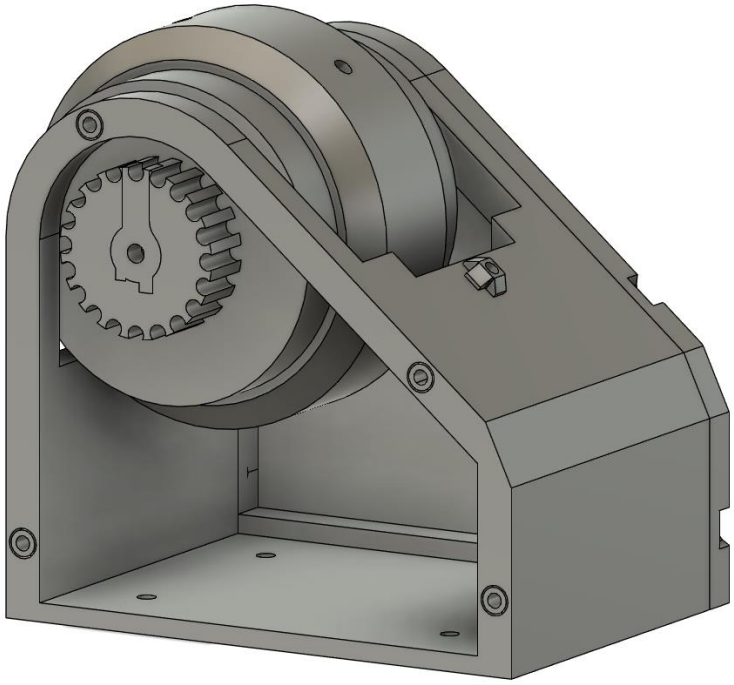


Figure 8-2-1 Shoulder box joint design

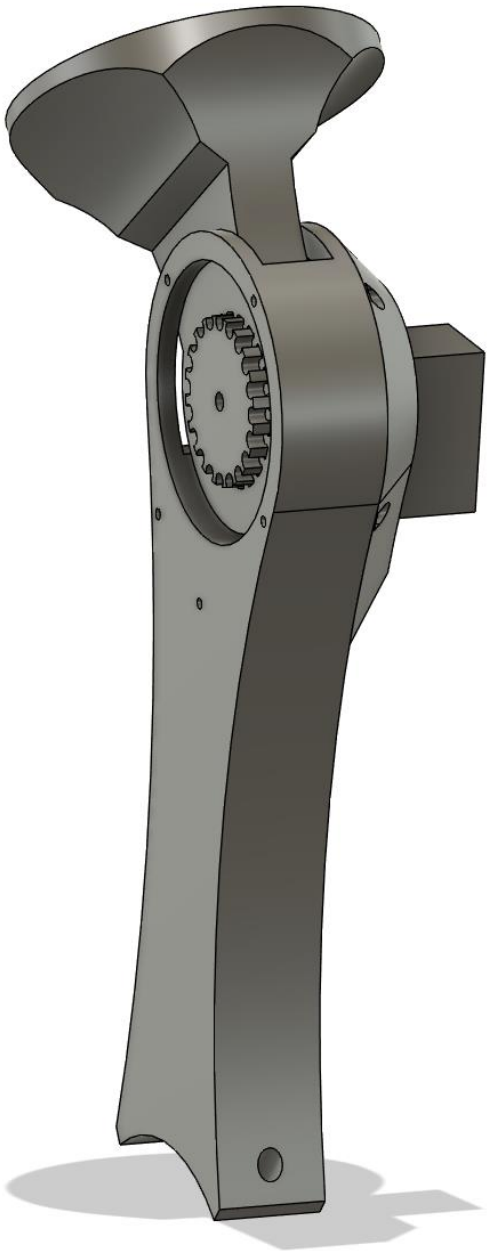


Figure 8-2-2 Elbow joint design

8.2.2 Quick Release Mechanism

The quick release mechanism features a “T-slot bayonet mount”. When the slots are aligned the two halves can press together and twist to lock in place. When locked in place the pogo pin connectors make contact and provide a conductive path for power and signal lines to flow to the end effector.

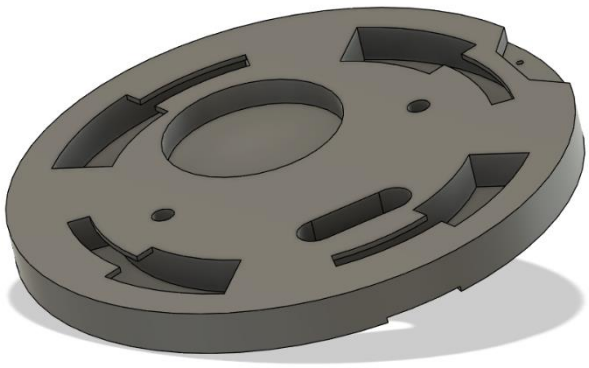


Figure 8-2-3 Female T-Slot Bayonet Mount

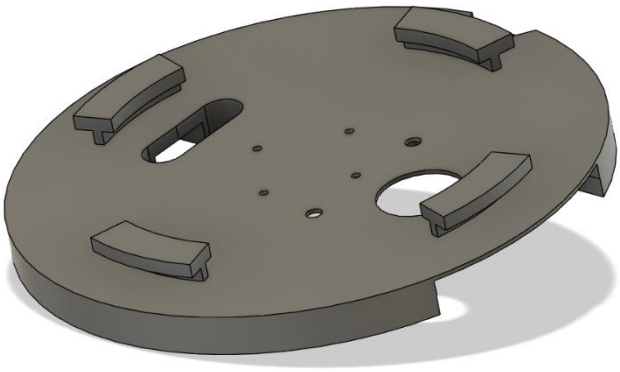


Figure 8-2-4 Male T-Slot Bayonet Mount

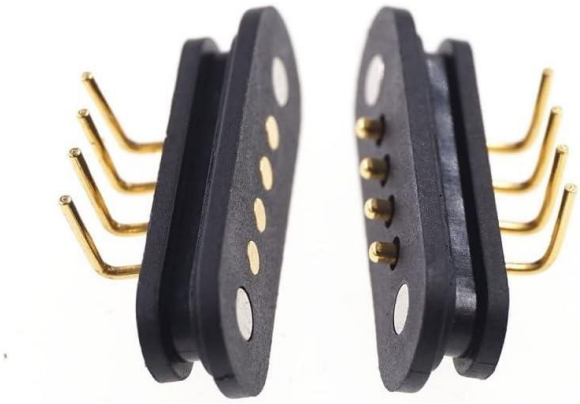


Figure 8-2-5 Four Pin Pogo Connector

8.2.3 End Effectors

8.2.3.1 Electromagnet Tool

The electromagnet tool consists of an 80mm long housing for the electromagnet. The housing is extruded so far from the quick release mechanism to match the gripper length for inverse kinematics calculations.

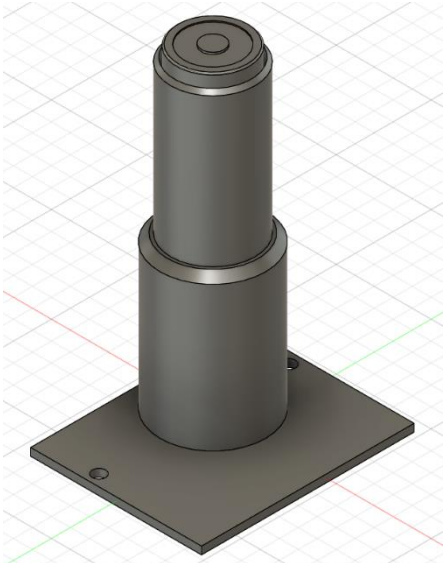


Figure 8-2-6 Electromagnet Tool Model

8.2.3.2 Geared Motor Tool

The geared motor tool also extends roughly 80mm from the quick release mechanism. The motor attaches to this housing using M3x8 screws.

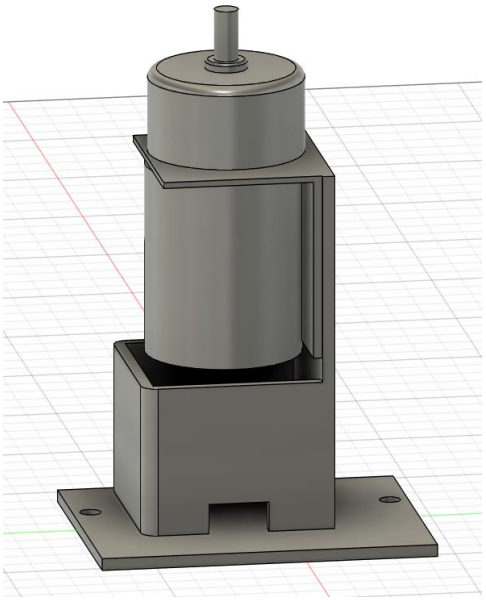


Figure 8-2-7 Geared Motor Tool Model

8.2.3.3 Two Jaw Gripper Tool

The gripper tool consists of five separate parts. The linear movement of the jaws is powered by a servo motor mounted to a rack and pinion mechanism. The racks and pinion translate the rotational movement of the motor shaft into the linear movement needed to close the jaws.

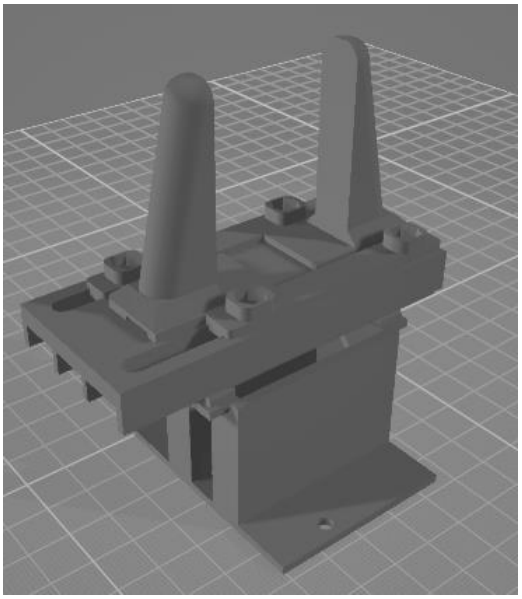


Figure 8-2-8 Gripper Tool Model

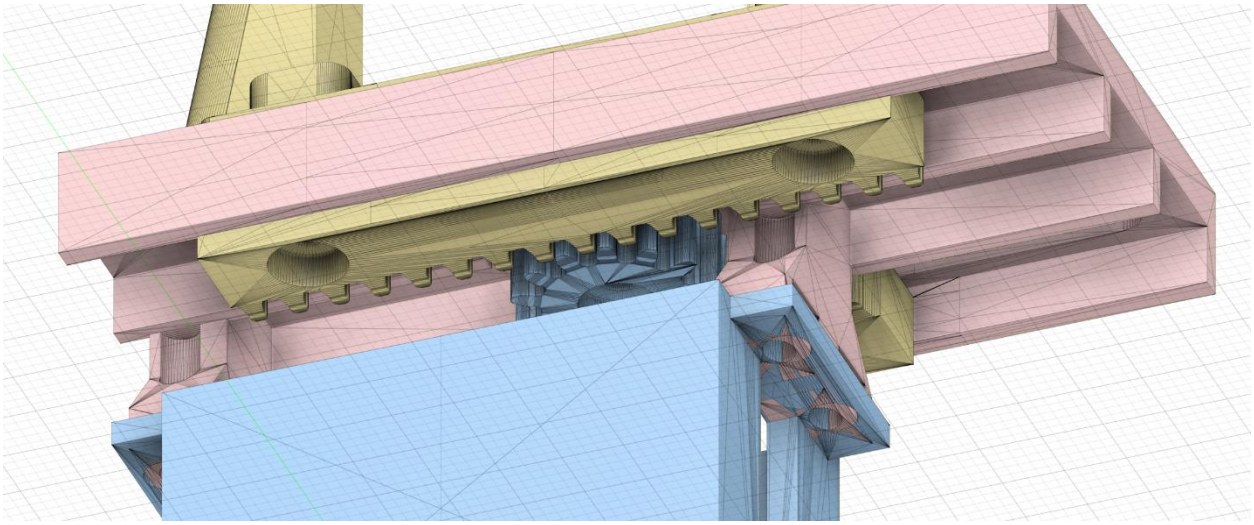


Figure 8-2-9 Rack and Pinion Mechanism

9 Software Development

9.1 NextJS Frontend

NextJS is an open-source React framework for building full-stack web applications. I chose to develop my web application in React because of the good developer experience and rich ecosystem.

9.1.1 User-Interface Design

UI design is extremely important in robot control applications. My goals when designing this application were usability, reliability, and efficiency. I wanted the application to be intuitive and easy to use, even for users with minimal experience in robotics.

Pre-Development Design:

I began by designing the entire UI in Figma, a web design application. Creating a clear idea of the finished product was highly beneficial when it came time to begin development. Having a design to reference kept me focused on the finish line.

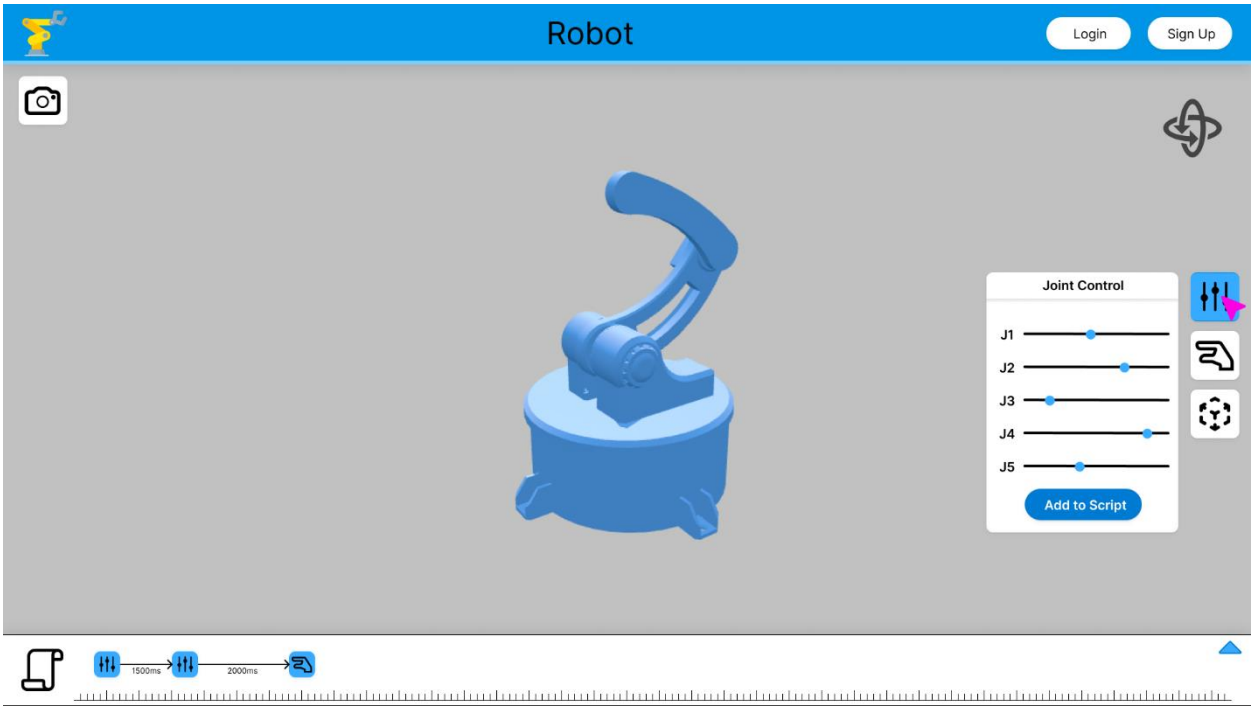


Figure 9-1-1 Pre-Development Design

Developing the User Interface:

My website is a single page application composed of twelve nested React components. Component based development kept my project maintainable and easy to make incremental design changes.

All the styling was done in vanilla CSS by hand. I had researched component libraries in the early stages of this project but felt that they did not fit this use case. I had a very loose design guide from my pre-development design, but I gave myself creative freedom to deviate whenever I felt it was right. I'm glad that I didn't enforce strict design rules because I feel that my finished web application looks significantly better than the pre-development version.

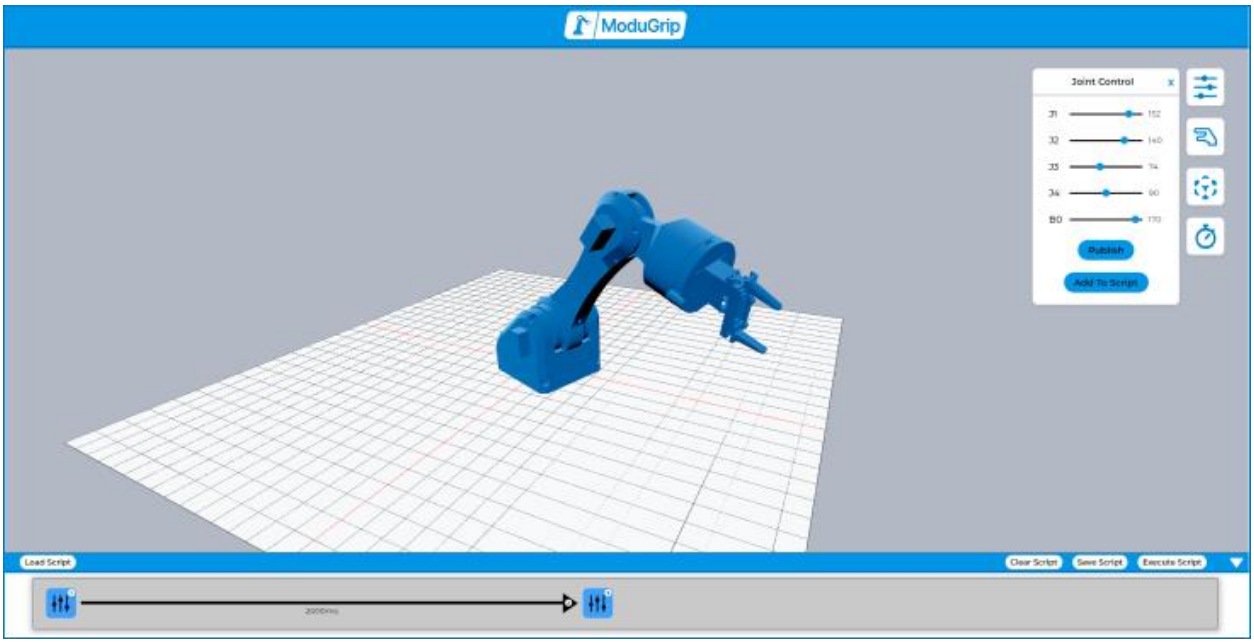


Figure 9-1-2 Developed Web App

Figure 9-1-3 Web App Colour Palette

9.1.2 Control Methods

Control methods make or break a robotics control application. My project offers the user three methods of moving the ModuGrip: manual control over each joint using sliders, entering coordinates for the end effector to manoeuvre to, and dragging a virtual target cube for the end effector to manoeuvre to.

I felt that these three methods would satisfy most users. The manual control can be used when the target position is unknown and is suited for unfamiliar environments. The coordinate-based control method is suited for controlled environments, such as assembly lines. The virtual target is suited for situations when the target position is roughly known but the user cannot measure the exact position.

9.1.2.1 Manual Joint Control

The joint control modal contains four sliders. The first three represent the shoulder joint, elbow joint, and the wrist joint respectively. The last slider represents the base angle of rotation. Each slider uses a HTML input element with min and max values matching the actuator they represent. When a change is registered the “updateVal” function sets their variable using React’s useState hook API and updates the virtual model.

```
function updateVal(command) {
    switch (command.value) {
      case "val1":
        setVal1(command.angle);
        modelState.updateModelState({
          joint: "joint1",
          angle: (command.angle / 4 + 90) * (Math.PI / 180),
        });
        break;
    }
}


<div className={classes.sliders}>
    <h3 className={classes.jointLabel}>J1</h3>
        <input
            type="range"
            min="0"
            max="180"
            defaultValue={val1}
            id="typeinp"
            className={classes.slider}
            onChange={(event) =>
              updateVal({ angle: event.target.value, value: "val1" })
            }
          ></input>
          <p className={classes.value}>{val1}</p>
</div>
```

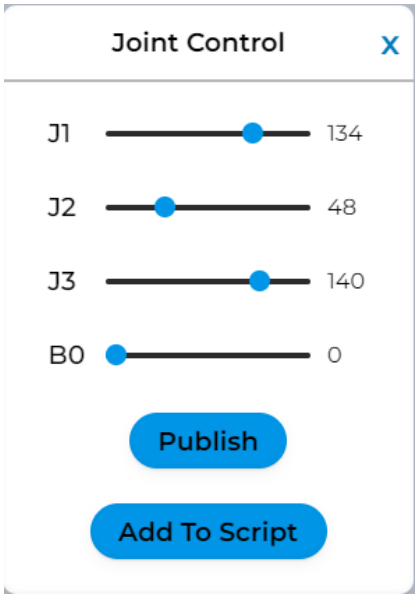


Figure 9-1-4 Slider Control Modal

When the user wants to publish the slider values to Ubidots the “publishActionHandler” function creates an object containing each slider value and sends them to the backend through a NextJS API route.

```
//slider-control-modal.js
async function publishActionHandler() {
  const action = {
    servo1: val1,
    servo2: val2,
    servo3: val3,
    servo5: val5,
  }

  console.log('Action:' + JSON.stringify(action))
  await globalCtx.publishAngles(action)
}
```

```
//Global Context file
async function publishAnglesHandler(action) {
    const response = await fetch("/api/publish-slider", {
      method: "POST",
      body: JSON.stringify(action),
      headers: {
        "Content-Type": "application/json",
      },
    });
    const data = await response.json();
    console.log(data);
  }
```

```
//publish-slider.js (API route)
async function handler(req, res) {
  console.log('Publishing Data to Ubidots: ' +  JSON.stringify(req.body))
  const response = await fetch('http://localhost:8080/publish-slider', {
    method: 'POST',
    body: JSON.stringify(req.body),
    headers: {
      'Content-Type': 'application/json',
    },
  })
  const data = await response.json()
  console.log(JSON.stringify(data))
  res.json(data)
}

export default handler
```

9.1.2.2 Coordinate Control

The coordinate control system follows a very similar pattern to the slider control. The modal features three pairs of arrows that will increment/ decrement their corresponding values when clicked. When the user clicks publish the x, y, and z values are put into an object and sent to the backend through API routes in the same way as the slider values. When the backend receives the coordinates, they are processed through inverse kinematics calculations to convert the x, y, and z values into servo angles. This process will be covered in detail in the Inverse Kinematics section.

```
//end-effector-control-modal.js
<div className={classes.sliders}>
  <div
    className={classes.triangleLeft}
    onClick={() => (val1 > -20 ? setVal1(val1 - 1) : setVal1(val1))}
  ></div>
  <p className={classes.value}>X: {val1}</p>
</div>
```

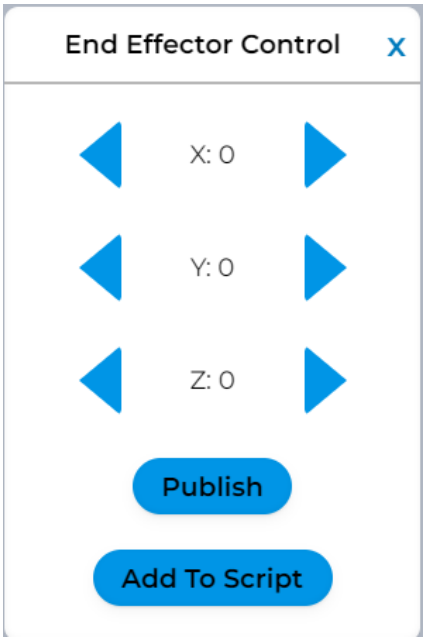


Figure 9-1-5 End Effector Control Modal

9.1.2.3 Virtual Target Block

The virtual target block is rendered using a library called React Three Fiber (R3F). This library will be covered in detail in the next sub-section, but I'll give a high-level overview of how I get the coordinates of the target block.

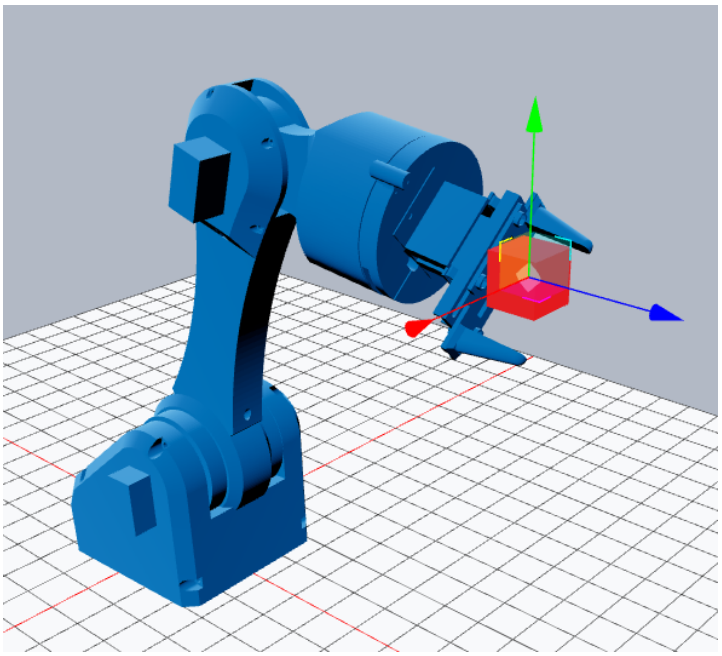


Figure 9-1-6 Virtual Target Block

Every mesh in R3F has a position attribute which is an array of three numbers. The three numbers represent the x, y, and z position of the mesh relative to the scene. I have wrapped the target cube mesh in a "TransformControls" element from the React Three Drei (R3D) library. TransformControls provides the user with a draggable gizmo to translate the cube.

```
//test-cube.js
<TransformControls
  onMouseUp={setBoxPositionHandler}
  translationSnap={0.5}
  position={[0, 4, 10]}
>
  <group position={[0, -3, -5]}>
    <mesh position={props.position} ref={target}>
      <boxGeometry args={props.size} />
      <meshStandardMaterial color={props.color} />
    </mesh>
  </group>
</TransformControls>
```

When the user releases the mouse click after dragging the TransformControls gizmo the "setBoxPositionHandler" gets the current position of the cube and offsets the y and z coordinates by 3 and 5. I had to offset these positions to align the TransformControls gizmo with the starting position of the target cube.

```
//test-cube.js
const target = useRef()

function setBoxPositionHandler() {
  target.current.getWorldPosition(position)
  let realPosition = { x: position.x, y: position.y + 3, z: position.z + 5 }
  modelCtx.setTargetBlockPosition(realPosition)
}
```

Now that I have the position of the target cube they can be published in the exact same way as the coordinates.

9.1.3 Virtual Robot Model

The virtual robot model is rendered using React Three Fiber (R3F) and React Three Drei (R3D). R3F is a React renderer for ThreeJS, a popular JavaScript library for rendering 3D graphics in the browser. It follows JSX component structure of React while offering all features available in ThreeJS. R3D is a collection of useful helpers and abstractions for R3F. It contains new camera types, controls, gizmos, and shaders.

9.1.3.1 Canvas Setup

The Canvas object is where the R3F scene is defined. In my project it wraps three forms of lighting, the ground plane, eleven meshes, and the OrbitControl tool which allows the user to control the camera with their mouse.

9.1.3.2 Lighting

Lighting is important when creating the scene. A bad lighting setup will make the user want to avoid your scene. I used two instances of directional light, pointing at either side of the robot. The ambient light is used to brighten the harsh shadows left by the directional light.

```
//test-scene.js
<directionalLight
  position={[-8, 12, 8]}
  intensity={5}
  castShadow
  color={"#3da0fd"}
/>
<directionalLight
  position={[10, 12, -10]}
  intensity={4}
  castShadow
  color={"#3da0fd"}
/>
<ambientLight intensity={10} />
```

9.1.3.3 The Model

Each element of the robot is imported separately as a GLTF file and positioned together.

```
//bone2.js
const gltf = useLoader(GLTFLoader, "/assets/models/Bone2.gltf");
// ...
useGLTF.preload("/assets/models/Bone2.gltf");
```

I use React's Context API to store the state of each mesh and connect the joint rotation to the sliders. Whenever the slider values change, they are immediately fed into the rotation attribute of each mesh.

```
//bone2.js
<group
  ref={groupRef}
  {...props}
  dispose={null}
  rotation={modelState.theModelData.bone2Rotation}
  position={[0, 0.4, -7.8]}
>
  <primitive
    object={gltf.scene}
    position={[0, -2.86, 15.035]}
    rotation={[
      190 * (Math.PI / 180),
      0 * (Math.PI / 180),
      90 * (Math.PI / 180),
    ]}
    scale={[0.03, 0.03, 0.03]}
  />
</group>
```

9.1.4 Script Creation

Script creation is the final aspect of my web application. In this context a script is a sequence of movements that the robot will execute. Scripts are a necessary feature in any decent robot control application.

9.1.4.1 Script UI

My script UI was inspired by the timeline view in Fusion 360 which I noticed while designing the robot. I found the timeline view to be extremely intuitive and easy to navigate. It takes no prior experience to understand the order of actions is flowing left to right. If there is confusion the arrows and element labels will guide the user.

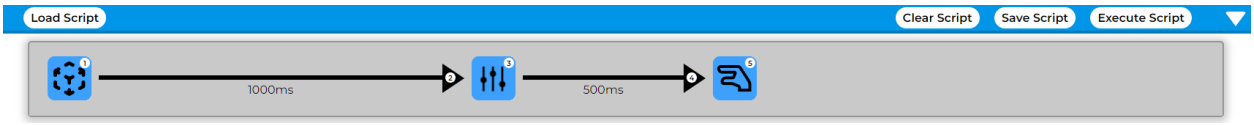


Figure 9-1-7 Web App Script UI

Figure 9-1-8 Fusion 360 Timeline UI

The data in each element of the script can be viewed in a pop-up modal. Each element can be deleted or have its position in the script changes. This is a very useful feature and prevents the user from making a whole new script if a mistake is made.

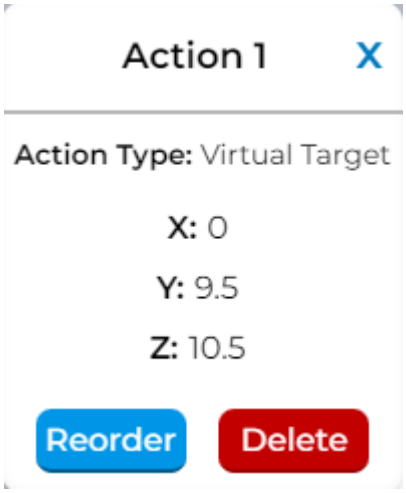


Figure 9-1-9 Script Element Modal

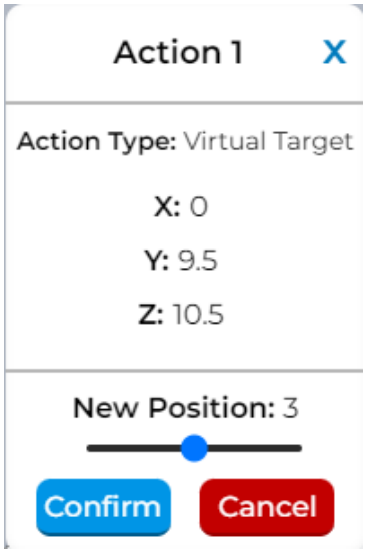


Figure 9-1-10 Script Element Reorder Menu

9.1.4.2 Script Structure

Each action is a JSON object with a data, actionType, index, and action field. This is enough data to be able to process each of the three action types and delete or reorder them. The script itself is an array of these action objects. Here is the example script shown above, visualised in a tree structure.

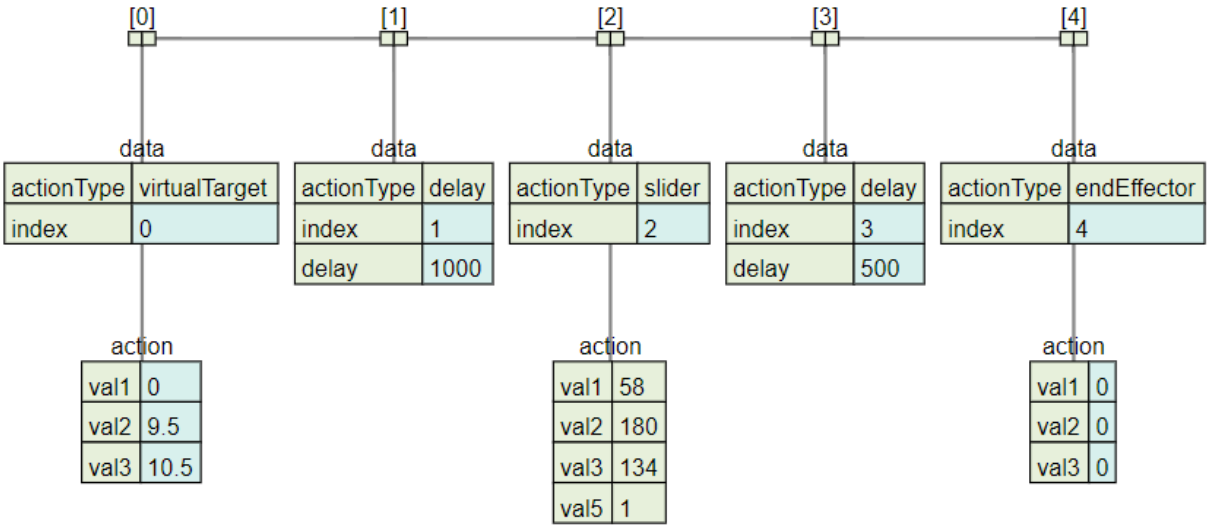


Figure 9-1-11 Script JSON Structure

When the backend receives the script array, each element is processed and sent to Ubidots.

9.2 Java Spring Boot Backend

Spring Boot is an open-source Java framework used for programming production-grade Spring based applications. In Spring Boot applications, annotations are used to provide metadata to Java code. Annotations are a concise way to express configurations and keep the code maintainable and readable.

9.2.1 Communication with Ubidots

When the user publishes their action a POST request is sent to the backend. This is received by the PublishController class which is annotated by the "@RestController" tag. A rest controller is responsible for handling incoming HTTP requests and producing HTTP responses.

```
@PostMapping("/publish-slider")
    public SliderData publishSlider(@RequestBody SliderData sliderData) {
        System.out.println("Publishing Slider data: " + sliderData);
        sliderDataService.publishSliderData(sliderData);
        return sliderData;
    }
```

When the endpoint receives a request, it passes the request body content to a service class. Service classes encapsulate business logic. These classes are where I process scripts and run inverse kinematics calculations.

```
//SliderDataService Class
public void publishSliderData(SliderData sliderData){
    System.out.println("Slider Data Service: "+ sliderData);
    publishServiceClient.publishSlider(sliderData);
}
```

In the above code I am only passing the slider data to the Feign client. A feign client simplifies the process of making HTTP requests to other RESTful services. I am using this Feign client to send a POST request to my Ubidots device.

```
//Feign Client
@FeignClient(name = "publish-service", url =
"https://industrial.api.ubidots.com/api/v1.6/devices/robot/", configuration = {FeignConfig.class})
public interface PublishServiceClient {
    @PostMapping("")
    void publishSlider(@RequestBody SliderData sliderData);

    @PostMapping("")
    void publishGripper(@RequestBody GripperData gripperData);

    @PostMapping("")
    void publishToolStatus(@RequestBody ToolStatus toolStatus);
}
```

Finally, before the request is sent to Ubidots an interceptor adds my API token to the request for authorisation purposes.

```
//Auth Interceptor Class
public class AuthInterceptor implements RequestInterceptor {
    @Value("${my.apitoken}")
    private String apiToken;
    @Override
    public void apply(RequestTemplate requestTemplate){
        requestTemplate.header("X-Auth-Token", apiToken);
        requestTemplate.header("Content-Type", "application/json");
    }
}
```

9.2.2 Database connection

Interacting with MongoDB follows a similar process to publishing. First the request is received by the interceptor and relevant data is passed to a service class.

```
//Script Controller Class
@PostMapping("/save-script")
public void saveScript(@RequestBody String script) throws IOException {
    if("[]".equals(script)){
        System.out.println("Script is empty");
    }
    else{
        scriptService.saveScriptToMongo(script);
    }
}

@GetMapping("fetch-scripts")
public List<ScriptString> fetchScripts(){
    return scriptService.getAllScripts();
}

@PostMapping("/delete-script")
public String deleteScript(@RequestBody String name){
    System.out.println("Name: " + name);
    scriptService.deleteScriptByName(name);
    return name;
}
```

When saving a script, the JSON array containing the script actions gets mapped to my model classes. I had to ensure that the structure of the model classes exactly matched the JSON array, this was a complicated process.

```
//Script Service Class
public void saveScriptToMongo(String json) throws IOException {
  ObjectMapper objectMapper = new ObjectMapper();
  ScriptData scriptString = objectMapper.readValue(json, ScriptData.class);
  if (scriptString.getAction().isEmpty()) {
      System.out.println("Script is empty");
  }
  else {
      scriptString.setName(scriptString.getName());
      scriptString.setAction(scriptString.getAction());
      scriptRepo.save(scriptString);
  }
}
```

When the JSON has been mapped to Java objects it is then passed to the “ScriptRepo” interface which communicates with MongoDB.

```
//Script Repo Interface
public interface ActionScriptRepo extends MongoRepository<ScriptData, String> {
}
```

9.3 ESP32 Firmware

The firmware running on the ESP32 uses libraries. “UbidotsEsp32Mqtt” is used to establish and maintain the connection to Ubidots. “ESP32Servo” is used to control servos and makes use of the ESP32 PWM timers. “Stepper” allows the ESP32 to control a variety of stepper motor, both unipolar and bipolar.

9.3.1 Communication with Ubidots

Esp32-mqtt is an open-source library that can be read on GitHub. My code begins by calling the Ubidots constructor and passing it my API token.

```
Ubidots ubidots(UBIDOTS_TOKEN);
```

Next, we move into void setup where I call four Ubidots member functions to initialise the connection.

```
void setup() {
  Serial.begin(115200);
  pinMode(relay, OUTPUT);
  ubidots.connectToWifi(WIFI_SSID, WIFI_PASS);
  ubidots.setCallback(callback);
  ubidots.setup();
  ubidots.reconnect();
  subscribe_to_vars(var_labels, DIMENSION_OF(var_labels));
}
```

“connectToWifi” connects to an available WiFi network using WAP2.

“setCallBack” sets the callback function that will process the data incoming from the subscribed topics.

“setup” configures all the necessary settings in the Ubidots instance to connect to the broker.

“reconnect” is a blocking function that will attempt to open a socket to the broker.

Void loop maintains the connection with the Ubidots broker.

```
void loop() {
  if (!ubidots.connected()) {
    ubidots.reconnect();
    subscribe_to_vars(var_labels, DIMENSION_OF(var_labels));
  }
  ubidots.loop();
}
```

If the ESP32 disconnects it will attempt to reconnect and subscribe to the topics defined in "var_labels". Otherwise, the loop method will continue to run. The loop method maintains the socket connection and periodically send a keep alive command.

When subscribed topics are updated the callback function is triggered. This function processes the payload, extracts the topic values, and adds them to an array of the most recent values. The array of most recent values is then looped through, and the values are written to the actuators.

```
//Callback function
void callback(char *topic, byte *payload, unsigned int length) {
  char *topic_cpy = strdup(topic);
  char *payload_str = (char *)malloc(length + sizeof(""));
  char *topic_item = strtok(topic_cpy, "/");
  char *label = NULL;
  float value = NAN;
  size_t index = DIMENSION_OF(var_labels);
  size_t i;

  memcpy(payload_str, payload, length);
  payload_str[length] = '\0';

  while ((NULL != topic_item) && (NULL == label)) {
    for (i = 0; i < DIMENSION_OF(var_labels); i++) {
      if (0 == strcmp(var_labels[i], topic_item)) {
        label = topic_item;
        value = atof(payload_str);
        index = i;
        break;
      }
    }
    topic_item = strtok(NULL, "/");
  }
```

```
  if (index < DIMENSION_OF(var_labels)) {
    var_last_values[index] = value;
    switch (index) {
      case 0:
        Serial.print("Servo 1 val: ");
        Serial.println(var_last_values[index]);
        moveServo(1, var_last_values[index]);
        break;
      case 1:
        Serial.print("Servo 2 val: ");
        Serial.println(var_last_values[index]);
        moveServo(2, var_last_values[index]);
        break;
      case 2:
        Serial.print("Servo 3 val: ");
        Serial.println(var_last_values[index]);
        moveServo(3, var_last_values[index]);
        break;
      case 3:
        Serial.print("Servo 4 val: ");
        Serial.println(var_last_values[index]);
        moveServo(4, var_last_values[index]);
        break;
      case 4:
        Serial.print("Stepper val: ");
        Serial.println(var_last_values[index]);
        moveServo(5, var_last_values[index]);
    }
  }
  free(topic_cpy);
  free(payload_str);
}
```

9.3.2 Actuator Control

9.3.2.1 Servo Control

The servo control code begins by initialising servo objects.

```
Servo shoulderServoStrong;
int shoulderStrongPin = 18;
shoulderServoStrong.attach(shoulderStrongPin, 1000, 2000);
int prevAngleShoudlerStrong = 0;
```

I then attach the servo object to a pin and define the min and max pulse width ranges. I also initialise a variable to store the previous angle of the servo, this is used to create smooth movement between the previous servo angle and new angle.

When a new angle is received, I constrain the value to be greater than 9 and less than 171. This is done to ensure I never hit the mechanical stops and damage the servo.

```
switch (servo) {
  case 1:
    if (angle_value ⩾ 170) {
      angle_value = 170;
    } else if (angle_value ⩽ 10) {
      angle_value = 10;
    }
    angle_value = 180 - angle_value;
    Serial.println(angle_value);
    if (angle_value > prevAngleShoudlerStrong) {
      for (int i = prevAngleShoudlerStrong; i < angle_value; i += 1) {
        shoulderServoStrong.write(i);
        shoulderServoWeak.write(i * 0.66);
        Serial.print(i);
        delay(20);
      }
    } else {
      for (int i = prevAngleShoudlerStrong; i > angle_value; i -= 1) {
        shoulderServoStrong.write(i);
        shoulderServoWeak.write(i * 0.66);
        Serial.print(i);
        delay(20);
      }
    }
    prevAngleShoudlerStrong = angle_value;
    break;
}
```

Next, I check if the new value is greater than or less than the previous value, this determines whether I increment or decrement the angle from the previous value. Each loop I write the new angle value to the servo and delay for 20ms, this is crucial for achieving smooth and stable operation of the servo while also increasing the lifespan of the servo. When the servo reaches the desired angle, I overwrite the previous angle with the new angle.

9.3.2.2 Stepper Motor

The stepper code is very simple. I begin by creating a stepper object with 200 steps per revolution and set the four pins connected to the stepper driver.

```
const int stepsPerRevolution = 200;
Stepper myStepper = Stepper(stepsPerRevolution, 32, 25, 33, 26);
```

Stepper motors do not have internal feedback circuits like servos; therefore, they have no knowledge of their current position. I need to keep track of the stepper's angle of rotation in code. The previous base angle is initialised as 0. The number of steps to reach the desired angle is equal to the new angle – the previous base angle. Finally, I set the speed of rotation (in RPM) and call the step function with the desired number of steps.

```
case 5:
  if(angle_value == prevBaseAngle){
    break;
  }
  int desiredAngle = angle_value - prevBaseAngle;
  Serial.println(desiredAngle);
  myStepper.setSpeed(5);
  myStepper.step(desiredAngle);
  prevBaseAngle = angle_value;
  break;
```

10 Kinematics

Kinematics is the branch of physics concerned with the geometrically possible motion of a body or system of bodies without consideration of the forces involved [9]. In this section I will only be concerned with kinematics in relation to robotic arms with three degrees of freedom.

10.1 Forward Kinematics vs. Inverse Kinematics

10.1.1 Forward Kinematics

Forward kinematics refers to the process of calculating the position of the end effector, given known joint angles. This process for a 3DOF robot is extremely straightforward and there is only one possible answer, if the joints are in set angles the end effector can only be in one position.

The forward kinematics of a 2-link planar can be calculated quite easily by using trigonometric rules.

In this example I have two bones and two joints.

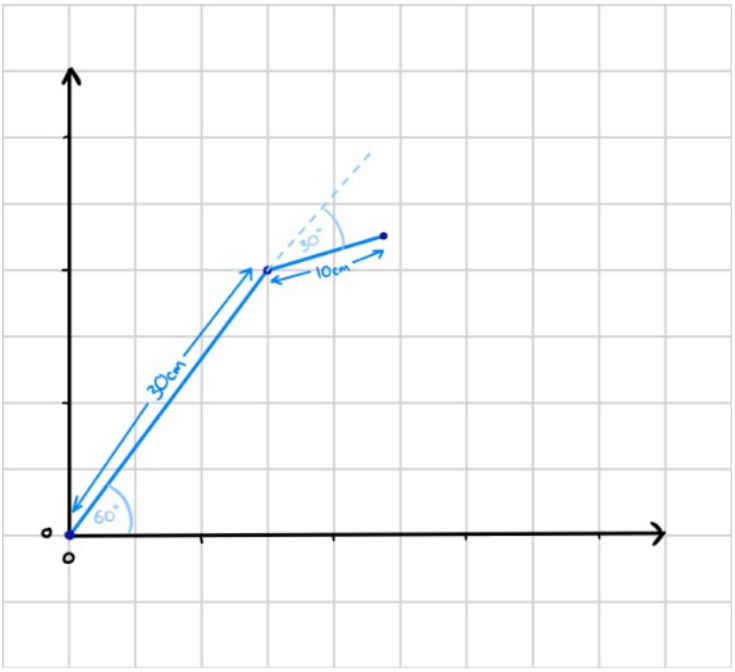
The base point is (0, 0).

Bone 1 is 30cm in length.

Bone 2 is 10cm in length.

Joint 1 is 60°.

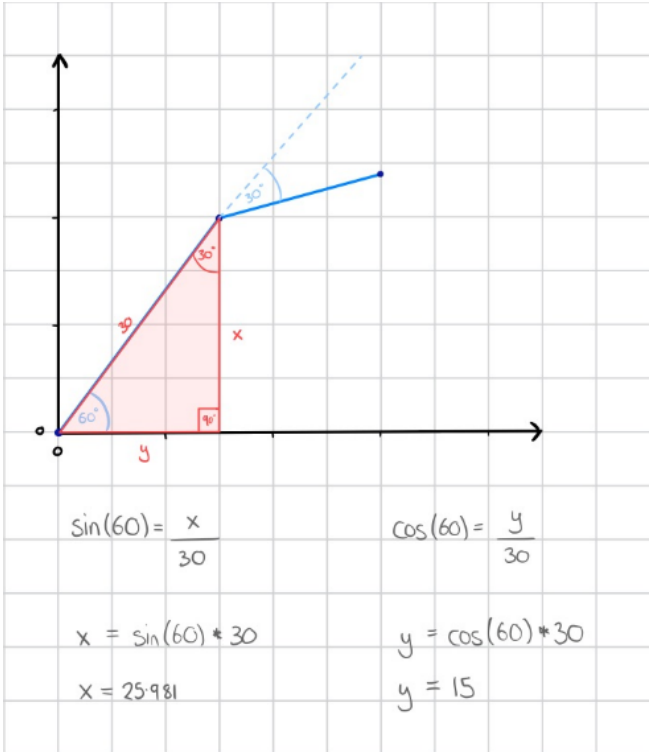
Joint 2 is -30° relative to joint 1.



The first step is to turn bone 1 into a right-angle triangle.

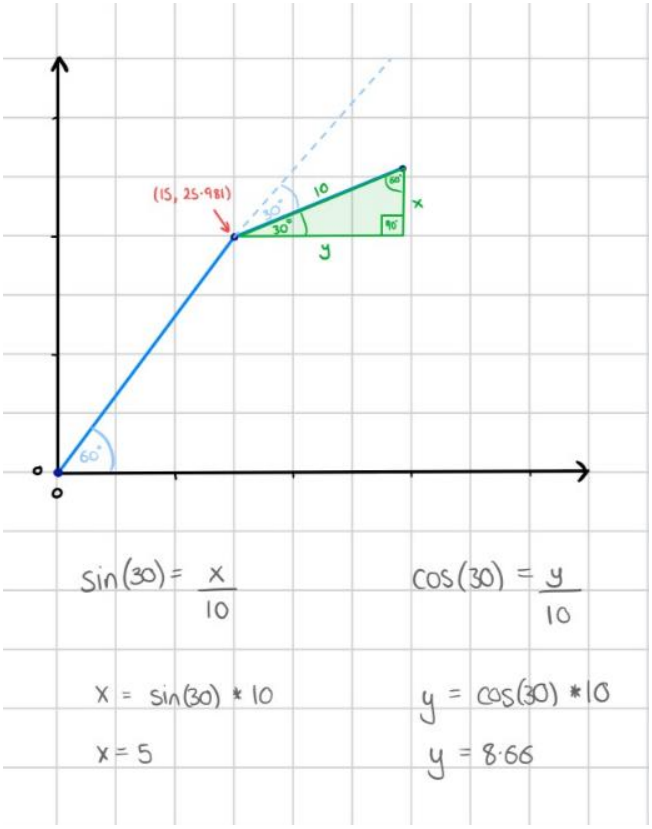
Then we can use the sin and cos rules to calculate the two unknown lengths.

We're starting from point (0, 0) so the location of joint 2 will be (0+y, 0+x)

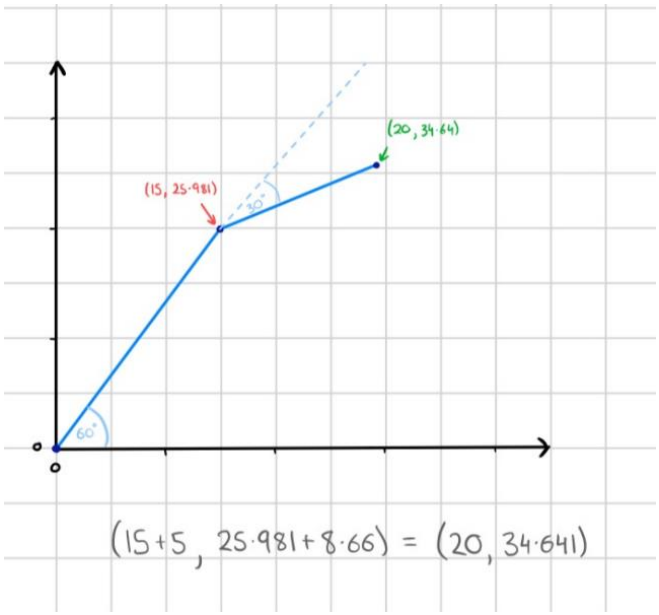


Now we repeat the previous steps to find the unknown lengths of triangle 2.

This time the origin points of triangle 2 are the coordinates of joint 2.



We calculate the end effector position by adding triangle 2's unknown lengths to the coordinates of joint 2.



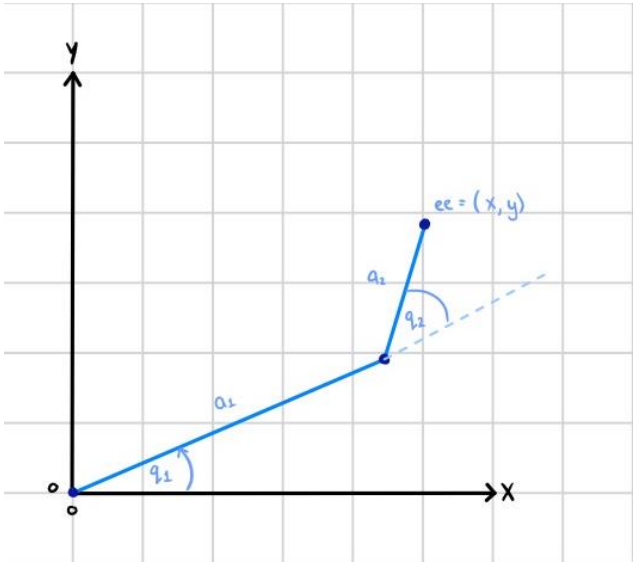
10.1.2 Inverse Kinematics

Inverse kinematics requires some more complex calculations. In this case the end effector position is known, and I need to calculate the correct joint configuration.

In the following example I will explain one of the many methods of calculation that I explored. This example only returns one joint configuration. It will fail if the end effector coordinates are inside range but impossible to reach. If the end effector coordinates are outside range, it will return joint angles that will point at the end effector coordinates in one straight line.

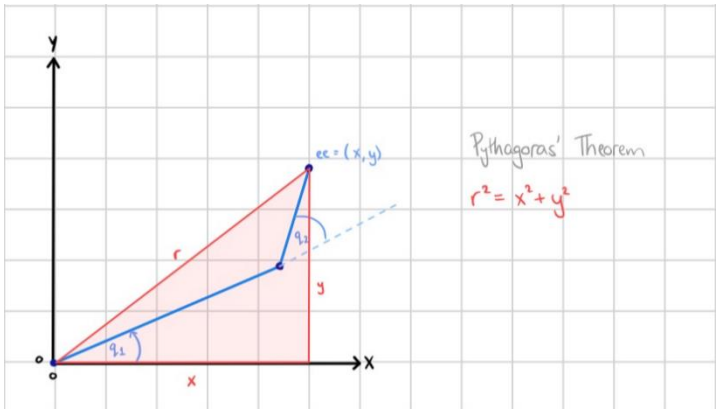
This will be the configuration for the following example.

This example will create the general equation that I have implemented in code.



I begin by drawing a right-angle triangle with points: (0, 0), (x, y), and (x, 0).

Length r is calculated using Pythagoras' theorem.

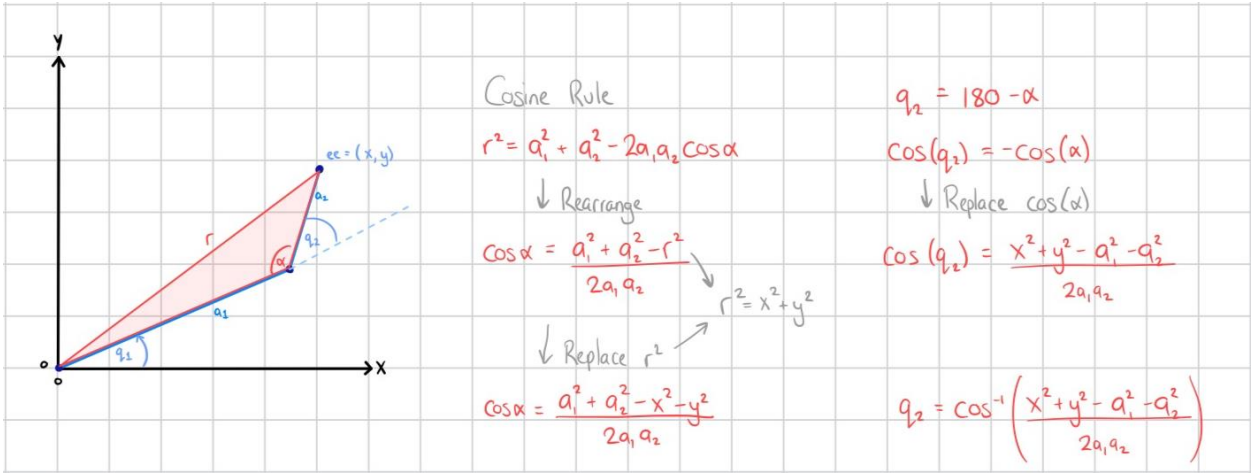


Now that the length r is known, I want to calculate the angle q2.

q2 can be calculated by finding the inner angle (α) using the cosine rule.

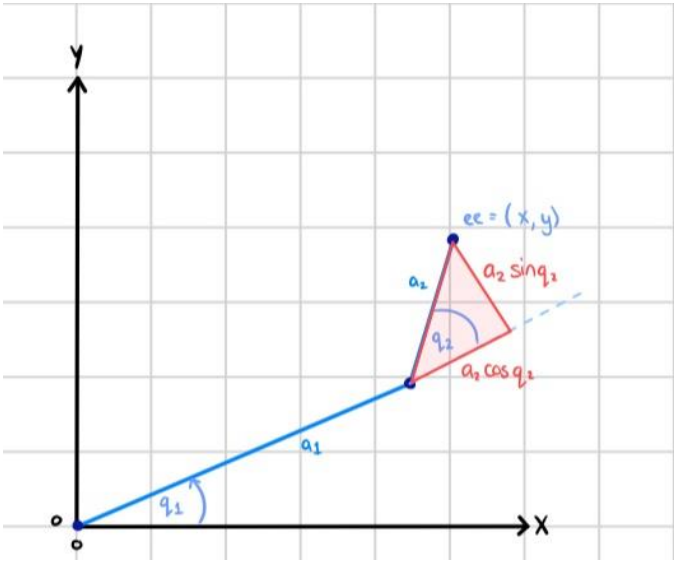
q2 and α exist on a straight line so q2 + α = 180°. Therefore q2 = 180° - α.

I can now solve for q2 by replacing α in the cosine rule.



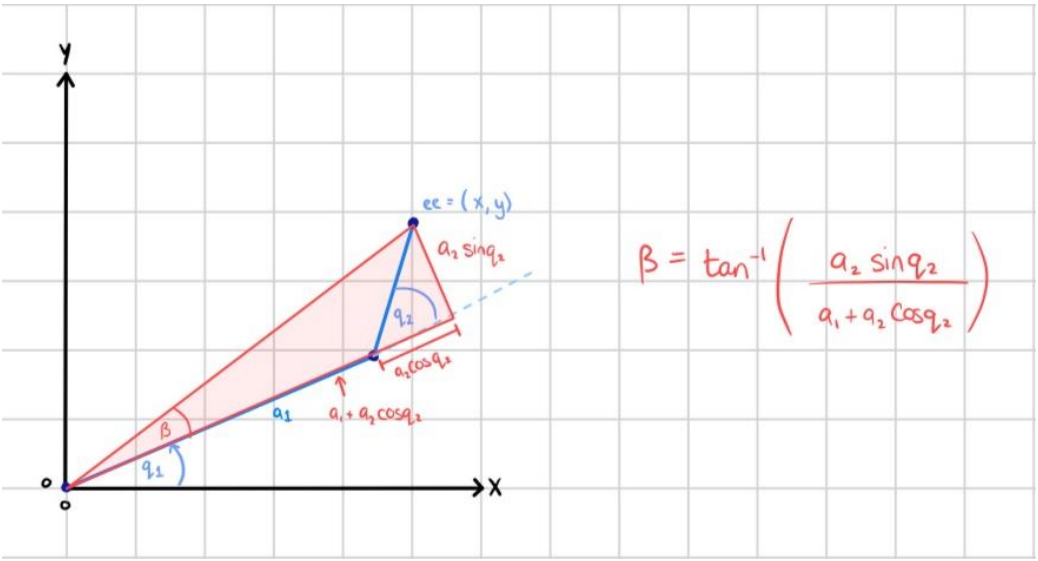
I will now focus on calculating q1.

I will begin by drawing a right-angle triangle and calculating the unknown lengths using the sin and cos rule.

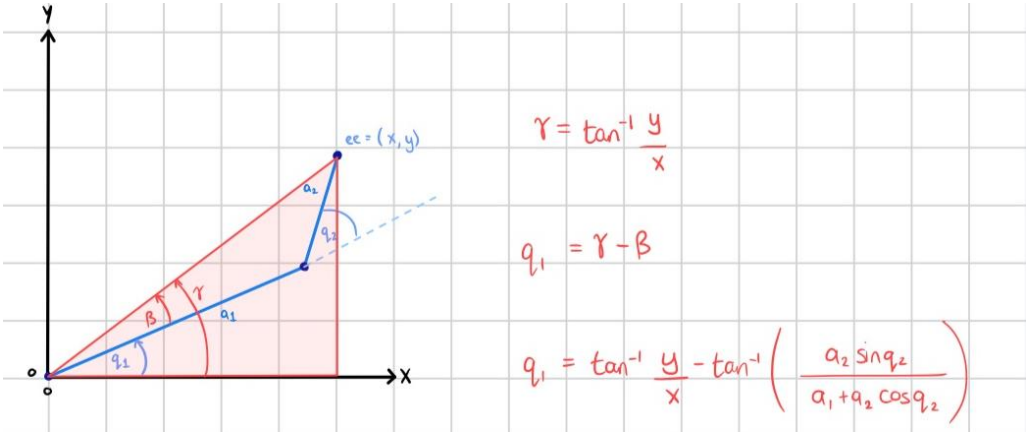


I can use the newly calculated lengths to calculate β.

This time I use a variation of the tan rule.

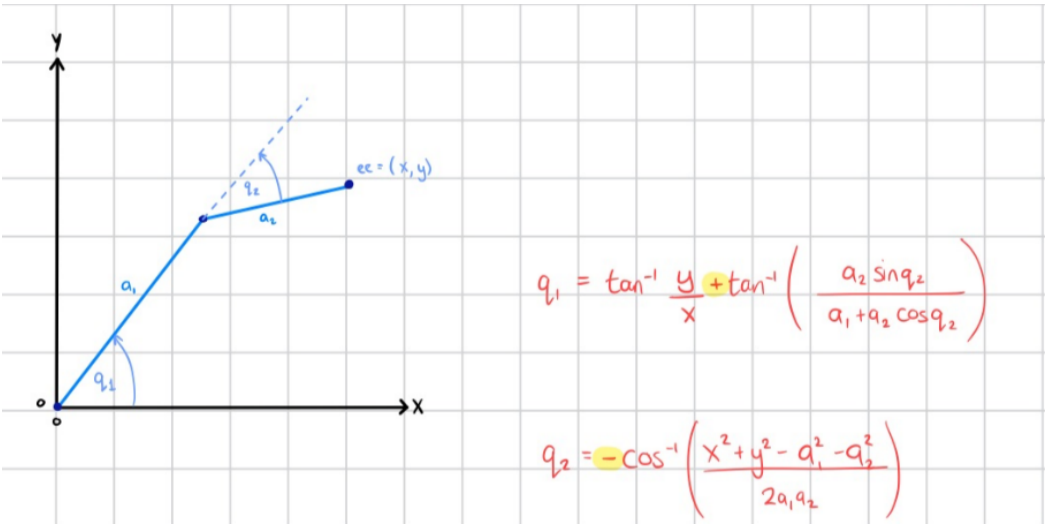


Finally, q1 can be calculation by constructing the original triangle, using the tan rule to find γ, then subtracting β from γ.



We now have a general equation for both joint angles: q1 and q2.

A minor adjustment to the equations can be made to flip the direction of bone 2. This is a necessary adjustment due to the constraints of the physical robot.



10.2 Inverse Kinematics Implementation

In my experience, implementing inverse kinematics is much more difficult than simulating the movements. When simulating you don't need to consider gear backlash, servo resolution, or range of motion limits. Inverse kinematics also requires extreme precision when measuring bone 1 and bone 2, discrepancies between the real lengths and simulated lengths will cause small errors in the calculation that will be exacerbated due to the distance between the end effector and shoulder joint.

10.2.1 Code

X, Y, and Z coordinates sent from the frontend are received by the "publishCoordinates" function in the publish controller class.

A coordinates object is created and passed to the "publishCoordinateValues" function.

```
@PostMapping("/publish-coordinates")
public Coordinates publishCoordinates(@RequestBody Coordinates coordinates) {
    System.out.println("Publishing Coordinate data: " + coordinates);
    coordinateService.publishCoordinateValues(coordinates);
    return coordinates;
}
```

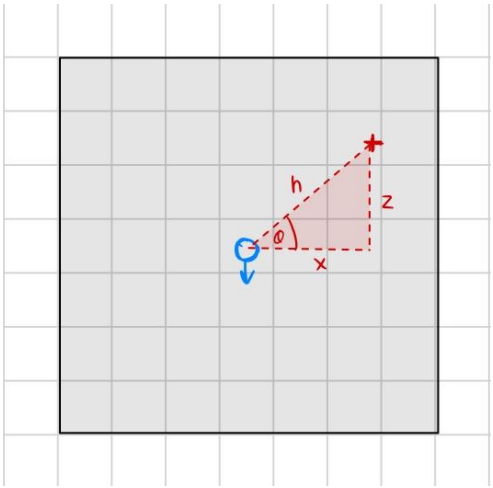
An inverseKinematics object is created containing the bone lengths, X, Y, and Z coordinates, and a placeholder for the hypotenuse value.

The hypotenuse value is necessary because I am now operating on three axes.

```java
//Coordinate Service
public void publishCoordinateValues(Coordinates coordinates){
    InverseKinematics ik = new InverseKinematics(26, 25, coordinates.getX()+5, coordinates.getY() - 15,coordinates.getZ()+5, 0);
    int baseAngle = ik.calcBaseAngle();
    double hypotenuse = sqrt(((coordinates.getZ() * coordinates.getZ()) + (coordinates.getX() * coordinates.getX())));
    ik.setHyp(hypotenuse);
    CoordsAngles ca = calcAngles(ik);
    ca.setServo5(baseAngle);
    if(ca.getServo1() != 0 && ca.getServo2() != 0){
        publishServiceClient.publishCoords(ca);
    }
}
```

Next, I need to calculate the “baseAngle” which is the angle of rotation needed to align the robot with the target point.

I do this by drawing a right-angle triangle with lengths X and Z. H can be calculate using Pythagoras’ theorem. Theta can be calculated using the sine rule.




```java
//Inverse Kinematics Class
public int calcBaseAngle(){
    int baseAngle = 0;
    double hypotenuse = 0;
    if(x <= 0 && z <=0){
        hypotenuse = sqrt(((z*z) + (x*x)));
        baseAngle = (int) (((asin(abs(x)/hypotenuse) * (180/PI)) / 1.8) + 100);
    }
    else if (x <= 0 && z >=0){
        setZ(getZ()-10);
        hypotenuse = sqrt(((z*z) + (x*x)));
        baseAngle = (int) (((asin(abs(z)/hypotenuse) * (180/PI)) / 1.8) + 150);
    }
    else if (x >= 0 && z >=0){
        setZ(getZ()-10);
        setX(getX()-10);
        hypotenuse = sqrt(((z*z) + (x*x)));
        baseAngle = (int) (50 -((asin(abs(z)/hypotenuse) * (180/PI)) / 1.8));
    }
    else if (x >= 0 && z <=0){
        hypotenuse = sqrt(((z*z) + (x*x)));
        baseAngle = (int) (50 + ((asin(abs(z)/hypotenuse) * (180/PI)) / 1.8));
    }
    return baseAngle;
}
```

Now the robot is aligned with the target point. The distance between the robot's origin points and the target point is once again the hypotenuse. This distance is used in the final inverse kinematics calculation.

The "negPosAngleCalc" function uses the general equation explained in the previous section.

The servo's range of motion is less than 180° so I added extra steps to map the calculated joint angles to the true servo angles.

```
//Inverse Kinematics Class
public CoordsAngles negPosAngleCalc(){
        CoordsAngles ca = new CoordsAngles();
        double q1, q2;
        DecimalFormat df = new DecimalFormat("0.00");
        q2 = -acos(((hyp*hyp)+(y*y)-(a1*a1)-(a2*a2))/(2*a1*a2));
        q1 = (atan(y/hyp) - (atan((a2*sin(q2))/(a1+(a2*cos(q2))))));
        ca.setServo1((int) (100 - ((q1 * 180/PI) * 0.445)));
        ca.setServo2((int) (130 - ((q2 * 180/PI + (q1 * 180/PI))*2.5)));
        boolean result = forwardKinematics(parseDouble(df.format((q1 * 180/PI))), parseDouble(df.format((q2 * 180/PI +
(q1 * 180/PI)))), a1, a2, parseDouble(df.format(hyp)), parseDouble(df.format(y)));
        return ca;
    }
```

11 Ethics

Several ethical considerations should be made when developing a robotic arm. These considerations should ensure the arm is used responsibly, beneficially, and does not cause harm to anyone.

11.1 Safety

Ensuring the safety of the user is the number one priority. During the early stages of this project, I conducted research into the safety standards of industrial robotic arms. The three laws of robotics developed by science-fiction writer Isaac Asimov int 1942 state that "(1) a robot may not injure a human being or, through inaction, allow a human being to come to harm; (2) a robot must obey the orders given it by human beings except where such orders would conflict with the First Law; (3) a robot must protect its own existence as long as such protection does not conflict with the First or Second Law." [10] Asimov's three rules became hugely influential in the sci-fi genre and subsequently found relevant in discussions involving technology.

I kept Asimov's three rules in mind while developing the ModuGrip. The ModuGrip is designed to be accessed remotely so I did not feel the need to incorporate a dead-man switch as the user should not be in direct contact with the device. The arm is also relatively low power so there is no real risk of injury.

I toyed with the idea of incorporating artificial intelligence in the very early stages of this project but ultimately decided against it, mostly because of the additional complexity but also because I felt it could conflict with Asimov's second rule.

11.2 Environmental Impact

Mitigating the environmental impact of producing, operating, and disposing of robotic devices is an area of utmost importance. The ModuGrip has been printed with PLA, a bioplastic derived from biomasses like corn, sugarcane, or seaweed [11]. PLA can be composted in controlled environments in less than 90 days, and its recyclable, however, it is not a perfect material. Bioplastics often compete for land with food crops and biodegrade naturally very slowly.

It is not feasible to develop a fully environmentally friendly hardware project, but it is important to make environmentally conscious decisions when we can.

12 Conclusion

This project has given me the opportunity to develop both my technical and soft skills in a variety of areas. I enhanced my hardware knowledge, full-stack development skills, project management skills, and most of all, my problem-solving abilities. There were many setbacks along the way, but I also learned valuable lessons when overcoming these challenges. I am satisfied with my performance, and I believe that I gave it my maximum effort. In the end I accomplished my goals of designing and producing a functional, demonstrable 3-DOF robotic arm with a custom control web application.

13 Appendix

DS3225MG_Datasheet.pdf

900-00005-Standard-Servo-Product-Do

Parallax_900-00005_Dimensions.pdf

rs-180-5278_Datasheet.pdf

14 References

[1] "ez-robot," [Online]. Available: https://www.ez-robot.com/learn-robotics-introduction-to-servo-motors.html. [Accessed 23 04 2024].

[2] "Sparkfun," [Online]. Available: https://www.sparkfun.com/servos. [Accessed 24 04 2024].

[3] "drones-spare-parts," [Online]. Available: https://www.drones-spare-parts.com/it-ie/products/servomotore-digitale-ds3240-6v-40kg-180-gradi-robot-baja-car-1-10?redirected=true. [Accessed 29 04 2024].

[4] "Banggood," [Online]. Available: https://myosuploads3.banggood.com/products/20220907/20220907032153DS3225datasheet.pdf. [Accessed 29 04 2024].

[5] "Parallax," [Online]. Available: https://www.parallax.com/package/parallax-standard-servo-downloads/. [Accessed 29 04 2024].

[6] "DigiKey," [Online]. Available: https://www.digikey.ie/en/blog/the-basics-of-stepper-motors. [Accessed 29 04 2024].

[7] "RandomNerdTutorials," [Online]. Available: https://randomnerdtutorials.com/getting-started-with-esp32/#ESP32-specs. [Accessed 29 04 2024].

[8] "Sharvi Electronics," [Online]. Available: https://sharvielectronics.com/product/kk-p20-15-12vdc-3kg-lifting-solenoid-electromagnet/. [Accessed 29 04 2024].

[9] "Britannica," [Online]. Available: https://www.britannica.com/science/kinematics. [Accessed 29 04 2024].

[10] "Britannica," [Online]. Available: https://www.britannica.com/topic/Three-Laws-of-Robotics. [Accessed 29 04 2024].

[11] "Filamentive," [Online]. Available: https://www.filamentive.com/how-sustainable-is-pla/. [Accessed 29 04 2024].

[12] "Parallax," [Online]. Available: https://www.parallax.com/product/parallax-standard-servo/. [Accessed 29 04 2024].

[13] "Parallax," [Online]. Available: https://www.parallax.com/package/servo-dimensions/. [Accessed 29 04 2024].

[14] "Radionics," [Online]. Available: https://ie.rs-online.com/web/p/stepper-motors/1805278. [Accessed 29 04 2024].

[15] "RS-Docs," [Online]. Available: https://docs.rs-online.com/ecfe/0900766b816b4d8f.pdf. [Accessed 29 04 2024].

[16] "Last Minute Engineers," [Online]. Available: https://lastminuteengineers.com/esp32-pinout-reference/. [Accessed 29 04 2024].

[17] S. Rentals. [Online]. Available: https://www.inlec.com/ie/thurlby-thander-instruments-ts3022s-power-supply. [Accessed 29 04 2024].

[18] "Switch Electronics," [Online]. Available: https://www.switchelectronics.co.uk/products/mfa-918d301-1-30-1-gearbox-and-motor-1-5-3v. [Accessed 29 04 2024].