Design and Construction of a Drawing Robot with an Alternative Mechanical Concept

Matura paper

Noah Joshua Werner

10. November 2025

Advisors: Raphael Barengo

Kantonsschule Uetikon am See

Abstract

This work presents the design, construction, and configuration of a drawing robot applying a non-Cartesian coordinate system. The development process includes the specification of a mechanical system of linkages, the formulation of its motion through kinematic equations and a kinematic solver based on these equations. Key decisions concerning mechanical design, electrical engineering, and software implementation are discussed and the robots performance is evaluated with respect to the design goals. It is found that the mechanical configuration chosen is capable of fulfilling the set goals, but that some of the methods implemented caused problems that were not adequately resolved.

Acknowledgements

I would like to express my deepest gratitude towards

- The cantonal school for providing the space and resources to make this project feasible.

- My Advisor Raphael Barengo for his ardent support of this project from the beginning.

- Nicolas Diener for his enthusastic support of the project and help in various ways.

- Guido von Burg for his advice on electrical engineering for this project.

- 2 generations of civil service workers at the cantonal school for managing the 3D printing of parts at the school and enabling me to work on this project during school holidays.

- My brother for his invaluable support and help in creating this document using LaTeX.

- My late father for supporting me in any and all my endeavours with his technical expertise and intuition, giving me the confidence to attempt a project like this.

Contents

Chapter 1

Introduction

This paper aims to explore the design and construction of a drawing robot that operates using a system of mechanical linkages rather than a more conventional system of linear rails driven with a Cartesian coordinate system. Many drawing robots rely on three motors to drive the pen position, where one axis rail is attached to two parallel rails making up the orthogonal axes. While this approach allows for straightforward control and high positional accuracy, it also increases material cost and may obstruct part of the working surface.

In contrast, the concept presented in this work seeks to minimize cost by eliminating the need for large linear rails and reducing the number of motors to two. By employing a linkage-based mechanism, the robots movement is achieved through coordinated angular motion rather than linear translation, offering a challenge in trying to find a way to accurately drive the system. The central objective of this project is to determine whether such a configuration can deliver sufficient accuracy and repeatability for the practical task of drawing images or patterns on paper.

Beyond demonstrating a proof of concept, this work also investigates some of the perks of such a linkage-driven robotic system and different methods to handle them or how they may be leveraged. Through this exploration, the paper also aims to show how creative mechanical design may compensate for less expensive components.

Chapter 2

Planning

2.1 Configurational Considerations

2.1.1 Kinematics

The basis of this work is the special geometry of this robot. Let us describe our ”Kinematic system”, that being the collection of links and their relationships that describe the functional geometry of our robot. This is necessary to describe and calculate the motion of our robot. We will first define the primary component of our system, the link, as two connected points, the first point, its rotational pivot as its ”root” and the second point as its ”tail”. When a link is attached to another link, its root will be placed at the tail of its ”parent”, and it can be referred to as that link's ”child”. We will call a collection of connected links a ”kinematic chain”.

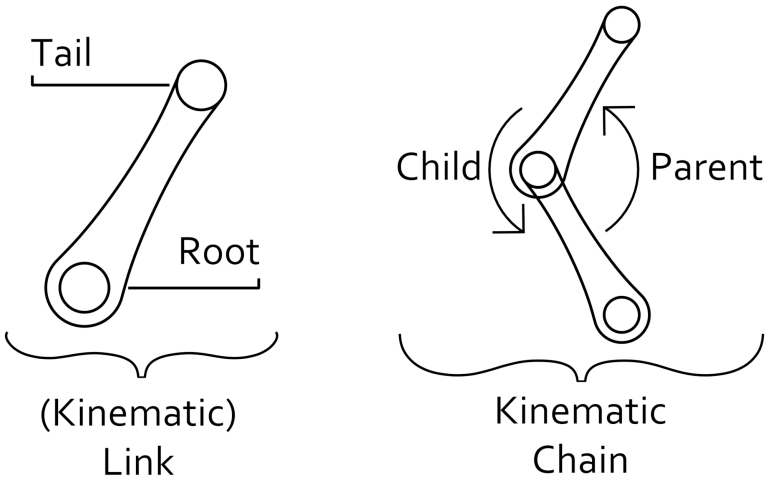

Figure 2.1: (following images unless otherwise specified were created by the author)

This definition is loosely based on the specifications of the ”Bone” component from the 3D graphics tool Blender and will suffice for our purpose.

The System

Our system consists of a total of four links with two kinematic chains. Each chain consists of two links, the first of whom we will call the ”Proximal” link or arm, and the attached link which we will refer to as the ”Distal” link or arm. The distal links of each chain are connected at their tail ends, making two interdependent kinematic chains. As a simplification for now, we assert that the system has mirror symmetry, which allows us to describe the measurements of the system in three simple variables: proximal length, distal length and offset. Offset hereby refers to half the distance between the two proximal root pivots or their distance from the origin of our Cartesian coordinate system. The distance between pivots could also be used, but avoiding the $\frac{1}{2}$ factor in formulas was preferred. This definition will suffice for the next step.

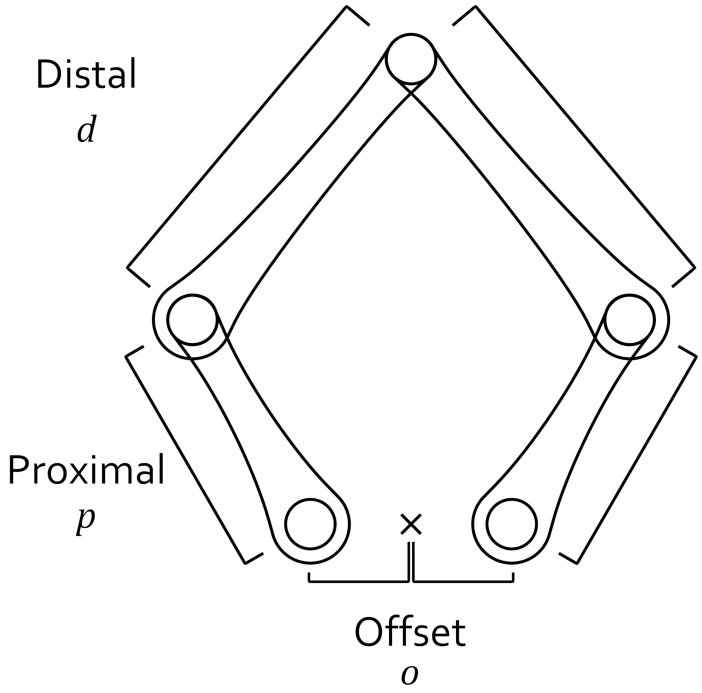

Figure 2.2: Visualization of a possible configuration based on our specification

2.1.2 The Solver and Equations

Having defined our kinematic system, we want to know how to drive it to a target position. What we need is an inverse kinematics (IK) solver. An IK

solver finds a solution for a kinematic chain, so that the end of the chain (or a specified end effector) reaches the target position, often fulfilling certain conditions and constraints. There are general purpose IK solvers, an example being FABRIK (see Andreas Aristidouad 2011), which works iteratively and is classifiable as a heuristic, although such a solver is not necessary in this case. This was only realized later during the project after more time was spent looking at solvers and this specific geometry.

Due to the fact that our kinematic chains are only two links long, a triangle can be formed by drawing a line from the proximal link's root to the distal link's tail. Given all the sidelengths of the triangle, we can calculate any angle within it using the law of cosines giving us the following formula:

$$\gamma = \text{acos}(\frac{a^2 + b^2 - c^2}{2ab})$$

As we will be mechanically driving the angles of the proximal arms, we are interested in the global angle of the proximal arms, or the angle between the proximal arm and one of the world axes. In our case calculating the angle of the arms to the world y axis returns more intuitive values, as an angle of 0° will have the arms pointing straight up, which seems a reasonable rest position for both arms.

To calculate this angle, we first calculate the angle between the vector drawn from the proximal root to our target point $\vec{T}$, and the x-axis using the atan2 function as such: $\text{atan2}(T_y, T_x \pm o)$ ($-o$ for the right chain and $+o$ for the left) and then either adding or subtracting the angle between the proximal arm and t given by our expression from before: $\text{acos}(\frac{p^2+t^2-d^2}{2pt})$. Combining these and subtracting 90° or $\frac{\pi}{2}$ to shift the reference from the x to the y-axis, we receive the following expression for this angle:

$$\measuredangle p = \text{atan2}(T_y, T_x) \pm \text{acos}(\frac{p^2 + t^2 - d^2}{2pt}) - \frac{\pi}{2}$$

We are also interested in the angle between the proximal and distal arms, as their angle will be physically limited later. We refer to it as $\measuredangle d$.

$$\measuredangle d = \text{acos}(\frac{d^2 + p^2 - t^2}{2dp})$$

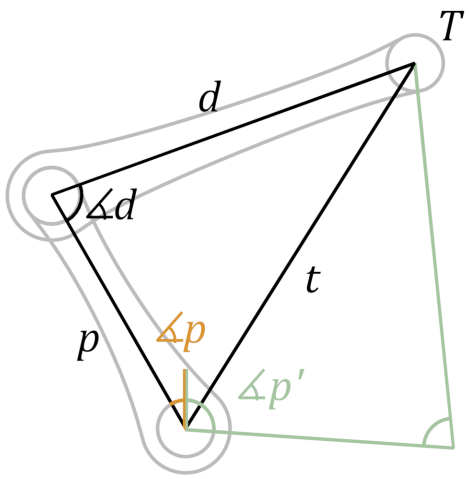

Figure 2.3: Visualization of the constructed triangle and alternative solution.

Looking at the expression, it also becomes apparent that there are 2 solutions for our chain for a given target, which are mirror images across the line t. We choose the solution where the the arm rotates away from the root of the other arm (starting from its rest position) to avoid collisions. The sign of the acos term for the left chain is positive and negative for the right chain in this configuration.

One last important step regarding the calculation of $\measuredangle p$ is to remap our result to a range of -180° to 180° to avoid unexpected issues. This can be done using the following function using floored modulo, as it works in Python:

$$f(\theta) = \mod(\theta + \pi, 2\pi) - \pi$$

This completes the method for converting a target position into angles for the proximal arms.

2.1.3 Measurements/Choosing Parameters

We now want to determine values for d, p and o to maximize the area within which the robot can draw, ideally while keeping the parts small enough for single-part printing. A parametric model was set up leveraging Blender's parameter driver system and using its built-in IK solver at first. This allowed for getting a first feel of how the system moves and behaves with different values for our parameters, but was insufficient to visualize the area the robot could draw on with angle limits set on the joints. To visualize the area the robot could draw, a shader was created using Blender's shader node system. Conventionally, shaders are used to calculate the pixel colors of surfaces in computer graphics, taking in many input variables to create a final color.

This is ideal in this case, as it allows us to evaluate the rotations for each of the joints for all points in the plane simultaneously. This could also be done using Python or any other capable tool, generating a single image. However, implementing it with shaders enables us to recalculate it at runtime in Blender with respect to our parameter values and permits zooming in as far as we like. The only input necessary for this shader is the world position of each pixel. If we use our algorithm to convert from Cartesian coordinates to our proximal angles and set thresholds on the angles of the joints (and do some color mixing for aesthetics) we get a neat visualization for our plotting area. (See Fig. A.1 in Appendix for nodes)

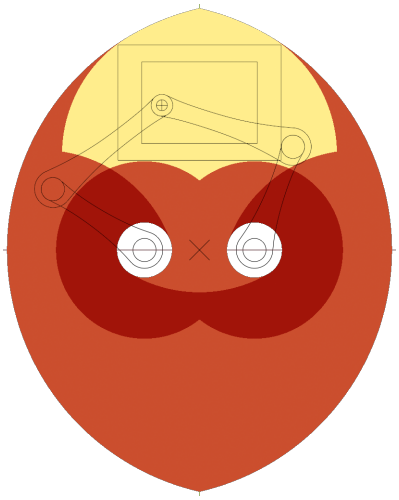

Figure 2.4: Render of the calculated plotting area overlayed with the parametric model of the chains from Blender.

The parameter values of the above plot are: $p = 200mm, d = 250mm, o = 100mm$ with $-35° < \pm\measuredangle p < 135°$ and $\measuredangle d < 35°$ (note: $\pm$ follows the sign of the acos term for the corresponding chain) They were chosen to be able to just about plot across an entire sheet of A4 paper and served as a baseline for the project.

It is worth mentioning that it was noticed the plotting area could be increased significantly, by using a value of 0 for o, making the proximal arms overlap at their roots. With such a configuration the system could also be driven more simply using polar coordinates. Its angle or direction is given by the mean of the angles of the proximal arms while the radius is dependent on the difference of their angle. It was chosen not to do this on the grounds that design and assembly could be complicated beyond the desired degree for the project, as the arms would have to be stacked vertically to clear each other.

2.2 Physical and Practical Considerations

2.2.1 Choosing a Motor Type and Drive Mechanism

To drive the rotation of the proximal arms, a motor had to be chosen. Along with the choice of motor, the manner in which it drives the arm must also be considered. With the primary goal of accuracy in mind, it was decided to transmit rotation to the arm with a reduction between the motor and arm. This reduction would trade speed for increased torque and precision. This also allows for the choosing of smaller and cheaper motors to achieve the torque necessary to move the arms. Gears were considered to achieve this, however, with regard to the part tolerances to avoid the then still remaining possibility of play when changing directions (also known as backlash), they were not seen as a favorable solution. A tensioned V-belt was considered as the first alternative, before stumbling onto a video by Purdue University student Aaed Musa, showcasing a drive system using tensioned rope wound around a spool known as a capstan drive (see Musa 2024b). This appeared optimal in terms of complexity while promising little to no backlash if tensioned properly (Mazumdar et al. 2017; Musa 2024a).

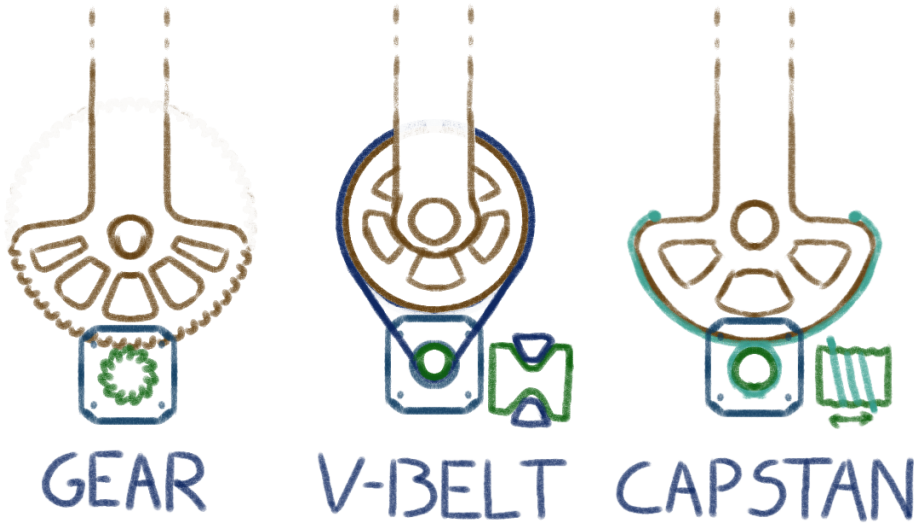

Figure 2.5: Sketch displaying how the named drive types could be implemented

Returning to the choice of the motors, stepper motors seemed to be a good choice for the application after some short research. They are commonly used in 3D printers for the same qualities that make them useful in this case. Stepper motors move in discrete steps, giving excellent control of the angle at which the motor has rotated. The caveat being that they typically operate without any feedback, necessitating calibration through a homing

sequence or by assuming a position on startup. It follows that the known and actual motor position can become de-synchronized if the motor skips a step, typically if it cannot produce the torque necessary to make a step. They can be found in various sizes with various torque ratings, and many stepper driver modules feature micro-stepping, where smoother motion of the motor can be achieved with current modulation. This is especially impactful at slower speeds, where the discrete steps become apparent and unexpected oscillations can occur (*Effects of Microstepping in Stepper Motors* 2014). A set of generic NEMA 17 form factor stepper motors were procured for the project. The torque was not measured, but assumed to be sufficient with the capstan overdrive.

2.2.2 Pen Lifter

To fulfill its purpose of drawing things onto paper, a pen must be fixed to our robot in some way. Furthermore, we want to lift the pen off the paper and set it back down after the robot has moved to another position, allowing it to trace multiple disconnected paths. An additional requirement set was flexibility with the types of pens that could be attached, primarily to allow for trials of different pens to see what would work best, but also to possibly allow for the switching of colors. Pen force or pressure should also be roughly adjustable, as it impacts the line quality depending on the type of pen. Reflected planning of this part was postponed while other parts of the project were worked on. At a later stage, where planning for this part was picked up again, satisfying the above requirements proved challenging. The initial plan was to fix the pen at the joint connecting both of the arms, as that would keep symmetry and require no alteration to our calculations. The pen itself was to be attached to a vertically floating sleeve, held from the side by a two-pronged lever ”fork” that would be raised and lowered by a solenoid piston.

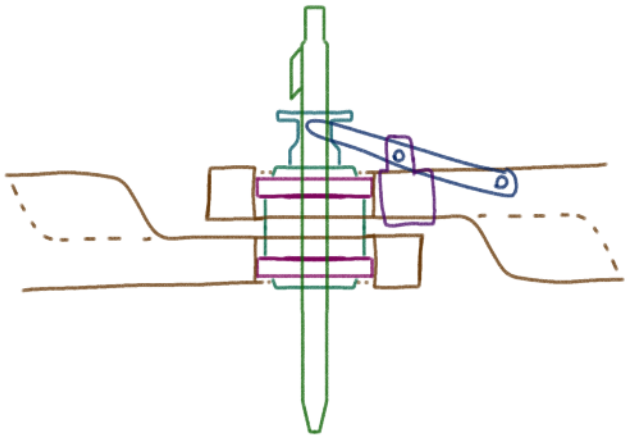

Figure 2.6: Initial concept for the pen lifter mechanism

This was determined to be too complex while also presenting possible problems. It seemed likely that this configuration would struggle with radial loads coming from friction forces of the pen tip on the paper. This would possibly allow the pen to slightly tilt sideways. This in combination with the fact that the pen's sleeve as seen from above would also not be stopped from rotating, and would likely make the resulting behavior hard to predict and account for.

The choice of solenoids to move the mechanism was also reflected upon. Solenoids heat up when powered, significantly so if held for a long enough time. This could cause issues with the surrounding 3D-printed parts. Three ways were considered to deal with this. The first would be to use a type of latch mechanism to keep the pen raised once it has been lifted. A draft for a system using pins running through grooves was made (see Fig. A.2), but this was deemed difficult to implement in a reliable manner while keeping a compact form factor. The second idea was to employ a known technique of limiting the current running through the solenoid after it has reached its extended position, decreasing the heat output while still effectively holding that position. This could be implemented through a purpose built solenoid driver such as the DRV110 by Texas Instruments or by building our own circuit using MOSFETs and capacitors, implementing current limiting through pulse width modulation. The last was to limit the amount of time the powered position was held. If the powered position was to raise the pen off the paper, the amount of time it would be raised while actually drawing would be limited, allowing the solenoid to remain below a critical temperature threshold.

Due to the desire to be able to keep the pen lifted off the paper without any of

these measures, it was decided to use a linear actuator instead of a solenoid piston for this purpose. A linear actuator is another type of electrically powered piston, typically using a DC motor to twist a screw to push and pull on its piston end. Linear actuators were found to be easier to implement, as they only draw current while pushing or pulling and resist movement of their piston even when not powered due to their internal screw mechanism, whereas solenoids do not resist movement at all while unpowered and often feature a return spring. Due to the larger size of the linear actuator compared to the solenoid, it was proposed not to have it aligned with the pen to drive it directly, but to use it to push and pull on a lever to which the pen fixture would be attached. After some sketching and trying, one could realize that this could be done while keeping the linear actuator fixed to the arm by introducing a link between the lever and piston.

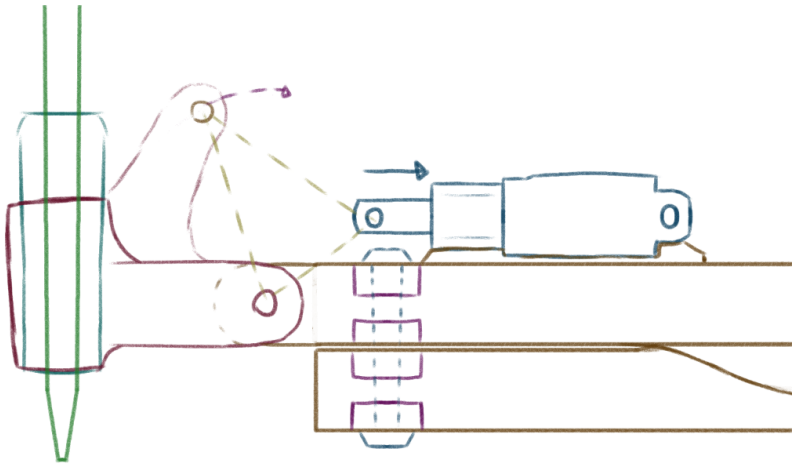

Figure 2.7: Revised concept for the pen lifter mechanism

Coincidentally, this happened to be very similar to the kinematic system describing the robot's arms, making it easy to set up another parametric model in Blender to find a good configuration for the lifter mechanism. The hinge positions for the lever itself and the link, the length of the link and the position of the linear actuator were all chosen to try to minimize horizontal travel of the pen tip, while keeping the final lifted angle reasonably small and retaining a practical form factor.

As can be inferred from the sketch above, this new mechanism will extend one of the distal arms. The pivot connecting the two arms remains unchanged.

2.2.3 Kinematics Revision

Due to the change of our kinematic system caused by the pen lifting mechanism, we must adjust our method for finding the angles for a given target position. Our method for finding the angle $\measuredangle p$ is still valid for a given target and can be used for the extended distal arm by adjusting its now unique value for d. To differentiate these parameters of the two chains, their variables will now be marked with a subscript a or b for left and right respectively: $d \rightarrow d_a, d_b$. Since the right chain now targets a point attached to the left distal arm, we must calculate its coordinates to use as a secondary target $\vec{T_2}$ dependent on our primary target $\vec{T}$. To do this we calculate the angle of the left distal arm relative to the world x-axis for the target $\vec{T}$, following the process described in Section 2.1.2 to get the expression:

$$\theta = \text{atan2}(T_y, T_x + o) - \text{acos}(\frac{d_a^2 + t^2 - p_a^2}{2d_a t})$$

To now obtain the secondary target position $\vec{T_2}$ we construct a vector using this angle θ and the difference of d_a and d_b and subtract it from $\vec{T}$

$$\vec{T_2} = \vec{T} - (d_a - d_b) \cdot \begin{pmatrix} \cos(\theta) \\ \sin(\theta) \end{pmatrix}$$

This target can now be used to calculate $\measuredangle p_b$.

Our updated expressions for $\measuredangle p_a, \measuredangle p_b, \measuredangle d_a$ and $\measuredangle d_b$ are as follows:

$$\measuredangle p_a = \text{atan2}(T_y, T_x) + \text{acos}(\frac{p_a^2 + t^2 - d_a^2}{2p_a t}) - \frac{\pi}{2}$$

$$\measuredangle p_b = \text{atan2}(T_{2_y}, T_{2_x}) - \text{acos}(\frac{p_b^2 + (t_2)^2 - d_b^2}{2p_b t_2}) - \frac{\pi}{2}$$

$$\measuredangle d_a = \text{acos}(\frac{d_a^2 + p_a^2 - t^2}{2d_a p_a})$$

$$\measuredangle d_b = \text{acos}(\frac{d_b^2 + p_b^2 - (t_2)^2}{2d_b p_b})$$

Updating our shader and model in Blender we get a new plotting area.

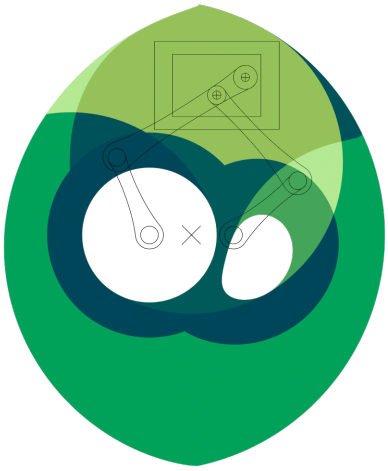

Figure 2.8: Render of the updated plotting area overlayed with the parametric model from Blender.

2.2.4 Baseplate

To mount the parts and provide a consistent surface to draw on a baseplate was chosen. The size of the baseplate was chosen based on the plot area and the reach of the arms, so that the arms would not exit the horizontal bounds of the plate while drawing. Its measurements were set to 700mm x 700mm.

2.3 Electronics

Microprocessor

To control and drive the motors for this project, some kind of microcontroller is necessary. An Arduino UNO was initially proposed as it was readily available and already used for basic testing of components, but it was later replaced in favour of a Raspberry Pi Pico. The Pi Pico microcontroller board using the RP2040, Dual-core processor and native MicroPython support and its own file system (*Pico-series Microcontrollers* 2025). Python seemed ideal to implement more complicated programs whereas Arduino uses its own programming language, which is similar to C++ but with certain oddities like being unable to evaluate certain functions inside other functions in a single line, complicating development somewhat. The most important feature is the file system however, as it allows the easy upload of drawing programs onto the controller, without needing to embed anything into the main file itself. The Pi Pico features 26 General-Purpose Input/Output (GPIO) pins(*Raspberry Pi Pico Datasheet: An RP2040 base microcontroller board* 2024), which will be used to drive components and receive signals.

Stepper Drivers

While building stepper motor driver circuits using more basic components is possible, it is quite involved, which is why dedicated stepper driver boards are a common solution for driving stepper motors (GreatScott! 2016b). Two popular models were found and ordered: the more common A4988 Stepper driver by Allegro Microsystems and Texas Instruments' DRV8825. The modules are nearly identical in terms of pinout and features, both featuring up to 16 microsteps and both rated to handle current up to 2 A with the option to set a current limit (*MOS Microstepping Driver with Translator and Overcurrent Protection* 2022; *DRV8825 Stepper Motor Controller IC* 2014).

These stepper drivers feature a small aluminum heatsink to provide cooling during operation, as they do get warm. Due to this, it was thought to be a good idea to add an active cooler to our circuit. A small 12 V, 90 mA brushless fan was appropriated from a 2008 ATI Graphics card for this purpose.

DC Motor Control

To drive the linear actuator for our pen lifter mechanism, we must be able to switch 12 V across the motor leads. This cannot be done directly by our microcontroller, instead requiring some kind of switch-like component, such as a relay, Bipolar Junction Transistor or MOSFET. Additionally, to change directions, we must be able to switch the polarity across the leads. The circuit necessary for this is called an H-Bridge (GreatScott! 2016a). This circuit could be constructed manually using the switch components named earlier, which was attempted but given up on as it was easier and significantly more compact to use a dedicated board for this as well. Texas Instruments' DRV8871 is such a dedicated DC motor driver IC featuring an H-Bridge and the option for PWM control (*DRV8871 3.6-A Brushed DC Motor Driver With Internal Current Sense (PWM Control)* 2016).

Circuit and Power

The motors for this project were all chosen to be operable at 12 V according to their manufacturers. A 12 V, 4 A Power supply was on hand and chosen to support the approximate load of 2.8 A calculated using the stepper motors' current rating of 1.4 A each (*17HS4023 Datasheet 2 Phase Stepper Motor* 2021), leaving more than enough capacity for the linear actuator and cooler.

The microcontroller requires a voltage input of less than 5.5 V and can be powered by a micro-USB cable by connecting it to a computer. The microcontroller steps this voltage down to 3.3 V to use as its system voltage and also outputs up to 300 mA of current over its 3V3_OUT pin (*Raspberry Pi Pico Datasheet: An RP2040 base microcontroller board* 2024). This 3.3 V output will be used to power logic circuitry of the stepper drivers.

To connect the components chosen into a functioning circuit, a full-size solderless breadboard was acquired as it was big enough to fit the microcontroller, two stepper drivers and the DC driver and allow for fast testing and reconfiguration.

Controls

To allow for a minimal amount of control over the robot outside of software, two buttons would also be implemented. One would be an emergency shutoff button to be used if unexpected behaviour occurs, and an action button to allow the operator to start a task at their own discretion after choosing a program.

Chapter 3

Fabrication and Assembly

3.1 3D Part Design

3.1.1 Modeling Process

Models were made in Blender partly using a tool called CAD-Sketcher, allowing the creation of parametric 2D sketches. This was critical when adjustments had to be made. These sketches would be brought into the 3rd dimension and combined using Blender's modifier system, allowing for recalculation of the entire shape upon change. Not all parts were made in a fully non-destructive way, certain adjustments were done manually after this calculation, either to save time, or because there was no known non-destructive way to achieve the adjustment. The parts were printed at the cantonal school out of PLA filament using its Bambu Lab printers.

3.1.2 The Proximal Arms

Bearings and Axle

It was expected for the proximal arms to bear significant radial load from the distal arms and pen holder assembly that would be attached, therefore it was decided to use ball bearings with a relatively large diameter to prevent tilting of the arms and enable smooth rotation. The use of tapered or conical roller bearings was considered to ensure the axial load from the weight of the attached parts could be carried, but upon checking availability of these parts at reasonable prices, this idea was discarded. Conventional ball bearings were considered sufficient as they can still bear axial loads due to the internal groove in the bearing races (SKF 2025). To fit the bearings in the part, an intermediate 3D printed ring was used. This avoided relying on tight tolerances on a large printed part. To resist the torque transmitted through the bearings, the axle holding the arms in place had to be rigid enough. This was accomplished by adding a large flange to the axle that would force the

axle to align itself with the baseplate when tightened down upon it. To be able to tighten the axle to the baseplate, draw-in nuts were embedded in the plate, as they would sit nearly flush with the bottom of the baseplate and remain attached to the plate when the bolts are removed for any reason.

Capstan features

The radius of rolling surface for the capstan drive was chosen as 8 cm to allow for flexibility in the radius of the spool while keeping the part small enough to print as a single part. Cutouts and holes were made to allow for the insertion of nails to tie the string to. The angular span of the rolling surface was set at 140° based on the span from our model with the angle limits of -30° and 110°. Angle markers every 5° were added to allow for easy calibration.

Physical features

Angular cutouts were made in the model to increase rigidity and visual appeal. Doing so increases the amount of material used at low infill percentages. The pivot for the Distal arm was designed to support the distal arm from the top and bottom to help counteract the torque on the distal arm. This was done assuming that the friction between plastic parts would be small enough to ignore. Intermediate printed fitment rings were used to set bearings here as well.

3.1.3 Capstan drive

A thin but robust synthetic braided string was chosen for the capstan drive. As creep - the stretching of string or rope under load over time - is an important factor with capstan drives (Mazumdar et al. 2017; Musa 2024a) and the creep rate of the string chosen was not known, it was decided to tension the string with small, relatively strong springs. The spool winding the string was designed with a spiral groove to guide the string and increase friction between string and spool to ensure the string would not slip under load. A desired ratio of 8:1 was chosen for the capstan overdrive, meaning the spool needed to have a radius of $\frac{1}{8}$ the radius of the rolling surface on the proximal arm. To truly achieve the desired ratio, both the rolling surface and spool should either have grooves containing half the string, or the radius of both would need to be reduced by $\frac{1}{2}$ the diameter of the string. This was not done and the ratio was determined empirically over the full range of rotation of the arm. The spools were press-fitted onto the motor D-shafts with tight tolerances.

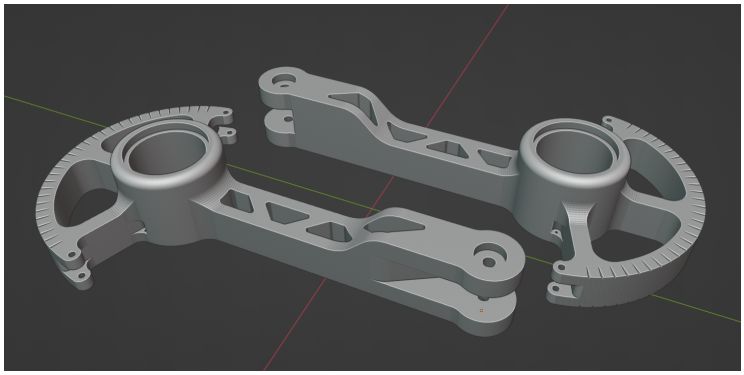

Figure 3.1: Screenshot of the proximal arm models from Blender

Counterweights

To slightly reduce the torque on the axles of the proximal arms due to the weight load of the attached arms in the front, small containers were printed to fit into the openings at the back of the proximal, placed there and filled with metal nuts and washers.

Motor Mounts

To mount the motors to the baseplate, a tight-fit case was made to fix the motors without having to utilize their screw mounts. This case was modeled with two slots for screws that were screwed down to the baseplate with draw-in nuts. The slots allow for adjustment of the distance of the capstan spool to the rolling surface on the proximal arm if the spool diameter was to be changed. This was also practical to allow the winding of the string onto the spool, as this has to be done under some tension.

The motors were mounted at an angle of 45° to the center of their respective proximal pivots. This distributes the 140° range of motion to allow the arms to rotate inwards by 25° and outward by 115° deviating from the limits used in our model.

3.2 The Distal Arms

The two distal arms needed to be vertically offset to clear each other. Since the hinges at the proximal arms are at the same height, this meant the arms would have two planes offset by some distance, creating an overhang which would be inconvenient to print. Turning the part 90° for printing was considered, but the angular cutouts that were already used on the proximal arms were to be adopted for consistency. Instead, since one of the distal arms would be too long to print in a single piece, it was chosen to make a two-part assembly for both of the distal arms. This allowed the printing

of this part without overhang support material. The two parts were to be connected over a sufficiently large mating surface to ensure rigidity and fastened together with nuts and bolts. Since the plane below or above the middle plane, depending on the arm, would also collide with the proximal arms, the mating surface was sheared to clear the arm at the minimal angle of 30° making it a parallelogram. The nature of this design allowed the first part of the distal arms to be identical, where it merely needed to be flipped 180° for one of the arms. While the shearing of the mating surface did help the distal arms clear their proximal arms, it actually reduces clearance between the distal arms. This was realized after the part had already been modeled to a near-final stage. To counteract this while retaining a sufficiently large mating surface, the distal arms were both extended by 30 mm. To avoid tilting of the axle connecting the distal arms, two bearings were pressed into each arm also using intermediary rings. The intention being to provide at least two points of contact per distal arm, so that tilting would be minimized. To further reduce mechanical play, the joint was fastened with spring washers, maintaining slight compression.

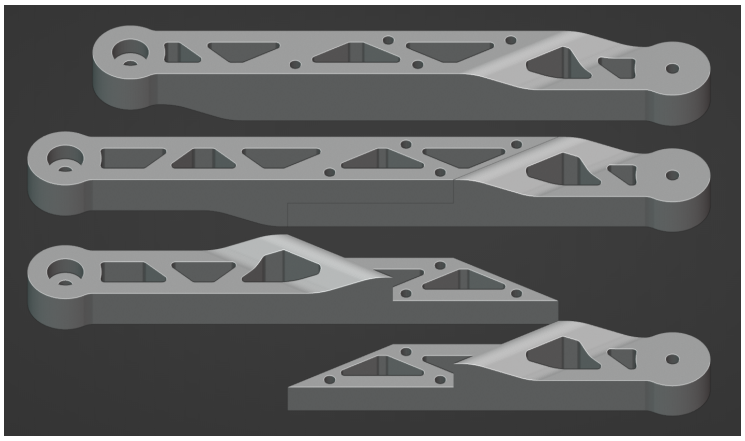

Figure 3.2: Screenshot of the distal arm modules from Blender. The topmost arm was rejected because it would collide with the other distal arm at sharp angles.

3.3 Pen Fixture and Lifter

The function of the pen lifting mechanism was specified, but there still needed to be a way to fix the pen onto this mechanism while ensuring its centering. two tapered vise jaws were printed that would close in on each other when a locking sleeve with a matching tapered opening is pressed down upon them. This sleeve was also horizontally fixed in the housing that would contain these jaws and the pen. To allow the pen to be pushed upon from below without the locking sleeve backing out of the housing, two pliable paddles were added to the locking sleeve, with slots for the paddles

to slip into creating a ratchet mechanism. The thickness of these paddles as well as the depth of the slots they fall into had to be adjusted a few times before the sleeve would reliably stay fixed in the position it was placed.

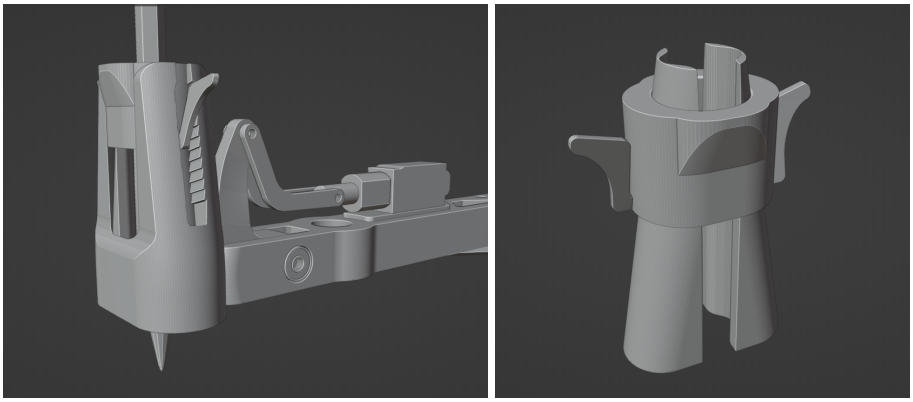

Figure 3.3: Screenshots of the Pen fixture models from Blender.

3.4 Wiring

Wiring was set up provisionally using breadboard jumper wires. By the time the circuit had to be mounted on the baseplate, custom connectors had been made using pin headers to reduce the height of the circuit and provide more secure connections. Clamps to hold down the protruding wires were also printed to allow for the right proximal arms to clear them.

The action and shutoff button were installed onto a 3D printed bracket. A button that allows the operator to switch power to the 12 V rail at their own discretion was added because the motors are powered by default until the program on the microcontroller is initiated.

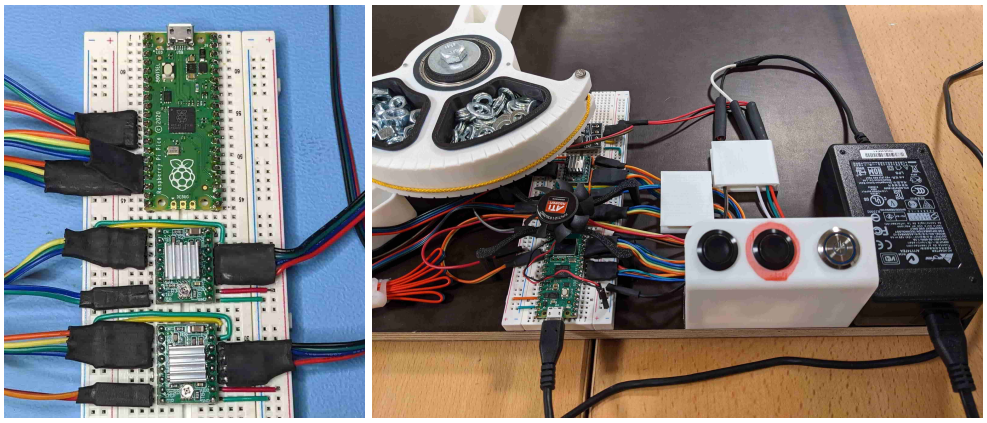

Figure 3.4: Photos of the circuit in progress and circuit mounted to the plate.

Chapter 4

Configuration

4.1 Movement Implementation

4.1.1 Coordinates

A coordinate system based on the the stepper motor positions was defined for clear and precise positioning. The coordinate components are integers and refer to the number of steps a motor has taken in either direction, making the values of this system dependent on the amount of substeps. To convert angles in radians to this system, the angle is multiplied with the empirically measured overdrive factor for the capstan drive, found as 8.04 and 8.03 for the left and right arm respectively, the total amount of steps per revolution for the motor, divided by the full rotation (2π or τ in radians) and finally converted to an integer. A code expression follows:

Listing 4.1: This implementation is not used in the final program, the full program code can be found in Appendix A.6.

```
1 def convertAngles(angle, overdrive_factor):
2     return (int(angle*overdrive_factor*total_steps/tau))
```

4.1.2 The "GoTo" Function

A function to smoothly travel to a given position in our "quasi-polar" coordinates was implemented. With the goal to move each axis at a constant speed while having a single step pulse time, an algorithm evenly distributing the amount of steps to take in the axis with fewer necessary total steps was devised. Inspired by rasterization algorithms from the field of computer graphics, the axis with the greater distance to be covered receives a pulse in every step, whereas the other axis is only stepped when doing so brings the distribution over the actually covered path closer to the desired distribution than not doing so. A reduced code snippet follows:

```
    if abs(delta_a) > abs(delta_b):
        quotient = abs(delta_b/delta_a)
        steps = abs(delta_a)
        for i in range(steps):
            step_a.on()
            actual_delta_a += 1
            if (actual_delta_b+0.5)/actual_delta_a < quotient:
                step_b.on()
                actual_delta_b += 1
            sleep_us(step_time)
            step_b.off()
            step_a.off()
```

Important to note is that this function does not trace straight lines in Cartesian space, this distributing of steps merely serves smooth motion.

Smoother motion

This implementation for moving from position to position worked, but appeared somewhat jerky and jarring at faster speeds, skipping steps in the worst case and losing position. A simple acceleration profile was added by adding a delay to the pulse time based on the progress of the movement. Phasing the delay out in the beginning and back in at the end allows the motors to handle the inertia of the arms. A maximum delay is defined and multiplied by one of two functions, one being an inverted semicircle that is exponentiated to exaggerate its peaks and valleys: $(1 - sin(arccos(2 \cdot (x - 0.5))))^6$ and the other being a polynomial function of a similar shape but less significant falloff: $((x - 0.5) * 2)^4$. These were derived through intuition and trial and error using the Desmos online graphing calculator (See Appendix Fig. A.3).

This proved satisfactory, allowing the tracking from and to different points quickly and smoothly. This function is intended to be used to cover distances while the pen is not touching the paper.

4.1.3 Tracing Files

Defining Paths

Before being able to trace any path, we need to define a format for these paths. Saving paths as Cartesian coordinate point sequences was chosen as it has the advantage of being able to reposition and re-scale the contents at runtime. The format chosen was CSV (comma separated values). A break in the path, where the pen should be lifted brought to the next point and set back down is signified by leaving a line empty. These files were generated by

specifying shapes in Blender and exporting them in the wavefront (.OBJ) file format, as its specification makes it easy to read and convert to our format using a short script (see Section A.4).

With a grand total of 264KB of sRAM memory available in the Pico (*Raspberry Pi Pico Datasheet: An RP2040 base microcontroller board* 2024), loading in all of the points at once was not an option, instead requiring to read from the file from flash memory line-by-line while tracing. To avoid possible stuttering if this reading and other operations were to take a significant amount of time before each movement, it was chosen to implement basic multithreading. The main thread reads from flash memory, scales and translates the Cartesian coordinates before converting them to the stepper positions. The main thread also calculates the step time for the next movement based on the real world distance to the next point and the set velocity. Once the main thread is finished, it waits for the stepping thread to complete the movement before launching the next thread with the values it prepared. This is repeated until the file is read to the end.

4.1.4 Pen Tip Position Compensation

Due to the nature of the pen lifting mechanism and worsened by the sag in the mechanical linkages, the pen tip moves laterally inline with the arm it is attached to when lifting or lowering. This causes unwanted streaks where the pen is lifted off the paper or lowered back down onto it. To prevent these streaks, the robot should move to keep the pen tip horizontally fixed while it is moved vertically. This was done only approximately, by calculating the stepper positions with d_a temporarily increased by a set distance determined through testing. This assumes that the pen tip is actually farther forward when setting it down or that it will be when lifting it off the paper. After some testing and adjusting the acceleration profile this proved to remedy the streaking to varying degrees likely depending most on the position of the robot.

Chapter 5

Evaluation

To evaluate the robot's performance, a benchmark had to be chosen. The logo of the National Aeronautics and Space Administration (NASA) was selected for its diverse visual features and complexity.

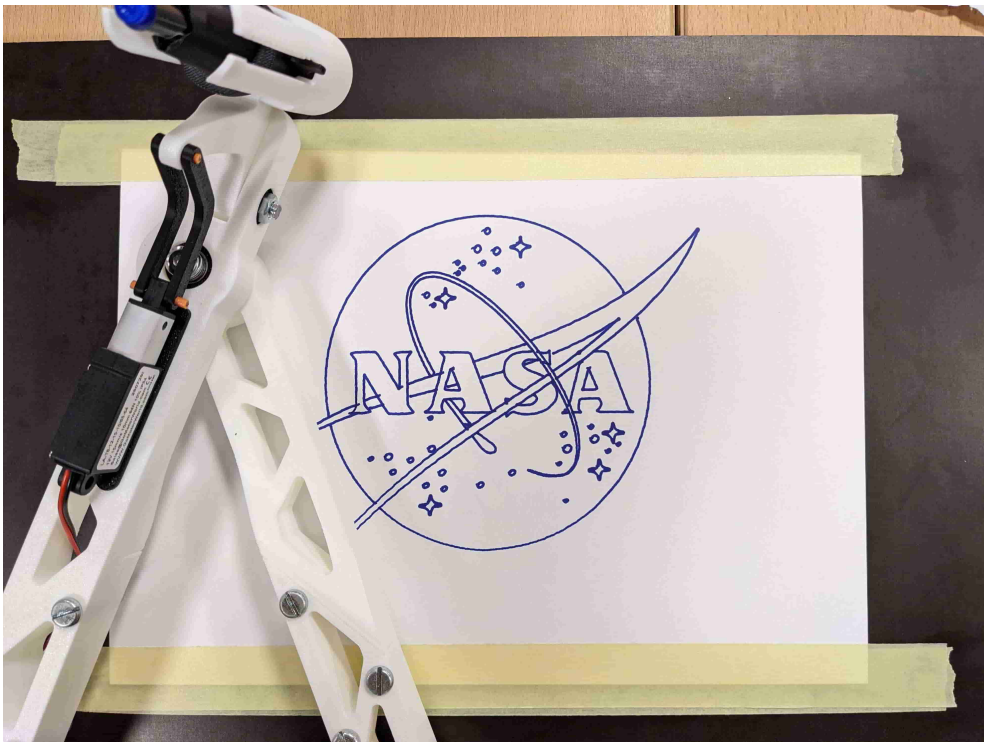

Figure 5.1: A photograph showing the NASA logo drawn by the robot.

We find the accuracy at this scale is satisfactory if the streaks caused by the pen lifting mechanism are ignored. The lines appear slightly wavy in places where they are not meant to be, leading to the belief that there might be oscillations disturbing the motion and or that the resolution of the stepper motors is insufficient with the current capstan drive ratio. The pen lifter is a bottleneck to speed; this is exaggerated by the pen tip position compensation, where the arm has to pre-position itself. The usage of a solenoid and a direct lifting system may be a better solution for artworks with many separate paths such as this benchmark. Tests using small scale artworks were also done. If pen lifting is avoided, the robot was fairly precise depending on where on the plotting area was drawn. This is most likely the case because the effective resolution the robot can draw at is not homogeneous across the plotting area. It was theorized that larger values for the offset o may increase homogeneity of resolution across the plotting area. Choosing a value of 0 for o yields a system equivalent to having polar coordinates, which would be the least homogeneous in resolution as the precision in angle needs to be increased dramatically as the radius is increased.

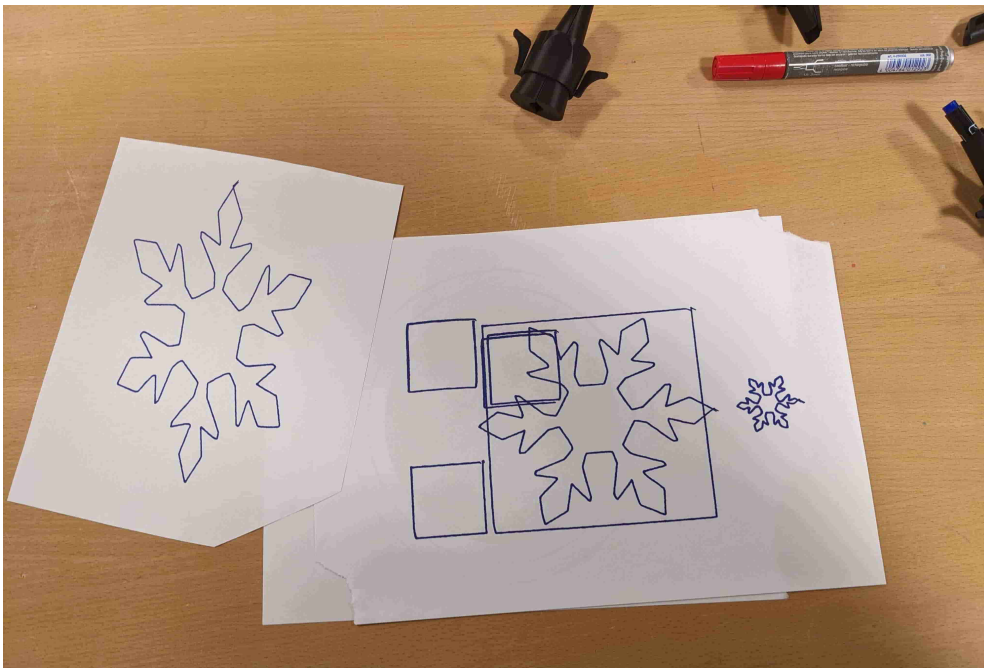

Figure 5.2: A photograph of drawings made by the robot during testing. This was before pen tip compensation was implemented.

Chapter 6

Conclusion

The robot's mechanical configuration demonstrates sufficient accuracy for the task of drawing and does so with a small number of bought parts, notably needing only two motors. However, the method by which the pen is lifted posed challenges that were not adequately resolved, leading to overall underwhelming performance. The robot's geometry makes it sensitive to mechanical play as its long linkages magnify inaccuracies and oscillations. This sensitivity proved to be manageable through the usage of methods that aim to minimize mechanical play. This without significantly complicating the assembly and adding cost, the primary example being the capstan reduction drive used to improve accuracy.

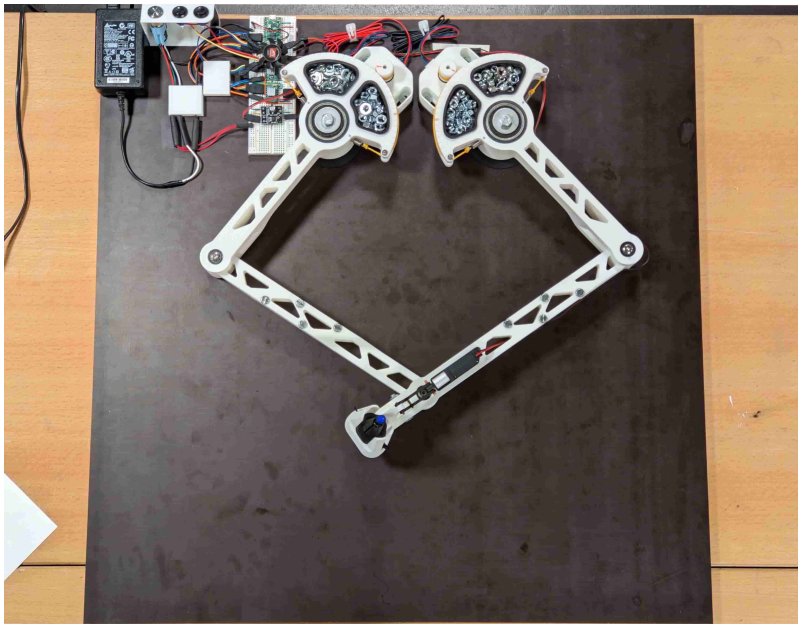

Figure 6.1: Photograph of the robot from above

Bibliography

17HS4023 Datasheet 2 Phase Stepper Motor (Mar. 4, 2021). Datasheetcafe. URL: https://www.datasheetcafe.com/wp-content/uploads/2021/03/17HS4023.pdf (visited on 2025-11-08).

Andreas Aristidouad, Joan Lasenby (May 9, 2011). *FABRIK: A fast, iterative solver for the Inverse Kinematics problem*. URL: http://www.andreasaristidou.com/publications/papers/FABRIK.pdf (visited on 2025-11-07).

DRV8825 Stepper Motor Controller IC (June 2014). Texas Instruments. URL: https://www.ti.com/lit/ds/symlink/drv8825.pdf (visited on 2025-11-07).

DRV8871 3.6-A Brushed DC Motor Driver With Internal Current Sense (PWM Control) (July 2016). Texas Instruments. URL: https://www.ti.com/lit/ds/symlink/drv8871.pdf (visited on 2025-11-07).

Effects of Microstepping in Stepper Motors (May 19, 2014). Oriental Motor. URL: https://youtu.be/tRoT3qpndbU?si=0UmhcS1bi2_xq88l (visited on 2025-11-07).

GreatScott! (July 24, 2016a). *Electronic Basics #23: Transistor (MOSFET) as a Switch*. URL: https://www.youtube.com/watch?v=o4_NeqlJgOs (visited on 2025-11-07).

GreatScott! (Sept. 11, 2016b). *Electronic Basics #24: Stepper Motors and how to use them*. URL: https://www.youtube.com/watch?v=bkqoKWP4Oy4 (visited on 2025-11-07).

Mazumdar, Anirban et al. (2017). "Synthetic Fiber Capstan Drives for Highly Efficient, Torque Controlled, Robotic Applications". In: *IEEE Robotics and Automation Letters* 2.2, pp. 554–561. DOI: 10.1109/LRA.2016.2646259. URL: https://www.osti.gov/servlets/purl/1340266.

MOS Microstepping Driver with Translator and Overcurrent Protection (Apr. 5, 2022). Allegro Microsystems. URL: https://www.allegromicro.com/~/media/Files/Datasheets/A4988-Datasheet.ashx (visited on 2025-11-07).

Musa, Aaed (May 31, 2024a). *Capstan Drive*. URL: https://www.aaedmusa.com/projects/capstandrive (visited on 2025-11-07).

Musa, Aaed (May 31, 2024b). *High Precision Speed Reducer Using Rope*. URL: https://www.youtube.com/watch?v=MwIBTbumd1Q (visited on 2025-11-07).

Pico-series Microcontrollers (2025). Raspberry Pi. URL: https://www.raspberrypi.com/documentation/microcontrollers/pico-series.html (visited on 2025-11-07).

Raspberry Pi Pico Datasheet: An RP2040 base microcontroller board (Oct. 15, 2024). Raspberry Pi. URL: https://datasheets.raspberrypi.com/pico/pico-datasheet.pdf (visited on 2025-11-07).

SKF (Aug. 2025). *BEARING INSTALLATION AND MAINTENANCE GUIDE: A practical guide to extend bearing service life*. URL: https://cdn.skfmediahub.skf.com/api/public/0901d1968024f02a/pdf_preview_medium/0901d1968024f02a_pdf_preview_medium.pdf (visited on 2025-11-09).

Kantonsschule Uetikon am See

Redlichkeitserklärung

Name _____ Vorname _____ Klasse _____

Titel der Arbeit _____

_____

Hiermit erkläre ich, dass ich die vorliegende Arbeit gemäss dem KUE-Reglement verfasst habe, das heisst im Besonderen:

- Ich habe diese Arbeit eigenständig verfasst.
- Alle Hilfsmittel, die ich verwendet habe, sind angegeben.
- Ich habe die Nutzung von KI-Tools (z.B. ChatGPT) gemäss Vereinbarung ausgewiesen.
- Alle wörtlichen und sinngemässen Übernahmen aus anderen Werken sind als solche gekennzeichnet.
- Die Leistung von Personen, die einen wesentlichen Beitrag zu dieser Arbeit geleistet haben (Betreuer/-in ausgenommen), habe ich ebenfalls erwähnt.

Datum Unterschrift

Appendix A

Appendix

A.1 Shader Nodes for the Plotting area

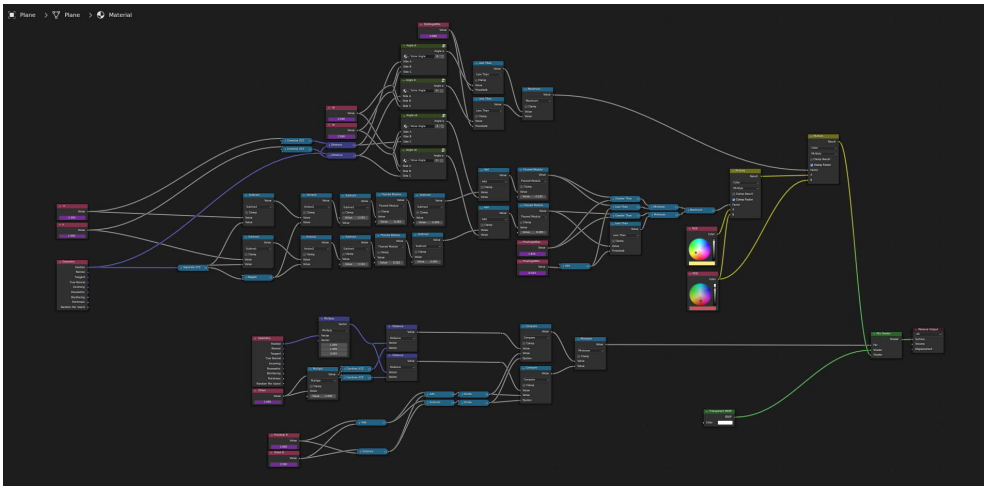

Figure A.1: A screenshot of the nodes used to render the plotting area with the first definition of the kinematic system

A.2 Draft for latching mechanism

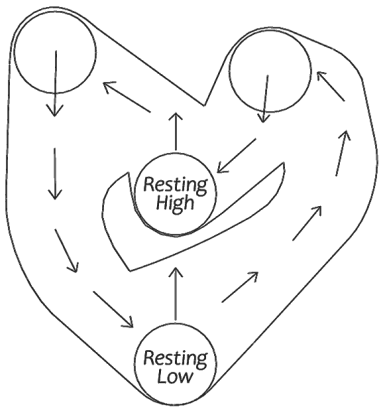

Figure A.2: A draft made for a latch mechanism that would use a pin running through a channel

A.3 Acceleration delay factor function plots

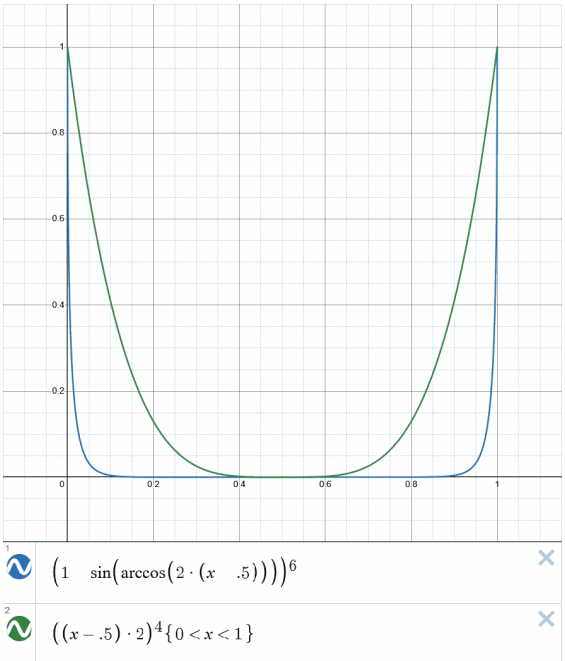

Figure A.3: A screenshot of the delay factor functions from within Desmos' online graphing calculator

A.4 CSV File Generation from OBJ

Wavefront OBJ is very easy to read.

Each line starts with a symbol to denote what follows

Vertices are marked with a v

Edges are marked with a | and contain the indices of 2 vertices

Faces do the same but for the indices of edges.

This script assumes that the vertices follow the order given by the edges this can be achieved in Blender by converting the mesh into a curve and then back into a mesh before exporting. This simply forces recalculation of the vertex orders. Checking this condition could be added to the script to prevent unexpected behaviour, were the vertex order incorrect.

```
# Converts OBJ files into csv for the drawing robot
# Assumes proper vertex order
coordinate_multiplier = 100 #mm
source = r"C:\Users\User\Desktop\2AP\nasa.obj"
output = r"C:\Users\User\Desktop\2AP\nasa.csv"
lastchar = "s"
with open(output,"w") as o:
    with open(source,"r") as s:
        for line in s:
            if line[0] == "o" and lastchar != "#": o.write("\n")
            elif line[0] == "v":
                content = line.split()
                o.write(f"{float(content[1])*
    coordinate_multiplier},{float(content[3])*-1*
    coordinate_multiplier}\n")
            lastchar = line[0]
```

A.5 Full Circuit Diagram

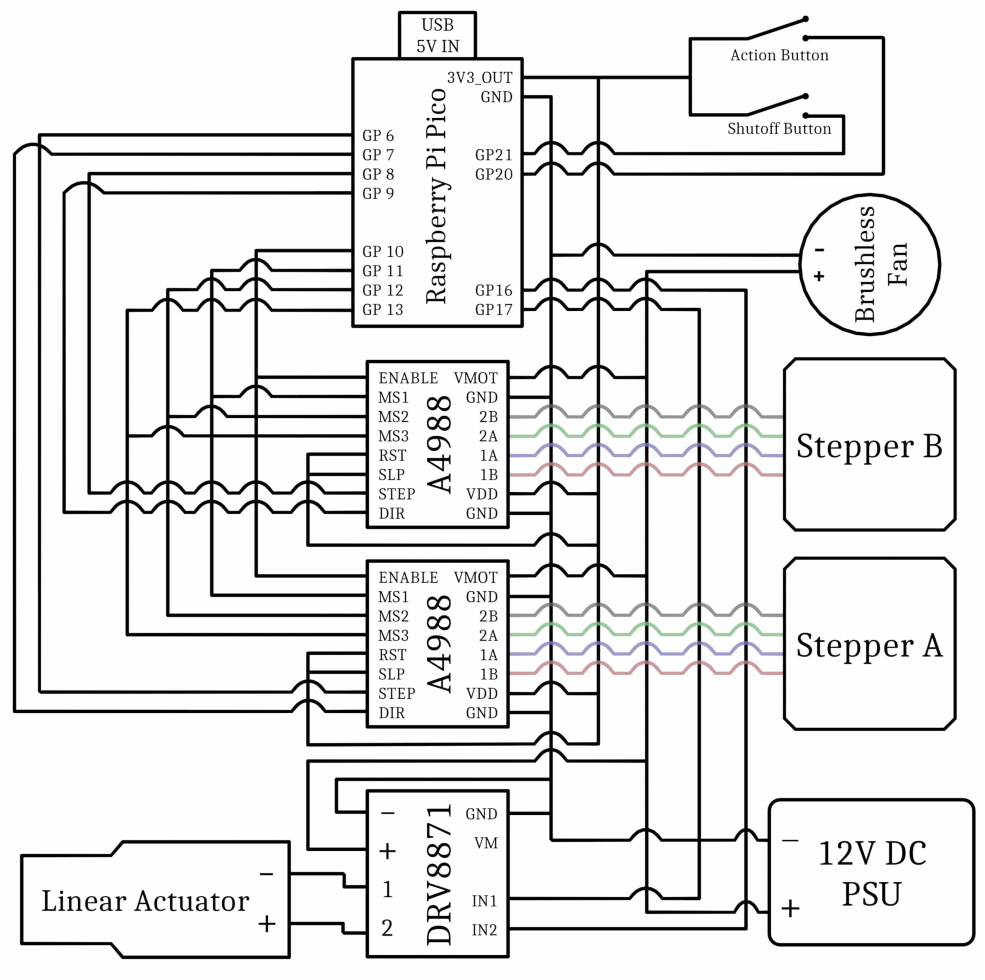

Figure A.4: The full circuit diagram for this project. Made in Blender

A.6 Full Code

Listing A.1: Full source code for the drawing robot.

```
# 2AP Drawing Robot "main.py" - Noah Werner
# Version 5, November 2025

# -- Configuration --

substep_index = 4 #(0->1, 1->2, 2->4, 3->8, 4->16)
substeps = int(2**substep_index)
total_steps = 200*substeps

base_speed = 3 # in mm/s for tracing
base_step_time = 8000//substeps # in us for simple moving
base_max_step_delay_factor = 20  # multiplied with base step time to make max step delay when
    damping

trace_filename = "nasa.csv"
trace_offset = (100,460) # mm x,mm y
trace_scale = 1.3

trace_settle_time = 500 #ms
thread_poll_time = 5 #ms

lift_offset = 10 # in mm
lift_duration = 300 # in ms
lift_max_step_delay_factor = 5


# - Physical Configuration -
# Dimensions, left is "a" right is "b", length units are always given in mm
pa =  200 # Proximal length
pb =  200
da =  360 # Distal length
db =  280
oa = -100 # Offset of proximal pivot from origin
ob =  100
# Capstan overdrive factors
overdrive_factor_a = 8.04
overdrive_factor_b = 8.03

# Reference angles (angles are stepper positions -> int)
reference_angle_a = int(-12.025/360 * total_steps * overdrive_factor_a) # acos(o/(p+d))
reference_angle_b = int( 12.025/360 * total_steps * overdrive_factor_b)
use_reference_angle = False

# -- Setup --
if use_reference_angle:
    home_angle_a = reference_angle_a
    home_angle_b = reference_angle_b
else:
    home_angle_a = int(0)
    home_angle_b = int(0)

angle_a = home_angle_a
angle_b = home_angle_b

# Motors setup
from machine import Pin
from utime import sleep_us
from utime import sleep_ms

dir_a = Pin(7,Pin.OUT)
dir_b = Pin(9,Pin.OUT)
```

```
step_a = Pin(6,Pin.OUT)
step_b = Pin(8,Pin.OUT)

motor_sleep = Pin(10,Pin.OUT)
ms1 = Pin(11,Pin.OUT)
ms2 = Pin(12,Pin.OUT)
ms3 = Pin(13,Pin.OUT)

def motorSleep():
    motor_sleep.on()
def motorWake():
    motor_sleep.off()

substep_configurations = [(0,0,0),(1,0,0),(0,1,0),(1,1,0),(1,1,1)]
ms1.value(substep_configurations[substep_index][0])
ms2.value(substep_configurations[substep_index][1])
ms3.value(substep_configurations[substep_index][2])

# Lifter pin & function setup
lifter_1 = Pin(16,Pin.OUT)
lifter_2 = Pin(17,Pin.OUT)
pen_up = True
def penUp():
    global pen_up
    lifter_1.on()
    lifter_2.off()
    pen_up = True

# Input button_input setup and shutoff function
from sys import exit as reset
in_key = Pin(20, Pin.IN, Pin.PULL_DOWN) # Black button
in_rst = Pin(21, Pin.IN, Pin.PULL_DOWN) # Reset button

def buttonInput(): # 0 -> idle, 1 -> go, 2 shutoff, 3 testsequence
    return in_key.value() + in_rst.value()*2

def shutoff():
    motorSleep()
    penUp()
    reset()


# Cardinal to angle space converter
from math import sqrt, sin, cos, acos, atan2, pi, tau

def transformToAngles(tx,ty):
    td = sqrt((tx-oa)**2+ty**2) # Distance to target from Proximal pivot
    da_angle = atan2(oa-tx,-ty) + acos((da*da+td*td-pa*pa)/(2*da*td)) # Opposed angle
    # Calculate secondary target position for arm b
    t2x = tx+(da-db)*sin(da_angle)
    t2y = ty+(da-db)*cos(da_angle)
    t2d = sqrt((t2x-ob)**2+t2y**2)

    pa_angle = -pi/2 + atan2(ty ,tx -oa) + acos((pa*pa+td *td -da*da)/(2*pa*td ))
    pb_angle = -pi/2 + atan2(t2y,t2x-ob) - acos((pb*pb+t2d*t2d-db*db)/(2*pb*t2d))

    pa_angle = (pa_angle+pi)%tau-pi
    pb_angle = (pb_angle+pi)%tau-pi
    return (int(pa_angle/tau*total_steps*overdrive_factor_a), int(pb_angle/tau*total_steps*
    overdrive_factor_b))

# Damped and threaded goTo functions in angle space
from math import copysign, sin, acos

def expsemicircle_step_delay(t, max_step_delay):
```

```
125     return int((1-sin(acos(2*(t-0.5))))**6*max_step_delay)
126 
127 def poly_step_delay(t, max_step_delay):
128     return int(((t-0.5)*2)**4*max_step_delay)
129 
130 def dampedGoTo(target_a,target_b, step_time = base_step_time, step_delay = expsemicircle_step_delay
        , max_step_delay_factor = base_max_step_delay_factor):
131     global angle_a
132     global angle_b
133 
134     max_step_delay = step_time * max_step_delay_factor
135 
136     delta_a = target_a-angle_a
137     delta_b = target_b-angle_b
138 
139     if delta_a > 0: dir_a.on()
140     else:       dir_a.off()
141     if delta_b > 0: dir_b.on()
142     else:       dir_b.off()
143 
144     if delta_a == 0 and delta_b == 0:
145         return
146 
147     actual_delta_a = 0
148     actual_delta_b = 0
149 
150     if abs(delta_a) > abs(delta_b):
151         quotient = abs(delta_b/delta_a)
152         steps = abs(delta_a)
153         for i in range(steps):
154             step_a.on()
155             actual_delta_a += 1
156             if (actual_delta_b+0.5)/actual_delta_a < quotient:
157                 step_b.on()
158                 actual_delta_b += 1
159             half_time = (step_time + step_delay(i/steps,max_step_delay))//2
160             sleep_us(half_time)
161             step_b.off()
162             step_a.off()
163             sleep_us(half_time)
164             if buttonInput() == 2:
165                 shutoff()
166 
167     else:
168         quotient = abs(delta_a/delta_b)
169         steps = abs(delta_b)
170         for i in range(abs(delta_b)):
171             step_b.on()
172             actual_delta_b += 1
173             if (actual_delta_a+0.5)/actual_delta_b < quotient:
174                 step_a.on()
175                 actual_delta_a += 1
176             half_time = (base_step_time + step_delay(i/steps,max_step_delay))//2
177             sleep_us(half_time)
178             step_b.off()
179             step_a.off()
180             sleep_us(half_time)
181             if buttonInput() == 2:
182                 shutoff()
183 
184     angle_a += copysign(actual_delta_a,delta_a)
185     angle_b += copysign(actual_delta_b,delta_b)
186 
187 
188 thread_active = False
```

```
189
190 def threadedGoTo(target_a,target_b,half_step_time):
191     global thread_active
192     thread_active = True
193     global angle_a
194     global angle_b
195     delta_a = target_a-angle_a
196     delta_b = target_b-angle_b
197
198     if delta_a == 0 and delta_b == 0:
199         return
200
201     if delta_a > 0: dir_a.on()
202     else:       dir_a.off()
203     if delta_b > 0: dir_b.on()
204     else:       dir_b.off()
205     actual_delta_a = 0
206     actual_delta_b = 0
207
208     if abs(delta_a) > abs(delta_b):
209         quotient = abs(delta_b/delta_a)
210         steps = abs(delta_a)
211         for _ in range(steps):
212             step_a.on()
213             actual_delta_a += 1
214             if (actual_delta_b+0.5)/actual_delta_a < quotient:
215                 step_b.on()
216                 actual_delta_b += 1
217             sleep_us(half_step_time)
218             step_b.off()
219             step_a.off()
220             sleep_us(half_step_time)
221             if buttonInput() == 2:
222                 shutoff()
223
224     else:
225         quotient = abs(delta_a/delta_b)
226         steps = abs(delta_b)
227         for _ in range(abs(delta_b)):
228             step_b.on()
229             actual_delta_b += 1
230             if (actual_delta_a+0.5)/actual_delta_b < quotient:
231                 step_a.on()
232                 actual_delta_a += 1
233             sleep_us(half_step_time)
234             step_b.off()
235             step_a.off()
236             sleep_us(half_step_time)
237             if buttonInput() == 2:
238                 shutoff()
239
240     angle_a += copysign(actual_delta_a,delta_a)
241     angle_b += copysign(actual_delta_b,delta_b)
242     thread_active = False
243
244 # Specialized Pen functions
245
246 def liftPenUpAt(point): # Lifts pen at point, the function assumes the position given is where the
        pen is at
247     global pen_up
248     global da
249     if not pen_up:
250         da += lift_offset
251         angles_compensated = transformToAngles(point[0],point[1])
252         da -= lift_offset
```

```
253         angles = transformToAngles(point[0],point[1])
254         steps = max(abs(angles_compensated[0]-angles[0]),abs(angles_compensated[1]-angles[1]))
255         lift_step_time = lift_duration*1000//steps
256         lifter_1.on()
257         lifter_2.off()
258         dampedGoTo(angles_compensated[0],angles_compensated[1],lift_step_time,poly_step_delay,
    lift_max_step_delay_factor)
259         pen_up = True
260 
261 def setPenDownAt(point): # Hovers to the compensated location then sets down the pen in the desired
     position
262     global pen_up
263     global da
264     if pen_up:
265         da += lift_offset
266         angles_compensated = transformToAngles(point[0],point[1])
267         da -= lift_offset
268         angles = transformToAngles(point[0],point[1])
269         steps = max(abs(angles_compensated[0]-angles[0]),abs(angles_compensated[1]-angles[1]))
270         lift_step_time = lift_duration*1000//steps
271         dampedGoTo(angles_compensated[0],angles_compensated[1])
272         lifter_1.off()
273         lifter_2.on()
274         dampedGoTo(angles[0],angles[1],lift_step_time,poly_step_delay,lift_max_step_delay_factor)
275         pen_up = False
276 
277 # Trace function
278 from math import sqrt
279 import _thread
280 
281 def lineToPoint(line, offset = (0,0), scale = 1):
282     if   len(line) == 0: return False
283     elif len(line) == 2: return True
284     line = line.split(",")
285     return float(line[0])*scale + offset[0],float(line[1])*scale + offset[1]
286 
287 def traceFile(filename, speed = base_speed, offset = (0,0), scale = 1):
288     with open(filename,"r") as file:
289         point = lineToPoint(file.readline(), offset, scale)
290         motorWake()
291         sleep_ms(trace_settle_time)
292         setPenDownAt(point)
293         last_point = point
294 
295         while True:
296             point = lineToPoint(file.readline(), offset, scale)
297             if type(point) != bool:
298                 angles = transformToAngles(point[0],point[1])
299                 distance = sqrt((point[0]-last_point[0])**2 + (point[1]-last_point[1])**2)
300                 steps = max(abs(angle_a-angles[0]),abs(angle_b-angles[1]))
301                 next_step_time = int(distance/speed/max(steps,1)/2e-6)
302                 last_point = point
303                 while thread_active:
304                     sleep_ms(thread_poll_time)
305                 _thread.start_new_thread(threadedGoTo,(angles[0],angles[1],next_step_time))
306 
307             elif point: # Lifts and sets down at next point
308                 while thread_active:
309                     sleep_ms(thread_poll_time)
310                 liftPenUpAt(last_point)
311                 point = lineToPoint(file.readline(), offset, scale)
312                 setPenDownAt(point)
313                 last_point = point
314             else:
315                 while thread_active:
```

```
                    sleep_ms(thread_poll_time)
                break
        liftPenUpAt(last_point)
        dampedGoTo(0,0)
        motorSleep()

# Testing functions
def testMoves():
    motorWake()
    dampedGoTo(0,0)
    sleep_ms(2500)
    dampedGoTo(int(100/360*total_steps*overdrive_factor_a),0)
    sleep_ms(3000)
    dampedGoTo(0,0)
    sleep_ms(2500)
    dampedGoTo(0,int(-100/360*total_steps*overdrive_factor_b))
    sleep_ms(2500)
    dampedGoTo(reference_angle_a,reference_angle_b)
    sleep_ms(2500)
    dampedGoTo(int(60/360*total_steps*overdrive_factor_a),int(-60/360*total_steps*
    overdrive_factor_b))
    sleep_ms(2500)
    dampedGoTo(home_angle_a,home_angle_b)
    motorSleep()

# -- Running Loop --

motorSleep()
penUp()
last_button_input = 0
counter = 0
while True:
    button_input = buttonInput()
    if button_input != last_button_input:
        counter = 0
    else:
        if counter < 6:
            counter += 1
        else:
            counter = 0

            if button_input == 1:
                traceFile(trace_filename, offset = trace_offset, scale = trace_scale)
            if button_input == 2:
                shutoff()
            if button_input == 3:
                testMoves()

    last_button_input = button_input
    sleep_ms(50)
```