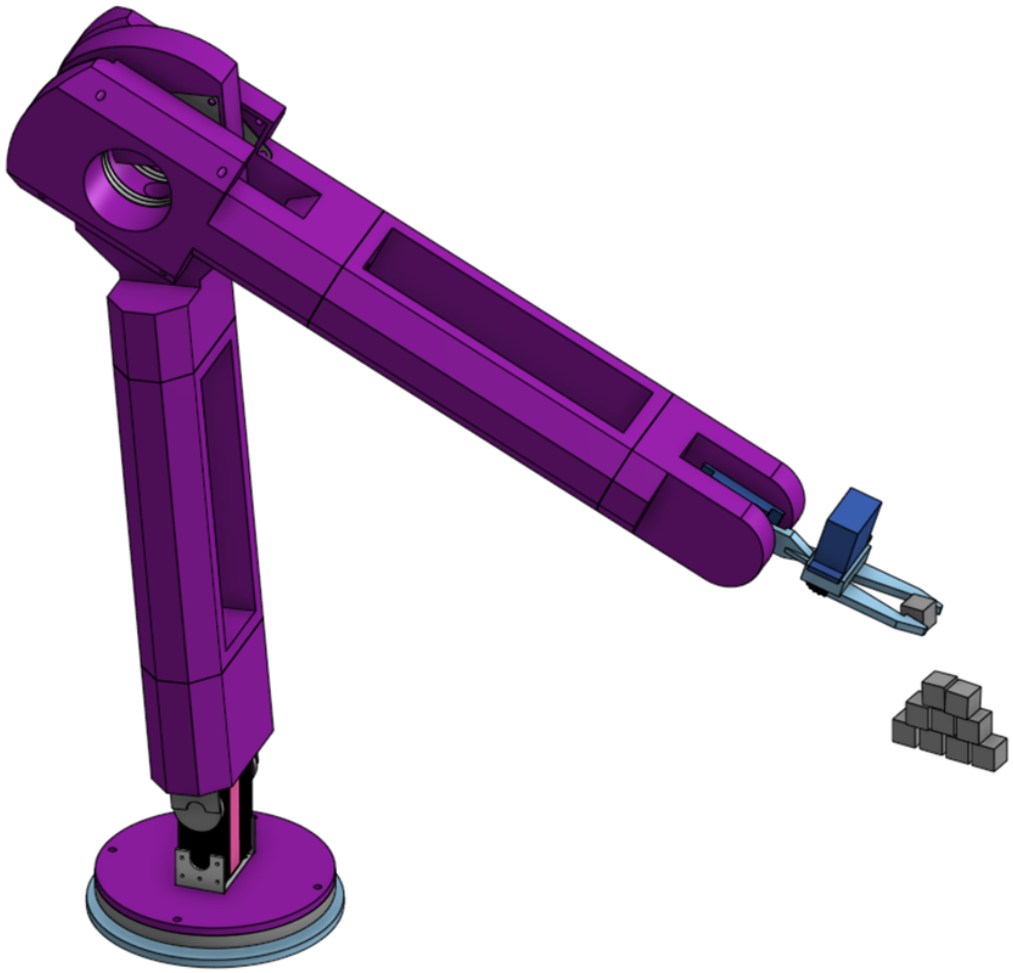
Creating a Polar Coordinate Robotic Arm for Completing Repetitive Tasks

Anton Chernyshov
Candidate Number: 7038
School Name: Tonbridge School
Centre Number: 61679

November 2025

Abstract

This project involved designing, constructing and programming a robotic arm that can move freely in 3D space. My goal was to make a robotic arm from the ground up that worked using hobby grade resources, and one that would consistently perform a task. The arm was built using 3D printed components, and controlled using custom Python software running on a Raspberry Pi microcontroller. Although a prototype was built and somewhat tested, the project's further development was halted due to time constraints encountered throughout the build.



Contents

1 Introduction

Robot arms are widely used in industry due to their precision and versatility in handling tasks. They are used to complete repetitive tasks often too hard for humans to do consistently. My goal for this project is to design and build a robotic arm using cheap and readily available components. The arm will have 5 Degrees of freedom, Base rotation, shoulder, elbow, wrist and gripper, all controlled using a polar coordinate system. I chose to do this project as this is something I have been interested in doing for a long time, and I want to build something that mixes my passion for programming and engineering. Unfortunately, due to time constraints, the final arm was not built in time for the EPQ submission, but I am planning to continue working on it over the summer. This write-up describes my process from start to finish, the design, research, construction, coding and testing of the arm, whilst also explaining the reasoning behind key decisions supported by relevant sources.

2 Research and Design

2.1 Research

My research started by reading the society of robots guide on how to build a robotic arm[11], and this gave me the base knowledge on what my arm should look like, and how it should be built. I knew the general objectives I wanted it to consist of, but I needed to create a specification.

2.2 Specification

- Minimum maximum extension of 700mm
- Have at least 4 degrees of freedom
- Minimum payload mass of 150g, no greater than 500g
- Must have an API or library that can be used to queue commands.
- Must also be controllable in real time, either through a controller or console commands.
- Must not be more expensive than £200.
- Must be able to:
  - Build a tower from wooden blocks, and then un-stack it.
  - Move a pile of blocks from one place to another.

With a specification in mind I had to start thinking of more specifics, such as what kind of end effector I want, and how I would control it. I decided to leave the maths and coding to after the arm was built, as it would be much easier to test code on a working arm than writing an accurate simulator to test the code on.

2.3 Design

2.3.1 Modeling

To start modelling the arm, I needed to use 3D modelling software. I chose to use Onshape[10] , a free online modelling software 1 which I was already familiar with, due to having used it extensively in my DT IGCSE.

I first decided to use images I found online as a reference to create a simple design, so I created a collage of a few existing polar robot arms to base my first design from. (see Figures 1, 2, and 3). [14] [13] [6] I had to model the servo motors, so I started by modelling a generic 9g servo motor.

Figure 1: Pre-existing product - orange arm

Figure 2: Pre-existing product - blue arm

Figure 3: Pre-existing product - white arm

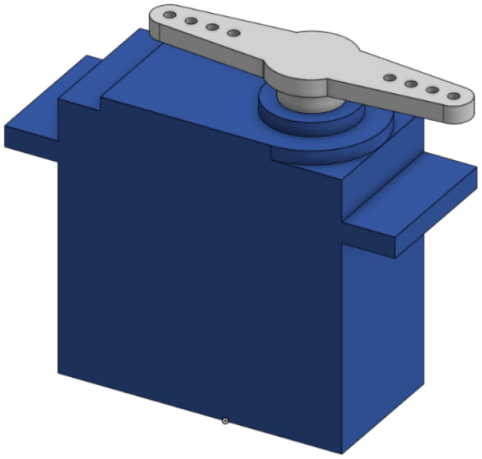


Figure 4: Blue 9g servo model

With this and my specification, I decided to start modelling the arm. I knew this arm wasn't going to be my final design, so I did not worry about making the components easy or realistic to print. Once I modelled a functional design, I knew that my product was feasible, so I decided to refine my concept into a first prototype.

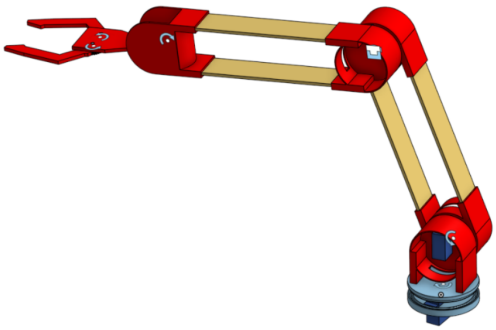


Figure 5: Robotic Arm Concept Model 1

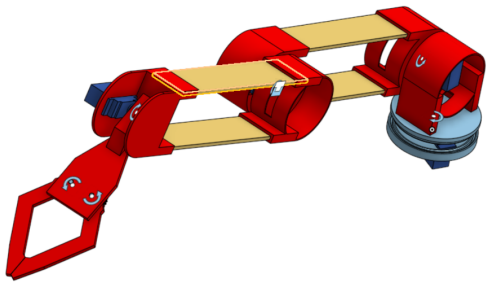


Figure 6: Robotic Arm Concept Model 2

Before designing my first prototype, I had to source some components such as motors. I knew that these 9g servo motors with a torque of 0.1NM would not be strong enough to power my arm, so I had to find some more powerful motors.

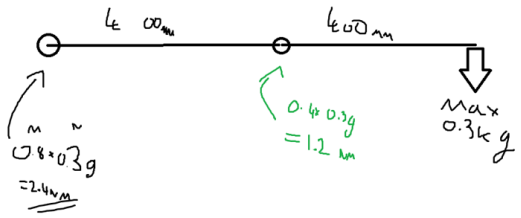


Figure 7: Torque Calculation Diagram

This shows that with a maximum payload of 300g, at 0.8m, the arm will only need 2.4NM of torque to keep it stationary. However this does not account for the mass of the arm itself, which at this stage in the process, I can only guess. Assuming each section of 400mm is 0.5KG, this gives us: Which shows that the shoulder motor needs at least 6.4Nm of torque to keep it still. This also

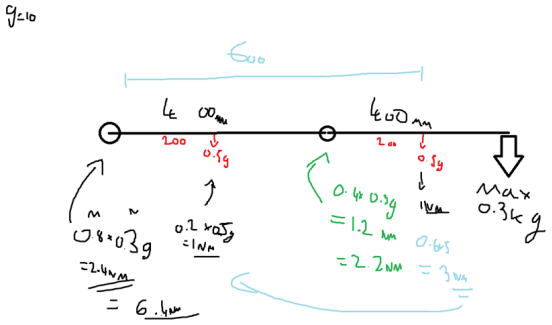


Figure 8: Torque Calculation Diagram

does not include the force needed to move the arm, which would require more torque. I decided to go for this motor [3] for the shoulder since it has 15NM of torque, and has a servo horn that is attached on both sides, meaning attaching it to the arm should be easy. For the elbow motor I decided to go for this 3.5NM torque motor [4], which provides sufficient torque to rotate the elbow.

Figure 9: Shoulder Motor

Figure 10: Elbow motor

For the wrist motor, since the distance is around 100mm, I will use a standard 0.4NM servo motor. This now meant I could go onto designing my first prototype.

2.3.2 First prototype

I first modelled these motors in Onshape

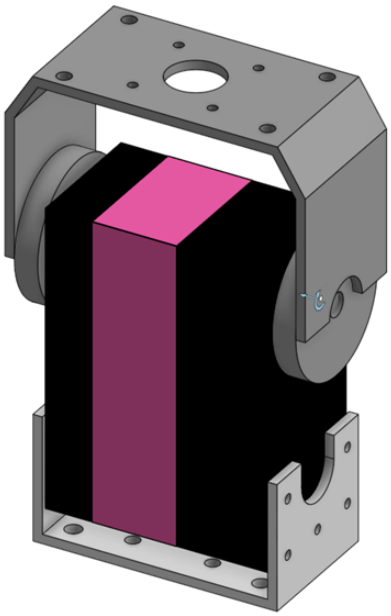


Figure 11: Shoulder Motor Onshape Model

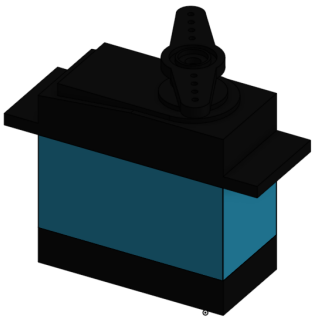


Figure 12: Elbow Motor Onshape Model

Using these servo motor models, I created mounts for the servos to attach to. The elbow had to rotate more securely, so I used large bearings to design the elbow hinge.

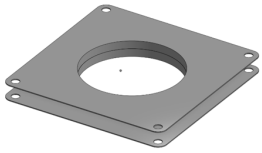


Figure 13: square bearing model

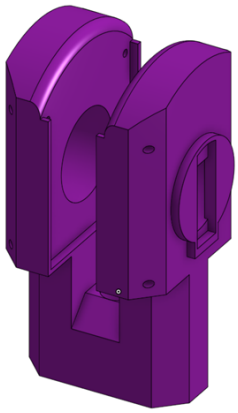


Figure 14: Elbow mount

Once the servo mounts have been designed, all that was needed was the end effector and the upper + lower arm. I decided to do the upper and lower arm first, since they would be relatively simple to build. This part will hold a lot of torque, so it would need to be strong, therefore I added a male – female joint on the parts so that there would be a large area for superglue to adhere to.

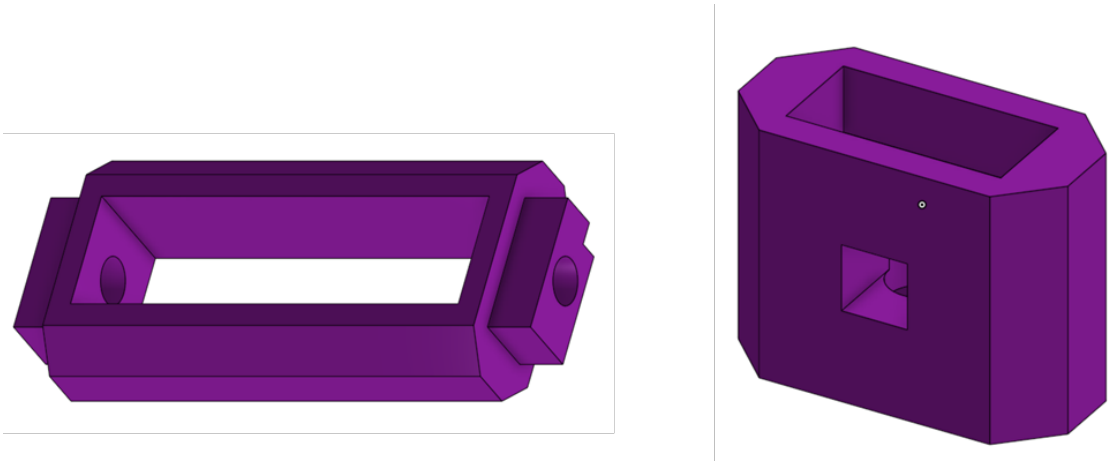


Figure 15: Enter Caption

Finally, I had to model the end effector. Usually, these could be anything, however for my purpose, I decided to go for a simple gripper, since that could be used for multiple purposes, including moving wooden blocks. Researching designs for claw end effectors, there were a lot of complex 4 bar linkage designs. (see Fig: 16[16])

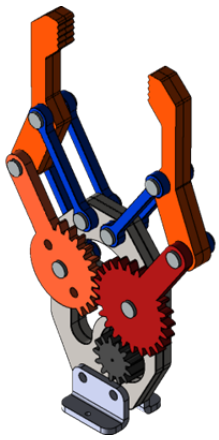


Figure 16: Example Claw

I decided that since this is going to be a first prototype to test the movement of the arm, I was going to go for a much simpler design, and chose to go for a simple 2 gear design. This has its disadvantages, such as the claws not being constantly parallel , however as a proof of concept it works great. I went for the simplest possible design I could think of, which was 2 gears, one of them being attached to the servo motor, and the other being attached to a pivot.

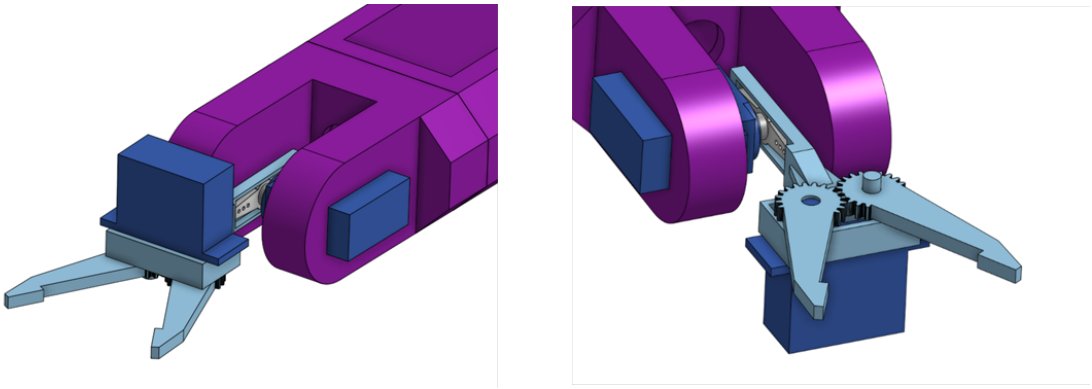


Figure 17: End effector Model

Once I finished each part, I had to spend some time in the assembly tool drawing mate relationships between each object.

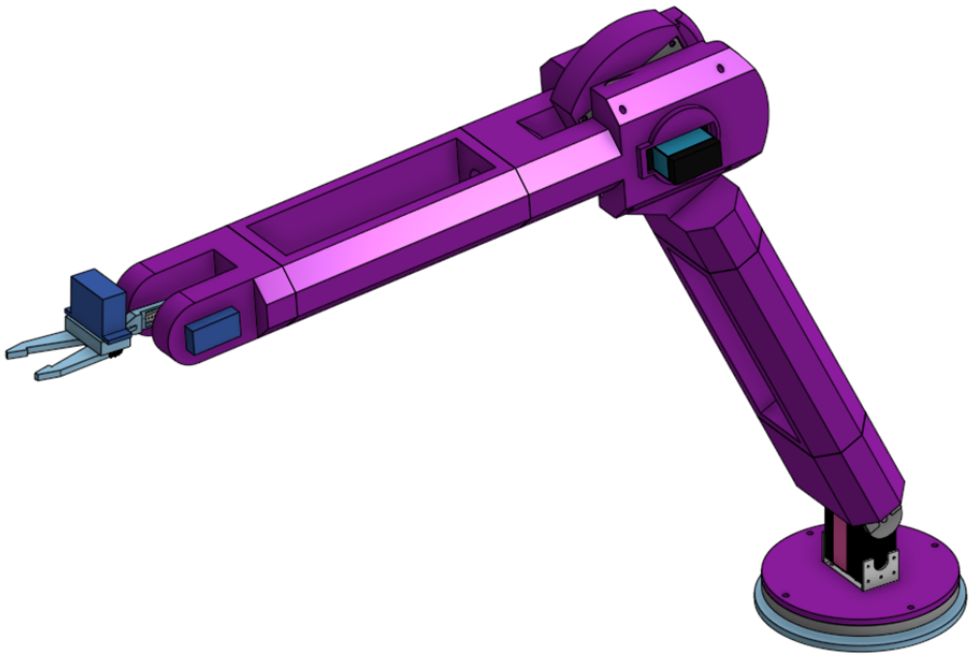


Figure 18: Arm model

This is going to be my first prototype design of the robotic arm that I was going to print. From the mistakes I will encounter during this design I will build my final design at a later date, most likely in a few months.

3 Construction

3.1 Risk Assessment

Before I could begin construction of the arm, I had to create a risk assessment to ensure that when I was building the arm, I was doing it as safely as possible, as there is a possibility for bodily harm if done incorrectly.

Risk	Chance /10	Severity /10	Risk Rating /20	Actions to Prevent Risk
Burning (soldering iron, heat gun, hot electronics)	6	7	13	I will ensure that any hot tools are left to properly cool down before moving, and that when they are hot, extra care will be taken when they are used.
Pinching	7	5	12	Hands will be kept clear of the arm when it is moving, and when I am working near the servo motors, the arm will be powered off.
Crushing	4	7	11	The arm will be secured by a desk clamp, and when moved will be moved with great care taken to avoid it being dropped.
Electric shock (low voltage)	4	6	10	Care will be taken around exposed wires, and all work on wiring will be done with the arm powered off.
Electric shock (high voltage)	5	10	15	When working around electronics that are powered by high voltage, they will only be worked on when unplugged, and I will ensure that all capacitors are properly discharged.
Cuts	4	6	10	Knives will be used on a cutting mat, and all items being cut will be secured to prevent them from sliding around and the knife from injuring me.
Chemical burns	3	7	10	When using superglue, I will ensure the bottle is closed when not in use.
Respiratory issues	5	6	11	All soldering and supergluing will be done in a well-ventilated area with fume extractors running to ensure that the toxic fumes do not get inhaled.

Table 1: Risk assessment

Having completed my risk assessment, I proceeded with constructing the arm.

3.2 Assembly

3.2.1 FDM 3D Printing

The first step of assembly was to make the body of the arm, and since I had designed the arm on Onshape, this made it simple to export each component as a .STL file and then print it. This was one of the reasons I chose to use 3D modeling software to design the arm, since it would make the construction process simpler. Once I exported the .STL files I needed to print them. I used a Bambu Lab 3D printer that my friend owned, the P1S model, which I could access remotely over Bambu Studio to upload and print the arm. This was done partly due to me being able to have almost constant access to the printer, at the cost of some rolls of filament. Before I started printing, I had to consider the strength of my parts, and how long the print would take. FDM printers work by extruding layers of plastic and depositing them onto the previous layer, creating a 3D object [1], and these layer lines are usually the weakest point in a 3D print, due to each layer having to be fused together. One way to fix this is by orienting your part to be normal to the layers [2]. Another problem would be the time taken to print each part, since the parts are of reasonable size, it would take a long time to print them. Since I had limited access to the 3D printer due to time constraints, I had to optimise the print for time, which I ended up regretting due to the fragility of some of the parts. Eventually I settled upon my print orientation and set the prints off. In total, the prints took 15 hours to finish.

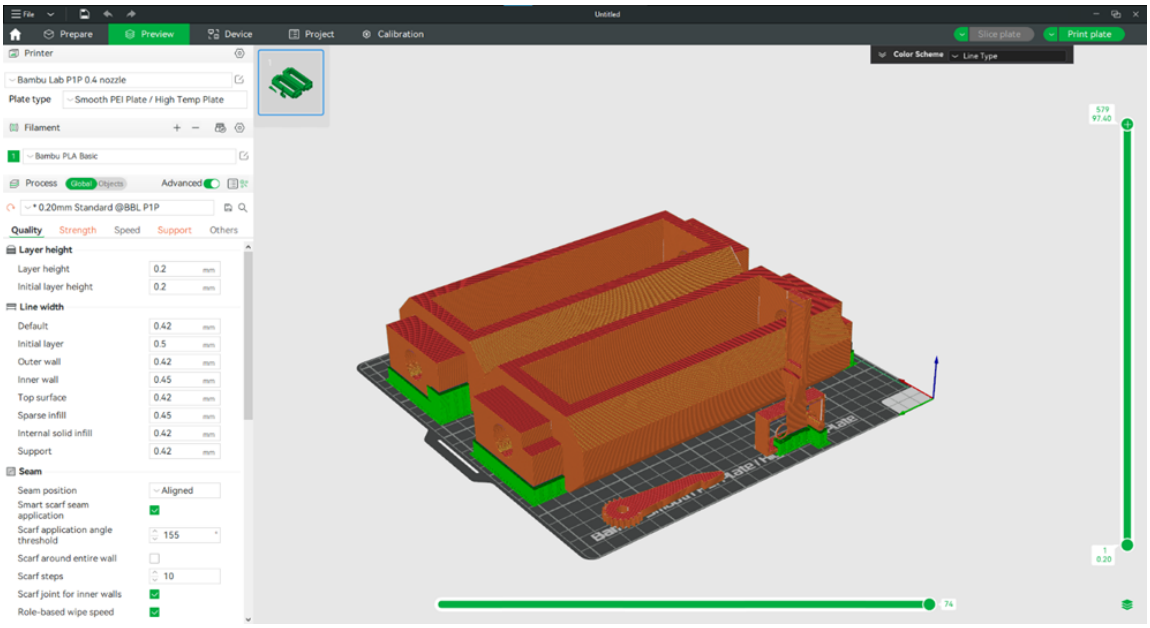


Figure 19: A screenshot of one of my sliced plates, from Bambu Slicer

Once the prints were fully printed, I now had to assemble the prints.

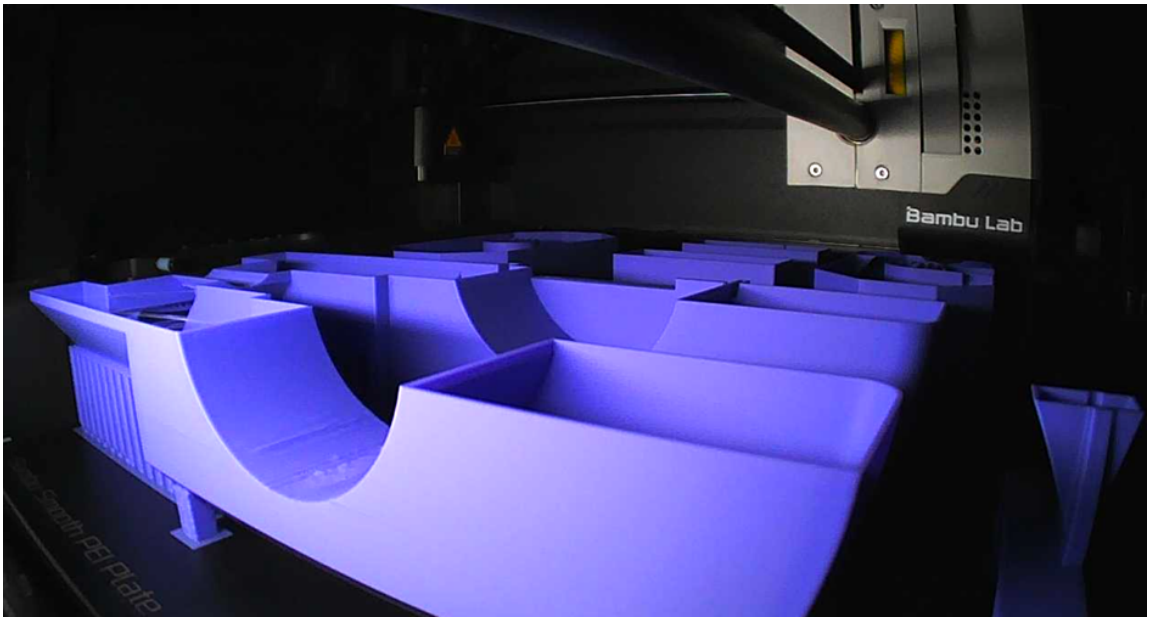


Figure 20: 3D printer printing the arm

3.3 Assembling the print

Once the prints were finished, I needed to clean them up, and this involved removing the support material (seen in green in Fig 19), and sanding rough parts on the print for a better fit. At this stage I encountered my first issues with the parts I designed and printed, namely I forgot to include tolerances for the joints, which meant that the parts did not fit and had to be sanded down even more to fit them. Then when it came to fitting the bearings and the hinge together, I realized that the servo mount was in the way of the hinge assembly, but luckily after some excessive force, the parts were assembled, and the hinge construction was done. Every part was superglued in place to ensure that it was secure. Once the arm had been mostly assembled, I had to mount the motors. This involved having to attach servo headers using superglue to the print, which required careful positioning. I decided to not glue the servo motors in, as that won't be necessary for the arm to function, and as this is the first prototype, I will be reusing the motor and electronics assembly. The final issue with assembly occurred when I was testing to see the arms range of motion and how it moves, and at this point some of the layer lines started splitting, resulting in a drastic weakening of the structure of the arm. Superglue worked to secure these layer splits in place, however these are likely points of failure for the arm, and will have to be reinforced in a temporary manner when testing the code.

Figure 21: Most of the arm structure assembled

3.4 Wiring

The arm requires power to be supplied to each motor and the microcomputer used to control it, and each component runs at a different voltage, so this ruled out a single power supply and bus for everything. This also meant that I was unlikely to find something off the shelf that provides 12v6, 8v6 and 5v0, therefore I had to create one myself. To create a power supply I first had to know the output voltages I needed, and the maximum possible current draw from each source. This was provided by the motor manufacturers, and doing some simple maths:

Using $P = IV$ $12.6v \times 8.3A = 104.6W$ $8.6v \times 2.6A = 24.1W$ $5.0v \times 1.0A = 5.0W$ $5.0v \times 1.0A = 5.0W$ This gives a maximum power draw of 138.66W, we can take this as 140W of power at the stall torque for each motor. The base motor, which has the largest power draw of all my motors, has 17.3NM of torque, and since during the design phase I calculated that we will only be using around 10NM or torque maximum, the motor is highly unlikely to ever reach its stall torque, the motor is unlikely to draw more than 30W of power, since the power drawn increases exponentially as the torque approaches stall torque. Therefore a power supply of 65-70W will be suitable for our purposes. Common power supplies that output this kind of power are old barrel jack laptop chargers. Therefore after finding an old laptop charger, I checked that it worked using a multimeter, and then cut off the end. After reading a tutorial on how somebody used an old laptop charger for a similar purpose [8],I soldered a XT-60 connector onto the laptop charger, and then had to create my power distribution circuit. Inside the cable for the laptop charger, the original barrel jack had an earth wire, and I decided to extend that even though I was unlikely to need it, but in case I had to earth any metallic parts in the future. I decided to add the XT-60 since I already had a few of these connectors, and in case I needed to replace the power supply I could easily unplug it and replace it with a higher power one. (ref 22 [5])

Figure 22: XT-60 connector

I then had to find a way of stepping down the voltage to the required voltages. I needed an adjustable DC-DC step down converter, and after doing some research into DC-DC converters [9]I found the LTM2596 Buck converter, and a tutorial on how to use one[7]. I then used 3 buck converters to step down the 18.6v output from the power supply to the required voltages. I wired these in parallel to each other, and connected them to a XT-60 plug, and after soldering on some wires onto the buck converters. With this done, the power supply was created.

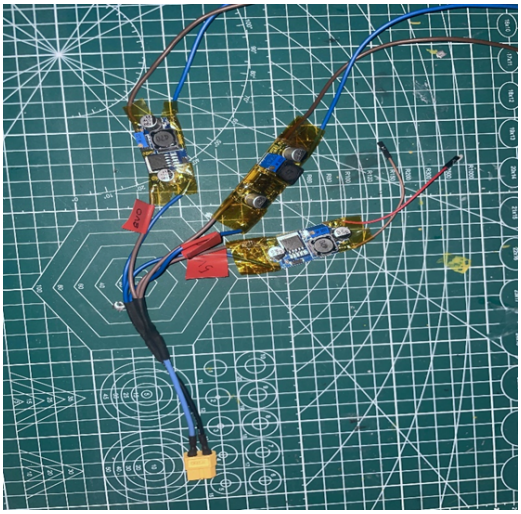


Figure 23: Circuit with 3 buck converters

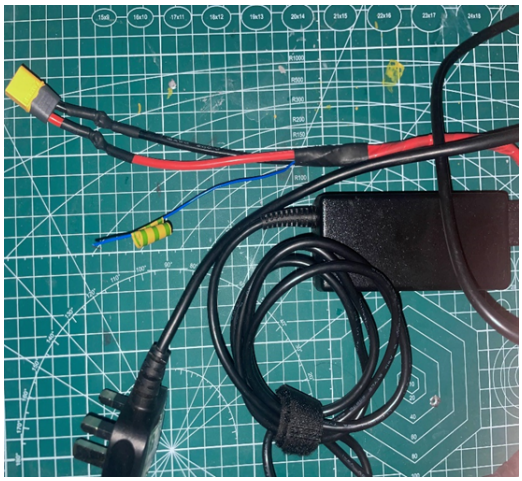


Figure 24: Laptop charger with XT-60 and Earth wire

The final decision that was needed was to choose a controller for the arm, this meant an interface that would take in user input, and translate that to movement of the arm. This could be done in a number of ways, but I chose to use a raspberry pi due to my previous familiarity with it, and its ability to run a desktop environment with a GUI, meaning i would not need to code an interface between my laptop and the arm, and not have to do a web-based controller. (ref 25[15])

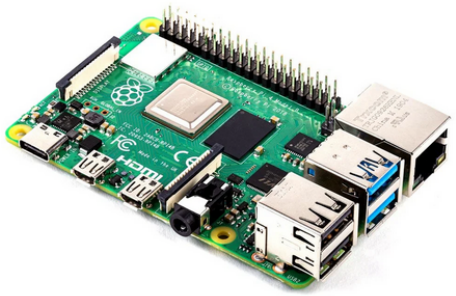


Figure 25: Raspberry Pi 4B

4 Coding

4.1 Polar Co-ordinates

Now that the first prototype of the arm was mostly built, I could now start on writing my code. Before starting to write my code, I needed to decide how my arm would be controlled, this meant I had to think how my arm would move through 3D space, would i use a fully Cartesian co-ordinate (x, y, z) system, a polar coordinate system (r, Φ, θ), or a mix of both. Initially, using a fully Cartesian system seemed like the obvious choice, since that is how humans think, however a polar system makes more sense, since it uses angles and lengths.

4.1.1 What are Polar Coordinates?

Polar coordinates are a coordinate system that use (r, Φ, θ), a length and 2 angles to describe a point in 3D space, rather than 3 lengths each on a separate plane, as with Cartesian. This is much more intuitive for a robotic arm, since it also uses known lengths and angles. (ref 26 [17])

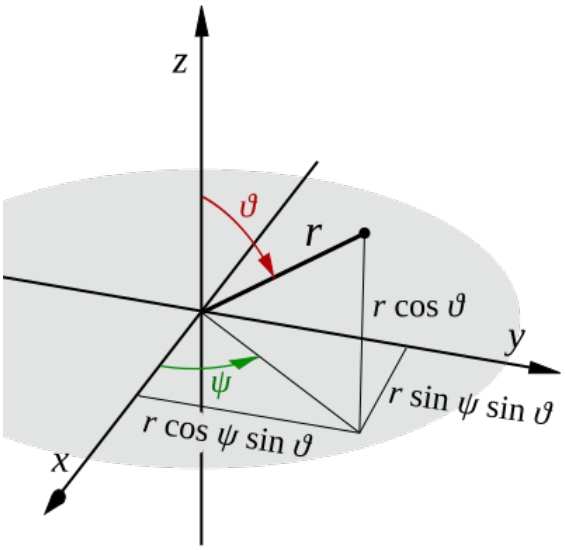


Figure 26: Polar Coordinates in 3D

By converting my x and y coordinates from the input into radius $r = \sqrt{x^2 + y^2}$ and base rotation $\theta = arctan2(x, y)$that makes it easy to convert the arms movement to planar. Then that means the rest of the arm (the shoulder, elbow and wrist) operate in the r-z plane, meaning it is simple 2D inverse kinematic calculations. Due to this, I chose to use a hybrid approach, the input coordinates will be Cartesian, but internally, the system uses polar coordinates to simplify the calculations done by the program. This let me keep the benefits of an intuitive input, and a simple control system, and so that i am less likely to make a mistake when coding.

4.2 The Controller

Once this has been decided, I could start implementing my controller for the arm. I chose to build the controller in Python since I was already familiar with controlling servo motors. I first started by creating a git repository, so that my code would not be lost, and this would allow me to work on it both at home, at school, and easily upload the code to my raspberry Pi that i am using to control the arm. I wanted the controller to both work over a GUI and over a CLI, which means that i needed to have an interface implemented so that i could control the arm by clicking some buttons.

First, I had to decide what library i was going to use for my graphical interface, and how i am going to interface with the servo motors. For the GUI, i am using Tkinter, since it comes with python on raspberry Pi's, and is more than advanced for this application. My interface with the servo motors is going to be over the Raspberry Pi's General Purpose Input and Output pins, which i will refer to as GPIO pins from now on. The raspberry Pi was chosen because of its 26 addressable GPIO pins, meaning i could control up to 26 devices with it, and i am going to use it to control my servo motors. Servo motors work by using Pulse Width Modulation, PWM, and this is a special technique that allows for controlling the average power delivered to a device, essentially achieving an analogue signal using digital pulses. [12]

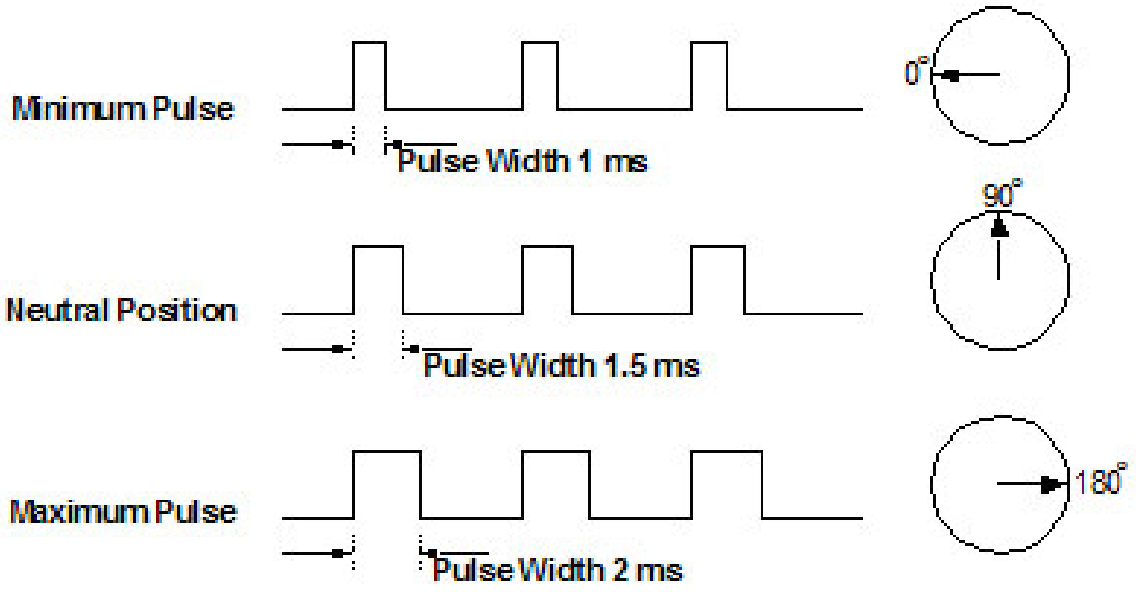


Figure 27: Pwm diagram

The Raspberry Pi has an in-built library called gpiozero that handles PWM output for me, meaning all I needed to do was pass an angle to a function. I wrote a simple servo class that I would use as part of my arm.

```
    class Servo:
    def __init__(self, pin, minAngle, maxAngle):
        self.__pin = pin
        self.__minAngle = minAngle
        self.__maxAngle = maxAngle
        self.__angle = 0
        self.__servo = gpiozero.AngularServo(self.__pin, self.__minAngle,
            self.__maxAngle)

    @property
    def pin(self) -> int:
        return self.__pin

    @property
    def angle(self) -> int:
        return self.__angle

    def __moveToAngle(self, angle):
        self.__servo.angle = angle
        print(f"Servo on pin {self.__pin} moved to {angle} degrees")

    @angle.setter
    def angle(self, value: int):
        """Move to angle value and clamp between minAngle and maxAngle"""
        if value < self.__minAngle:
            self.__angle = self.__minAngle
            self.__moveToAngle(self.__minAngle)
        elif value > self.__maxAngle:
            self.__angle = self.__maxAngle
            self.__moveToAngle(self.__maxAngle)
        else:
            self.__angle = value
            self.__moveToAngle(value)
```

This lays out the base servo class with an angle getter and setter, and the angle setter also clamps the angle between the minAngle and maxAngle values, to ensure it is not going out of bounds.

```

class Arm:
    def __init__(self, base: Servo, shoulder: Servo, elbow: Servo, wrist:
        Servo, gripper: Servo, upperArmLength: int, lowerArmLength: int,
        wristLength: int):
        self.base = base
        self.shoulder = shoulder
        self.elbow = elbow
        self.wrist = wrist
        self.gripper = gripper
        self.upperArmLength = upperArmLength
        self.lowerArmLength = lowerArmLength
        self.wristLength = wristLength

```

```
    def moveJoint(self, joint, angle):
        joint.angle = angle

    def moveArm(self, x, y, z):
        baseAngle = math.degrees(math.atan2(y, x))
        r = math.sqrt(x**2 + y**2)
        d = math.sqrt(r**2 + z**2)
        L1 = self.upperArmLength
        L2 = self.lowerArmLength
        if d > (L1 + L2):
            print("Target out of reach.")
            return

        try:
            elbowAngleRad = math.acos((L1**2 + L2**2 - d**2) / (2 * L1 * L2
                ))
            elbowAngle = math.degrees(elbowAngleRad)

            alpha = math.atan2(z, r)
            beta = math.acos((d**2 + L1**2 - L2**2) / (2 * d * L1))
            shoulderAngle = math.degrees(alpha + beta)


            wristAngle = - (shoulderAngle - elbowAngle)
        except ValueError:
            print(f"Domain error, unreachable?? {(elbowAngle, alpha, beta,
                shoulderAngle, elbowAngle, wristAngle)}")
            return

        self.base.angle = baseAngle
        self.shoulder.angle = shoulderAngle
        self.elbow.angle = elbowAngle
        self.wrist.angle = wristAngle
        print(f"Arm moved to position ({x}, {y}, {z})")
```

This is the code that I implemented to move my arm. The moveArm function takes a Cartesian x, y, z and moves the arm based on a polar system. The only issue is, I never got to test that function since the arm broke before I got to that stage of testing. This function may not work, and I will be interested to see what happens if I finish the arm over the summer.
Now the code needs some way of being told where to move, and this is where a GUI (Graphical User Interface) comes in. I wrote a simple GUI using Tkinter, which looks like this:

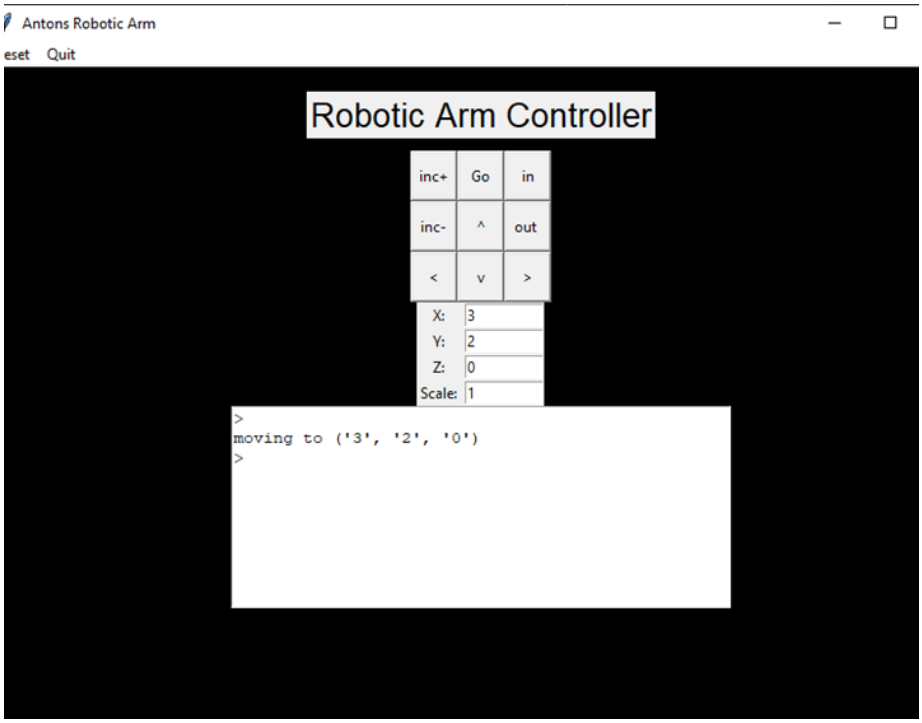


Figure 28: GUI

The GUI works by pressing the buttons, and that increments (or decrements) the XYZ value. the big GO button tells the arm to move to that position by running moveArm(x,y,z) in the backend. The text box at the bottom shows the outputs from the functions ran, and allows for command inputs. This means that I could implement a simple script that can take commands from a text file and queue them, to make the arm run repetitive tasks.I tested this with print statements and it worked, but unfortunately, as became an issue, I could not test most of my code on the arm due to its lack of existing. This scripting looked like this:

```
moveto 0 0 0
wait 2
moveto 10 10 0
wait 2
moveto 0 0 0
```

Each line was ran by the code sequentially, and called the relevant function with the following parameters.
Here is the rest of the GUI code, including all function definitions and GUI management.

```
import tkinter as tk
import servoClass
import os
import time
arm = servoClass.Arm(servoClass.Servo(1, 0, 180), servoClass.Servo(2, 45,
```

```
    135), servoClass.Servo(3, 0, 180), servoClass.Servo(4, 0, 180),
    servoClass.Servo(5, 0, 180), 30, 30, 10)

armScriptsDirectory = "/home/anton/Programming/EPQ/src/armscripts"

def incrementString(string, increment):
    return str(int(string) + increment)

########## button calls ##########
def up():
    YEntry.set(incrementString(YEntry.get(), int(scaleEntry.get())))
def down():
    YEntry.set(incrementString(YEntry.get(), -int(scaleEntry.get())))
def left():
    XEntry.set(incrementString(XEntry.get(), -int(scaleEntry.get())))
def right():
    XEntry.set(incrementString(XEntry.get(), int(scaleEntry.get())))
def inward():
    ZEntry.set(incrementString(ZEntry.get(), int(scaleEntry.get())))
def out():
    ZEntry.set(incrementString(ZEntry.get(), -int(scaleEntry.get())))
def go(): # moveTo FUNCTION
    x, y, z = XEntry.get(), YEntry.get(), ZEntry.get()
    stdout("moving to " + str((x, y, z)), 0)
    #print(console.get(1.0, tk.END))
    arm.moveArm(x, y, z)


def ScaleUp():
    scaleEntry.set(incrementString(scaleEntry.get(), 1))
def ScaleDown():
    scaleEntry.set(incrementString(scaleEntry.get(), -1))

def reset():
    XEntry.set("0")
    YEntry.set("0")
    ZEntry.set("0")
    scaleEntry.set("1")

def moveTo(x, y, z):
    XEntry.set(x)
    YEntry.set(y)
    ZEntry.set(z)
    go()

def loadScript(name):
    stdout(f"loading script {name}", 0)

    with open(f"{armScriptsDirectory}/{name}.arm") as f:
        for line in f:
```

```
54              #stdout(line, 0)
55              parseCommand(*(line.strip("\n").split(" ")))
56
57
58  def listScripts():
59      return os.listdir(armScriptsDirectory)
60
61
62
63
64
65  ######### Console functions #########
66  entryChars = "> "
67  def clear():
68      console.delete(1.0, tk.END)
69      console.insert(tk.END, entryChars)
70
71  def getConsole(*args):
72      #print(args)
73      data = console.get(1.0, tk.END)
74      #print(data)
75      return data
76
77  def parseCommand(*args):
78      print(args)
79      command = args[0]
80      cargs = args[1:]
81      match command:
82          case "x":
83              XEntry.set(cargs[0])
84          case "y":
85              YEntry.set(cargs[0])
86          case "z":
87              ZEntry.set(cargs[0])
88          case "scale":
89              scaleEntry.set(cargs[0])
90          case "up":
91              up()
92          case "down":
93              down()
94          case "left":
95              left()
96          case "right":
97              right()
98          case "in":
99              inward()
100         case "out":
101             out()
102         case "moveto":
103             moveTo(*cargs)
```

```
104             case "go":
105                 go()
106             case "reset":
107                 reset()
108             case "exit":
109                 main.quit()
110             case "wait":
111                 stdout(f"Waiting {cargs[0]} seconds", 0)
112 
113                 time.sleep(int(cargs[0]))
114             case "clear":
115                 clear()
116             case "load":
117                 loadScript(cargs[0])
118             case "list":
119                 print(listScripts())
120             case "help":
121                 print("Commands: up, down, left, right, in, out, moveTo, go,
                        reset, exit, wait, clear, load, list, help")
122             case _:
123                 print(f"Invalid command {command}")
124 
125 
126 
127 
128 def stdout(string, mode, *args):
129     """Mode: 0 = stdout, 1 = stderr"""
130     console.insert(tk.END, "\n")
131     if mode == 0:
132         console.insert(tk.END, string)
133     elif mode == 1:
134         console.insert(tk.END, string)
135     else:
136         raise ValueError("Invalid mode")
137     console.mark_set("insert", tk.END)
138     console.insert(tk.END, "\n"+entryChars)
139     console.see(tk.END)
140 
141 
142 def stdin(*args): # called upon return key (enter)
143     #print(getConsole())
144     entry = "".join(getConsole()).split(entryChars)[-1].strip("\n")
145 
146     print(entry)
147     entry = entry.split(" ")
148 
149     parseCommand(*entry)
150 
151     console.mark_set("insert", tk.END)
152     #console.delete(1.0, tk.END)
```

```
153
154         console.insert(tk.END, "\n"+entryChars)
155         return "break"
156
157     def clear():
158         ...
159
160
161
162     ########## MAIN ##########
163
164     width = 800 # x
165     height = 600 # y
166     mWidth = width//2
167     mHeight = height//2
168
169     main = tk.Tk()
170     main.title("Antons Robotic Arm")
171     main.geometry(f"{width}x{height}")
172
173     main.config(bg="black", padx=10, pady=10, cursor="tcross")
174
175     title = tk.Label(main, text="Robotic Arm Controller", font=("Arial", 20))
176     title.pack(pady=10)
177
178
179     XEntry = tk.StringVar()
180     YEntry = tk.StringVar()
181     ZEntry = tk.StringVar()
182     scaleEntry = tk.StringVar()
183
184     menuBar = tk.Menu(main)
185     menuBar.add_command(label="Reset", command=reset)
186     menuBar.add_command(label="Quit", command=main.quit)
187     main.config(menu=menuBar)
188
189
190
191
192     buttonGrid = tk.Frame(main)
193     tk.Button(buttonGrid, text="inc+", width=4, height=2, command=ScaleUp).grid
            (row=0, column=0)
194     tk.Button(buttonGrid, text="Go", width=4, height=2, command=go).grid(row=0,
             column=1)
195     tk.Button(buttonGrid, text="in", width=4, height=2, command=inward).grid(
            row=0, column=2)
196     tk.Button(buttonGrid, text="inc-", width=4, height=2, command=ScaleDown).
            grid(row=1, column=0)
197     tk.Button(buttonGrid, text="^", width=4, height=2, command=up).grid(row=1,
            column=1)
```

```
198 tk.Button(buttonGrid, text="out ", width=4, height=2, command=out).grid(row
    =1, column=2)
199 tk.Button(buttonGrid, text="<", width=4, height=2, command=left).grid(row
    =2, column=0)
200 tk.Button(buttonGrid, text="v", width=4, height=2, command=down).grid(row
    =2, column=1)
201 tk.Button(buttonGrid, text=">", width=4, height=2, command=right).grid(row
    =2, column=2)
202
203 # + x /
204 # - ^ /
205 # < v >
206 buttonGrid.pack()
207
208 textGrid = tk.Frame(main)
209 XEntry.set("0")
210 YEntry.set("0")
211 ZEntry.set("0")
212 scaleEntry.set("1")
213 tk.Label(textGrid, text="X: ").grid(row=0, column=0)
214 tk.Label(textGrid, text="Y: ").grid(row=1, column=0)
215 tk.Label(textGrid, text="Z: ").grid(row=2, column=0)
216 tk.Label(textGrid, text="Scale: ").grid(row=3, column=0)
217 tk.Entry(textGrid, width=10, textvariable=XEntry).grid(row=0, column=1)
218 tk.Entry(textGrid, width=10, textvariable = YEntry).grid(row=1, column=1)
219 tk.Entry(textGrid, width=10, textvariable=ZEntry).grid(row=2, column=1)
220 tk.Entry(textGrid, width=10, textvariable=scaleEntry).grid(row=3, column=1)
221 textGrid.pack()
222
223 consoleGrid = tk.Frame(main)
224 console = tk.Text(consoleGrid, width=50, height=10)#, textvariable = "
    consoleText")
225 console.pack()
226 console.insert(1.0, entryChars)
227 consoleGrid.pack()
228
229 console.bind("<Return>", stdin)
230 console
231
232
233
234 main.mainloop()
```

5 Testing

I decided to start testing the arm after construction, and I wanted to do these tests by varying the mass lifted by 50g each time, then moving the arm and seeing how much it "drifts" by each time. The drift would come from imprecise control of the motors, due to either poor build quality, fast movement, slack in the servo motor gears, gravity, encoder noise, etc... some of these I cannot

control, but I wanted to see if it was accurate to a centimetre of movement. First, I mounted the arm onto a clamp stand to test its movement, and to see if the code I had written works, just to move the arm forward and backward, and to test each motor. Unfortunately, when i decided to test full extension, as in a shocking turn of events that no-one could have predicted, the arm snapped in half under its own weight along one of the layer lines. I looked at how the arm broke, and realised it was due to it snapping across one of the layer lines. Even though I did position the prints in such a way that they would be stronger, I unfortunately rushed to print it and had to print on minimal infill settings. This caused the print to essentially come out as walls with almost no internal bracing (other than for overhanging parts) . Due to being so early on in testing, i did not take any photographs of the arm on its mount. The testing of the arm would have looked something like this (this is a 3D recreation)

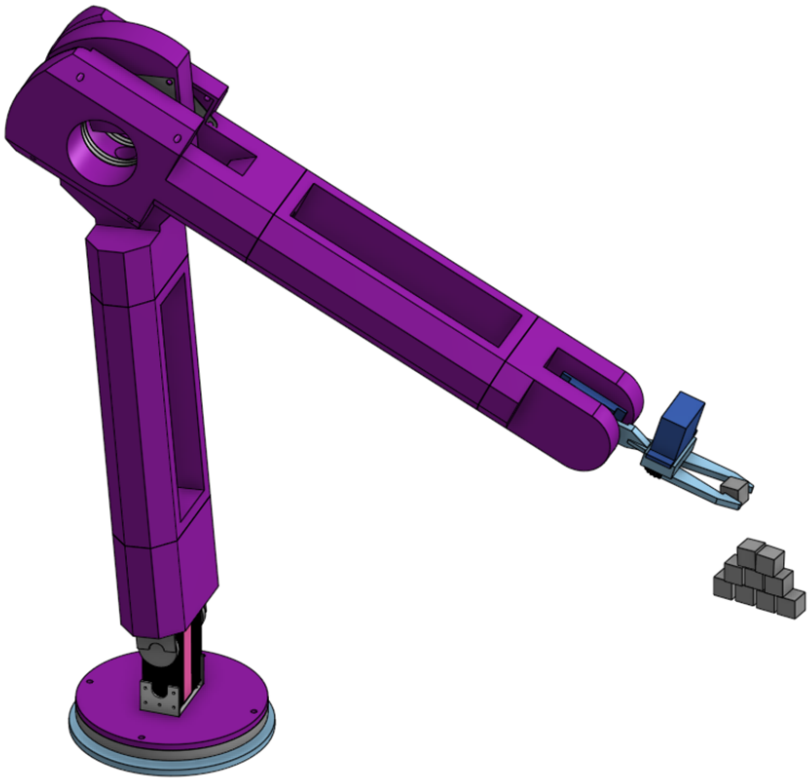


Figure 29: Arm moving blocks

5.1 The Final Design

Now with a snapped arm, I could not continue testing, and I had to make a decision, on whether I should fix this current build, or start designing the final arm. A problem I did encounter after this, and I mentioned this during my EPQ presentation, was around this point I had to delay working on my EPQ due to having the Weizzman safe cracking competition, UKROC nationals, and other competitions going on, which took my focus. This meant that the final version of the arm was not built, or in fact fully designed.

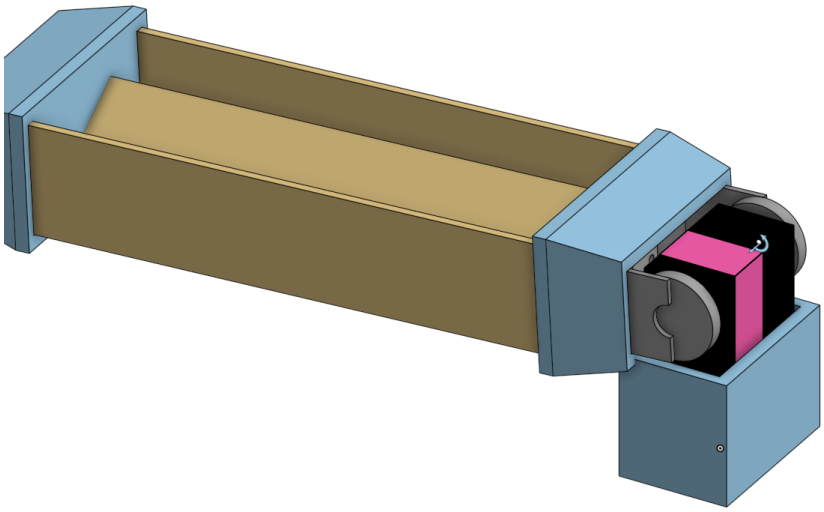


Figure 30: Shoulder to elbow joint of the final arm design

Unfortunately this meant that the project came to a halt and with only 1 week left until the EPQ was to be submitted I had to make the hard choice to not finish the build, and focus on finishing my write-up. The final arm was going to be built with 2 main changes:

- It was going to use minimal 3D printed parts, and try to use more laser-cut components, as this would take significantly less time to build
- It would have a simpler design, so no huge elbow bearing, as that adds unnecessary weight for not much gain

but this did not happen due to me running out of time. I am however planning to continue this project in my free time over the summer holidays.

6 The write-up

When writing the write-up for my project, I first chose to use Microsoft Word due to its simplicity and my familiarity with the software. However, as you may have seen this write-up contains a lot of differently sized images, and the formatting in Word is too restrictive, so after doing half of my write-up I chose to switch to writing it in LaTeX, which allowed for much greater control over all my images.

```
\section{The write-up}
When writing the write-up for my project, I first chose to use Microsoft Word due to its simplicity and my familiarity with the software. However, as you may have seen this write-up contains a lot of differently sized images, and the formatting in Word is too restrictive, so after doing half of my write-up I chose to switch to writing it in LaTeX, which allowed for much greater control over all my images.

\begin{figure}[H]
    \centering
    \includegraphics[width=\linewidth]{images/latex-screenshot.png}
    \caption{A screenshot of this very image ( and paragraph above ), as LaTeX}
    % I found this quite funny at 4am
    \label{fig:latex-screenshot}
\end{figure}
Writing it in LaTeX also let me compile it straight to a PDF by using PDFTeX compiler, and as I used Overleaf, an online LaTeX editor, I could edit my project on all of my devices, even those that couldn't run desktop word.
```

Figure 31: A screenshot of this very image (and paragraph above), as LaTeX

Writing it in LaTeX also let me compile it straight to a PDF by using PDFTeX compiler, and as I used Overleaf, an online LaTeX editor, I could edit my project on all of my devices, even those that couldn't run desktop word.

7 Conclusion

The project was not a success, the arm is not complete, but overall I was happy with my progress. I learned a lot not only on how to build a robotic arm, but mostly on time management and self discipline. I believe that if I had started building the arm earlier, and spent less time procrastinating, I could have had a finished product. Procrastination was the reason this project is not finished, and I am dissapointed that I let myself down due to poor time management. Whilst I don't have a finished product, I have learned a lot from this process, and did enjoy spending time working on it.

7.1 What did I learn?

- Better 3D modelling skills in Onshape
- Build custom power circuits using pre-built components (Bucks converters)

- Referencing my sources properly in large writeups
- How to overcome issues that you may encounter along the process of working on projects

8 Bibliography

[1] all3dp.com. *FDM Explained*. URL: https://all3dp.com/2/fused-deposition-modeling-fdm-3d-printing-simply-explained/.

[2] all3dp.com. *Print Orientation*. URL: https://all3dp.com/2/3d-printing-strength-strongest-infill/#i-2-build-orientation.

[3] amazon.co.uk. *Servo 1*. URL: https://www.amazon.co.uk/ANNIMOS-Brackets12V-Voltage-Digital-Steering/dp/B0C69W2QP7?crid=1GK4G1FM6NKMY&th=1.

[4] amazon.co.uk. *Servo 2*. URL: https://www.amazon.co.uk/Miuzei-Servo-Digital-Waterproof-Making/dp/B0CBBS6KX6?th=1.

[5] cdn.ecommercedns.uk. *XT60 Connector*. URL: https://cdn.ecommercedns.uk/files/8/219978/0/11527590/xt60-connectors-yellow.jpg.

[6] dorna.ai. *White arm*. URL: https://dorna.ai/wp-content/uploads/2023/10/2-1-edited.jpg.

[7] instructables.com. *Buck Converter How-To*. URL: https://www.instructables.com/How-to-Use-DC-to-DC-Buck-Converter-LM2596/.

[8] instructables.com. *Laptop Charger Reuse*. URL: https://www.instructables.com/Re-Purposing-Your-Old-Laptop-Charger/.

[9] monolithicpower.com. *Buck Converter Info*. URL: https://www.monolithicpower.com/en/learning/mpscholar/power-electronics/dc-dc-converters/buck-converters.

[10] Onshape Inc. *Onshape: Product Development Platform*. URL: https://www.onshape.com/en/.

[11] Society of robots. *Robot arm tutorial*. URL: https://www.societyofrobots.com/robot_arm_tutorial.shtml.

[12] servocity.com. *Servo FAQs*. URL: https://www.servocity.com/servo-faqs.

[13] store.clearpathrobotics.com. *UR5 image*. URL: https://store.clearpathrobotics.com/cdn/shop/products/ur5_300x300.png?v=1541621897.

[14] t4.ftcdn.net. *Orange arm*. URL: https://t4.ftcdn.net/jpg/05/48/25/19/360_F_548251959_cTiSpWRma1lza6qqPuorfRRmn6ABfWot.jpg.

[15] thepihut.com. *Raspberry Pi 4*. URL: https://thepihut.com/cdn/shop/products/raspberry-pi-4-model-b-raspberry-pi-14879186059326_1000x.jpg?v=1646247617.

[16] Unknown. *end effector image*. URL: https://www.xixspecials.shop/?path=page/ggitem&ggpid=1982613.

[17] wikipedia.com. *Polar Coordinate diagram*. URL: https://commons.wikimedia.org/wiki/File:Polar-coordinates-3D.svg.

9 Appendix

All the code and 3D files are available on my Github at `https://github.com/Anton-Chernyshov/EPQ/tree/main`
a PDF version of this project is downloadable from `https://github.com/Anton-Chernyshov/EPQ/blob/main/EPQ.pdf`
Thank you to Mrs Davis and all the EPQ supervisors for supporting me throughout my project.