Official Documentation

6-Axis Robotic Arm

10/10/2025

Table of Contents

Overview

The goal of this project was to design and fabricate a 6-axis robotic arm spanning 2 feet, completely from scratch, with a budget of $300 from the University of Illinois for presentation at the 2025 Engineering Open House (EOH). This project was associated with the University of Illinois's chapter of the American Society of Mechanical Engineers (ASME) as part of their Special Projects team. It was split into three subteams: mechanical, electrical, and programming. Mechanical focused on the motors, torque transmission, and all CAD. Electrical focused on high and low voltage circuitry, power, and connections and wiring to motors, encoders, and end-effectors. Programming was focused on motion planning, inverse kinematics, motor controls and embedded systems programming including encoder feedback, and instruction streaming from the PC to microcontroller. While each team had their own work to be done, there was obviously significant overlap between teams for many tasks. More concretely, mechanical and electrical collaborated on wiring and plug placement, and programming and electrical collaborated heavily on all motor control algorithms and embedded system programming.

The project's administrative structure had two leads, Andrew Park and Christopher Egly, along with three subteam leads: Nicole Canfield (Mechanical), Neil Maushard (Electrical), and David Savenok (Programming). Due to being a part of ASME, this project met officially twice a week for two hours each. However, near the end of March, we began to host additional meetings on the weekends for extra work time. In total, approximately 40 students worked on the robot in total, with an average attendance of about 15 students each meeting. There was a core group of about 20-25 students that completed the majority of the work, primarily on the mechanical team.

The end result, due to errors that will be described in the mechanical section, was a smaller 5-axis robotic arm that was presented at EOH. The entire project spanned approximately 7 months and was done entirely from scratch. No open source software, code, controls, or CAD were used, and all designs and algorithms were created organically from members of the team. This document is meant to be a relatively informal summary of the project as a whole including design decisions and their backings, problems, solutions, and other notable events.

Mechanical

This section will review all mechanical design aspects of the robot. This includes all CAD, general structure, motors and torque calculations and design, electrical mounts, and physical wiring.

Motors

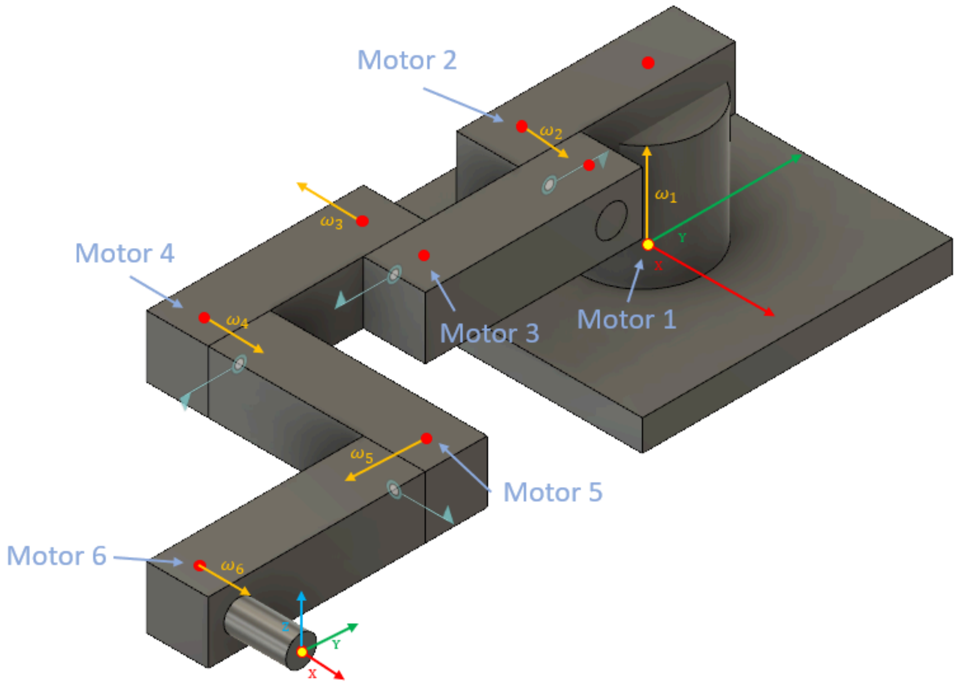

Figure 1. Rough CAD with motors and axes

The goal of this project was to build a 6-axis robotic arm that was 2 feet long. All joints were chosen to be rotational as opposed to prismatic, meaning we needed six motors. Due to the tight budget, a collection of approximately 15 motors were found with different torque-to-weight ratios. They were then run through a custom python script that found the best combinations of motors and link lengths by iterating through all likely combinations of motors and arm length combinations and calculating the remaining payload at full extension.

However, since the motors were not all purchased immediately, many choices went out of stock. This led to quick compromises being made without re-calculating the torque and weight specifications. In the end, 4 Nema 17 stepper motors were used (motors 1-4), 3 with 20:1

gearboxes, and one with a 50:1 gearbox, with the 50:1 gearbox being placed at motor 2. 2 DC motors were used (motors 5 & 6), one being approximately 0.5lbs, and the other about 2 ounces, respectively. Finally, for the end-effector, a 9g servo motor was used to actuate a custom padded gripper to grab jenga blocks.

Ultimately, the lack of accuracy in the torque calculations, along with the nature of the motor drivers, which will be explained in the electrical section, caused motors 3 and 4 to fail upon final assembly. The result was the removal of motor 3 and the arm length from motor 2 to 3, which reduced the weight and torque requirements enough for the entire arm to function properly.

Torque Transmission

The motors of the robot were primarily connected by basic 8020 aluminum extrusions, except for the final motors, which were simply connected from 3D printed PLA. The issue of proper torque transmission was solved by creating three main parts for each motor connection. First was obviously the 8020 connection itself, which took up the vast majority of the length between joints. Secondly, each motor sat in its own mount, which had connections to the 8020 from the previous motor. Finally, each motor had a shaft connected to a flange, which then screwed into a torque plate. The flange was screwed directly onto the motor shaft and was supported by a bearing which was connected to the motor mount. This allowed for better support of the flange and motor shaft, allowing the force to be distributed through the plastic mount and along the arm instead of solely by the motor shaft.

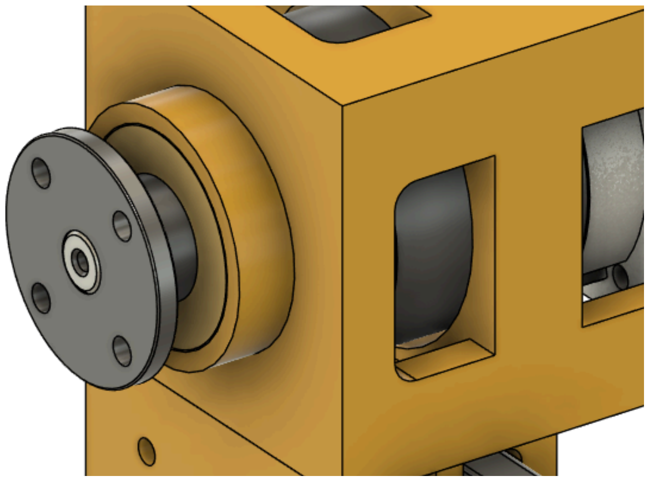
Figure 2. Flange connection with bearing and motor mount

Base Design

The base was essential for the functionality of the robot. Without a good way to support the arm, it would tip over and collapse on itself. In the end, we decided to mount the robot on a base made of PLA, then bolt the PLA base down to a large sheet of plywood to keep the robot steady. Motor 2 was then mounted on a structure made of 8020 aluminum, which was then placed on top of this PLA base and connected to motor 1 via a large torque plate. Some key features of this PLA base were the bolt holes for stability, encoder mount for motor 1 position tracking, an access hole for power and USB cables, and a snap-fit thrust bearing, which was the only point of contact between the base and the 8020 structure above apart from the motor 1 torque plate.

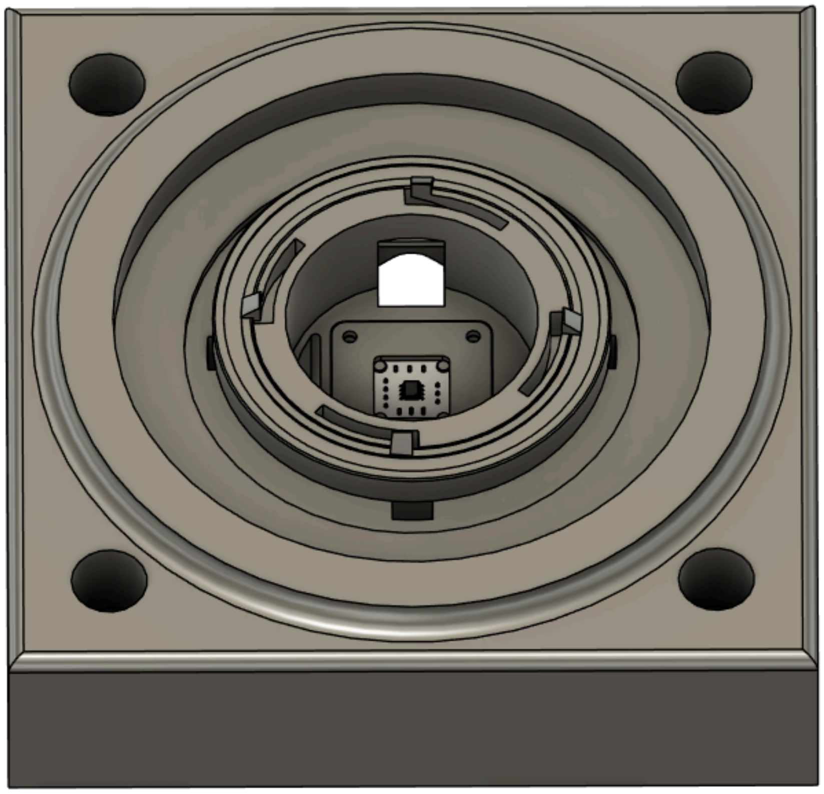

Figure 3. PLA base with encoder mount, cable access, and thrust bearing

After some basic wire counts were done, it was determined that there would be less wire required if all the motor drivers were kept at the base of the robot, so all drivers as well as the Arduino were mounted between motors 1 and 2, with most mounted directly on the motor 1 torque plate, as shown below.

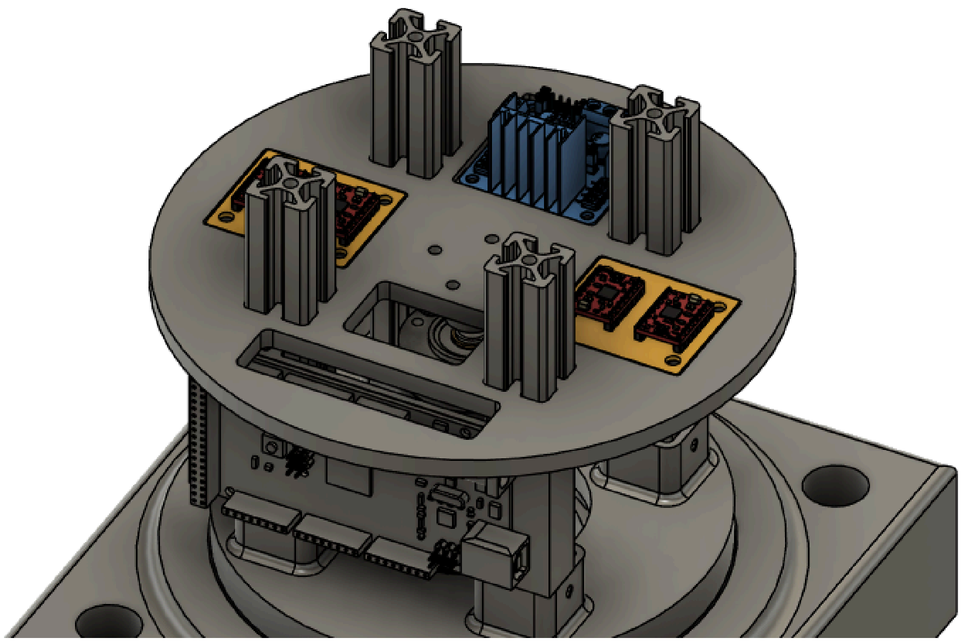
Figure 4. Motor 1 torque plate with electrical components mounted

After this section, however, the structure of the robot was mechanically very similar, with the classic torque plate/8020/motor mount combo on every joint. Minor changes were made at every iteration, and these are shown in the figures below.

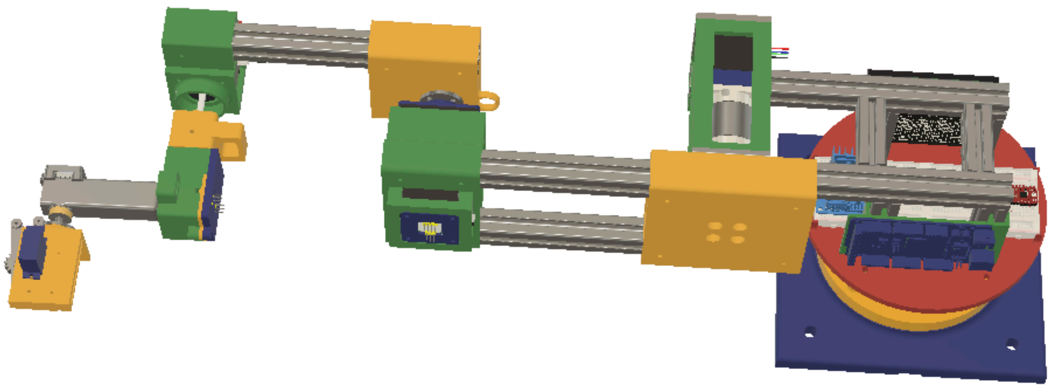

Figure 5. Full assembly 1

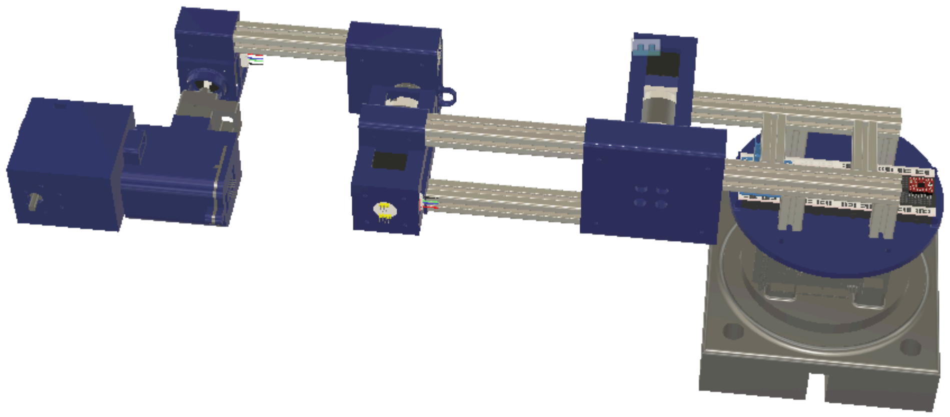
Figure 6. Full assembly 2

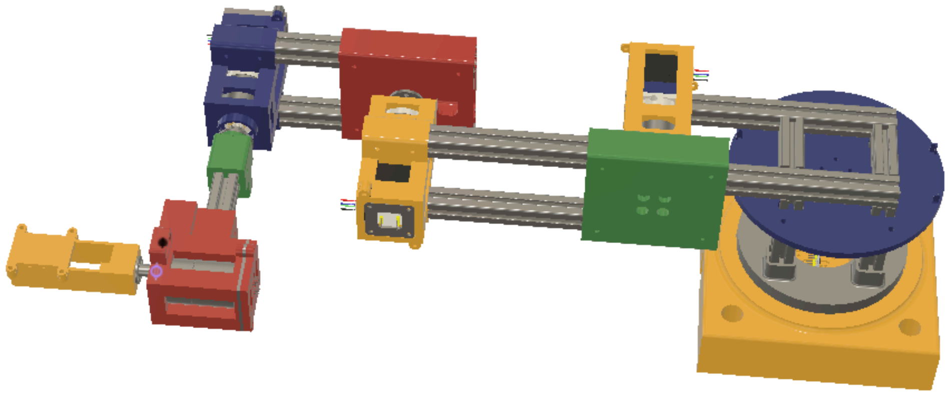
Figure 7. Full assembly 3

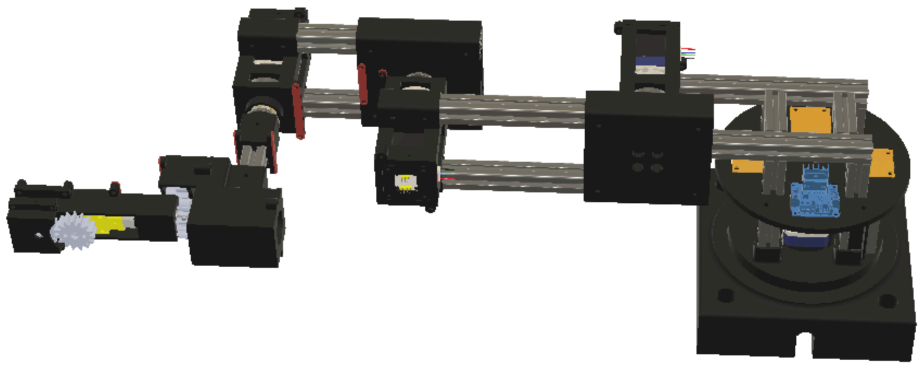
Figure 8. Full assembly 4

Encoder Mounts

To properly control the position of the motors, it was decided to implement magnetic encoder feedback on each of the motors. The controls and wiring will be discussed in later sections, but all of these needed to be mounted to properly read the position of the motors at all times. For the steppers, this process was relatively straightforward. Each of the motors had a shaft exposed at the back of the motor, so the magnets for the encoders were simply superglued to these shafts. The motor mounts then included a set of plates for the encoders to be mounted on in the back, each of which featured a halo to protect the electrical pins from damage. As seen below, there were 6 total mounting holes for screws. The four in the center actually screwed directly into the stepper motor making it a compact piece with the encoder and encoder mounts. The two on the outside then fixed the motor-encoder block into the mount.

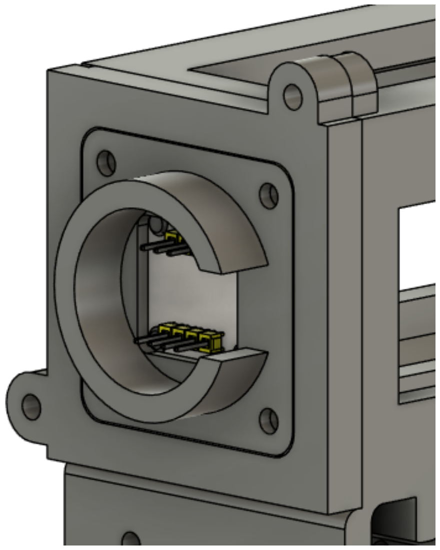
Figure 9. Stepper encoder mount

Unfortunately, the mounting for the DC motors was significantly more difficult. One of the DC motors did have a shaft in the back, but it was far too small to place a magnet on. When we attempted to cut the casing of the motor to further expose the shaft, the motor stopped functioning and we were forced to order a spare. After consulting a robotics professor at the university, we designed a geared system to run in tandem with the output shaft of the DC motors. This geared shaft then had the magnet glued on and was connected to the encoder system, shown below.

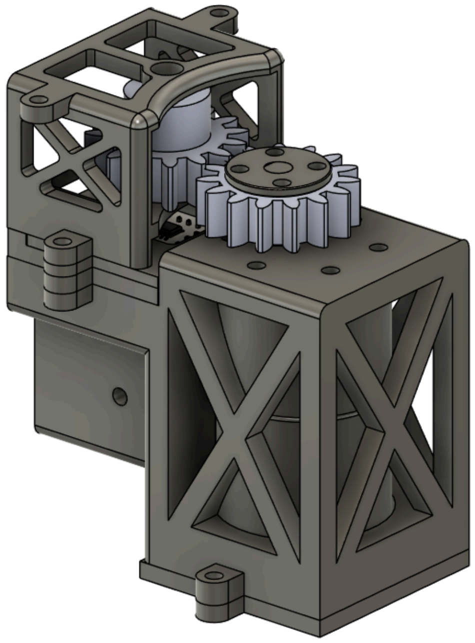
Figure 10. Motor 5 mount with geared encoder system (second shaft not pictured)

While this creative approach certainly functioned, due to our limited manufacturing resources and time, even the small backlash that was left in the geared system led to difficulties with the controls. If this part of the project were to be redone, great care would be taken to have tighter tolerances and to minimize the extra space used by this setup.

Wiring

The final mechanical consideration was the wiring, which turned out to be one of the most tedious parts of the entire project. While it was not difficult to come up with strain relief tactics, physically cutting and soldering the various wires took significantly more time than expected. Due to the rotating parts of the robot, we originally considered slip rings to allow the joints to be able to spin infinitely. However, these are extremely expensive and the amount of wires we needed to run along the arm was far too great. Instead, we set boundaries within the code and measured out a specific amount of slack for the wires at each joint before implementing the strain relief via small tabs on the motor mounts and torque plates, as shown below.

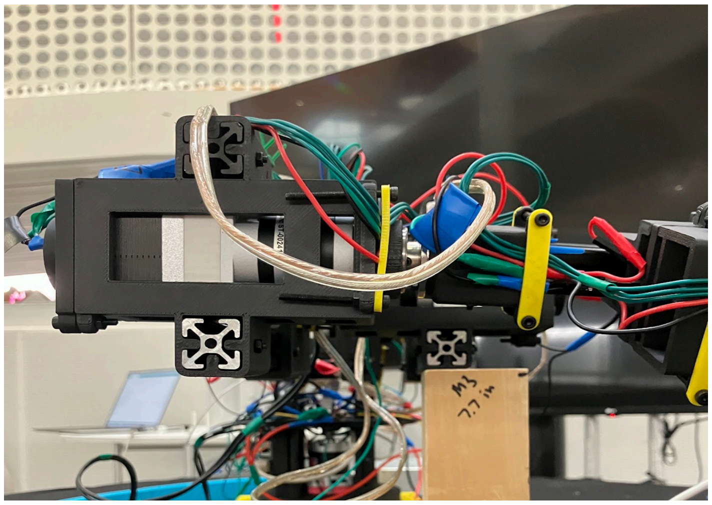

Figure 11. Wiring with strain relief

End-effectors

The final portion of the mechanical design was the various end-effectors for the different applications. While they were not actually put to use on the robot, they were constructed and all functioned independently. These included a servo-actuated claw, a rake, and a small LED box (which was actually used). The claw mechanism was a basic linear actuator using a small linkage-based system, as shown below. It was then programmed to grab jenga blocks and padding was placed on the grippers to keep the block in place even when lifted up. The LED rake was simply for dragging through sand, and the LED box featured a small bit of perf board with some resistors to control an RGB LED, which was used to light paint on the 5-axis robot end product.

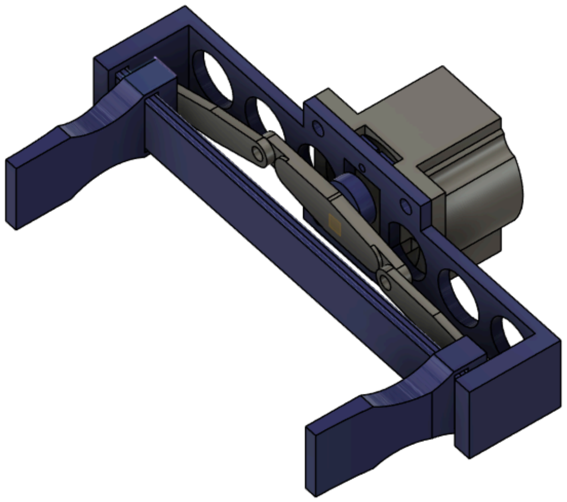

Figure 12. Jenga block gripper

Electrical

This section will review all electrical aspects of the project including the control scheme, despite it being embedded systems programming. These aspects include the power system, motor drivers, open and closed loop controls, microcontroller feedback processing and decision making, and timers and PWM generation.

Power System

All electrical systems need power, especially high-torque stepper motors. Luckily, we were provided with a 12V power supply from one of ASME's previous projects, leaving us with some extra money on the budget. This power supply was more than capable of running all of the motors, so it was run straight to each of our 5 drivers, and while we could have implemented a 5V regulator for the low-voltage system, the Arduino was already receiving power from the PC, so the low voltage system was simply run off of that board. This included the encoders, LEDs, and even the servo motor on the end-effector. A full schematic can be seen below, including the Arduino, 7 motors, 6 encoders, 5 drivers, and capacitors for voltage stabilization.

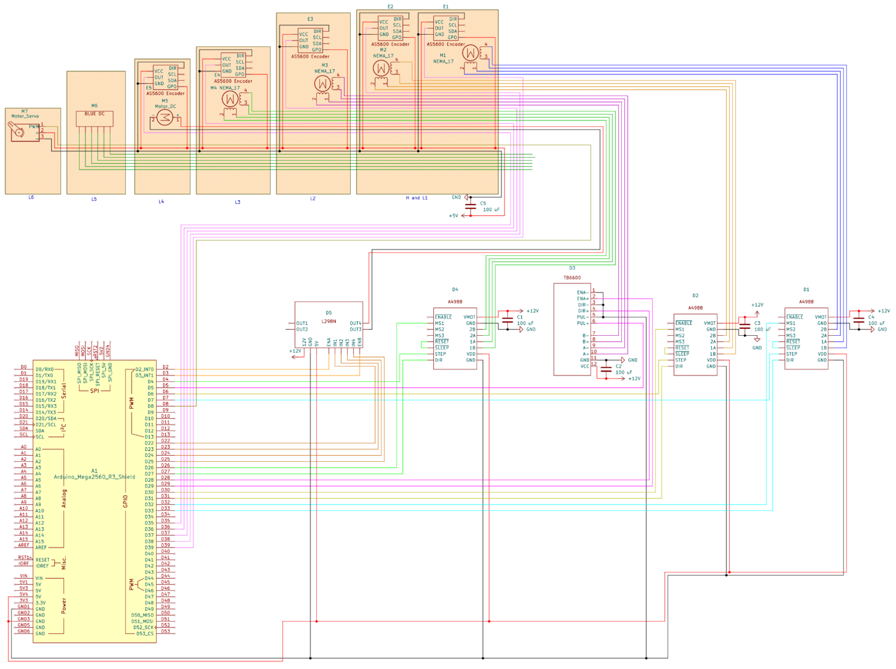

Figure 13. Full electrical schematic

Motor Drivers

For the two different types of motors, we had 2 types of drivers. The Nema 17 stepper motors were run off of A4988 stepper drivers and the DC motors were both run off of the same L298N DC motor driver, which was capable of controlling up to 2 DC motors simultaneously.

The A4988 drivers function by receiving a small electrical pulse (+5V) corresponding to one step on the stepper motor (1.2° before gearbox). The shortest pulse it would register was approximately 500μs, also found through experimentation. Based on this, we realized that you can set varying stepper motor speeds by changing the frequency of a PWM signal provided the high pulse is consistently 500μs, which played heavily into our control scheme. To reverse the direction of the motor, the driver simply had an additional direction pin.

The L298N drivers, however, functioned entirely differently and ran on a scaled voltage between 0-5V. Fortunately for us, this could very easily be run from the pre-set PWM pins of the Arduino. The L298N drivers also accepted reversed voltage to reverse the direction of the motor. The final feature of the L298N driver was that if both inputs were set to +5V, this would lock the DC motor in place.

To recap, both motors were run off of PWM signals, but the DC motors were controlled via the duty cycle while the stepper motors were controlled by the frequency. While the L298N driver had no issues and was very easy to use, the A4988 drivers were very small and not capable of running very high current. This was determined experimentally and it was found that the maximum current that they could pass reliably over repeated exposure was about 1.2A, which was below the maximum rated current for the stepper motors. This meant that there was no way to attain the maximum rated torque on the stepper motors without completely burning out the drivers, which was another reason the motors failed upon final assembly.

Encoders

Multiple of the most frustrating problems in the entire project came from the encoders. These AS5600 magnetic encoders had to be placed between 1-3mm away from their corresponding magnet, and wires with high enough currents looping or passing closely to the encoders would induce a magnetic field and cause noticeable errors, which often kept the DC motors from converging on a final position during testing. To make matters worse, the encoders relayed feedback via analog signal (0-3.3V) which was very difficult to accurately measure on the Arduino's ADC, as it fluctuated by a significant amount while the motor stayed completely still.

The largest problem came while we were first testing the encoders and running them from the Arduino's built-in 3.3V power supply. They seemed to be returning values nowhere near expected, increasing generally when expected to increase, but shooting back down at degree values that did not correspond with full rotations (i.e. the encoder would loop back to 0V after 410°). What we ended up discovering was that the Arduino itself was outputting a 4.2V source from the 3.3V terminal, causing the encoders to return values between 0-4.2V instead of our expected 0-3.3V values. From that point forth, we always checked the source voltage from the Arduinos and it tended to stay at 3.3V about 75% of the time.

The second problem came with the introduction of gearboxes. Since the encoders were all mounted to the backs of the stepper motors (all of which had gearboxes), it was necessary to track the number of rotations of the encoder to determine the absolute position of the output shaft. However, we began having problems due to fluctuations around the 0° mark, causing the absolute position to be frequently off by about 18° for the 20:1 gearboxes, corresponding to one full rotation of the base motor. This was solved by implementing a sort of "guessing" system by which the controls calculated the theoretical position of the motors before movement and proceeded to compare the value read by the encoder to the theoretical position. If they were within 1° of each other, the controls would consider the movement "complete" even if that range passed over the 360° mark. This allowed us to simply use the encoders as a sanity check instead of a more rigid open-loop control method for the stepper motors.

Surprisingly, this issue with counting rotations was also pervasive for the DC motors. Despite the fact that the encoders were connected directly to the output shafts, the total rotations needed to be counted to set limits of about 2 rotations to prevent wire twisting. However, it was not an issue in the same way it was for the steppers because the DC motors were constantly checking and updating the values of the encoders, so by comparing the current value to the last value, we were able to track the rotations relatively simply.

Motor Controls

The goal for the control flow was that all of the motors run simultaneously with all of the stepper motors finishing at the same time. However, with a mix of motors, simultaneous controls are exceedingly difficult to create without causing significant lags for the PID controls. To solve this issue, we discovered that we could use the built-in timer peripherals on the Arduino and manually generate a PWM signal of varying frequency using timer interrupts and Clear Timer on Compare Match (CTC) mode.

For the stepper motors, calculating the speed was relatively simple. Given the current angle and target angle, the number of steps were calculated for each motor with gearboxes taken into account. This speed was then turned into a frequency denoting the steps per second for each motor. Then, this frequency was used to calculate an "Output Compare Register" (OCR) value. The 16-bit timers would all be counting at a predetermined frequency (16MHz) up to $2^{16} - 1 = 65535$, and whenever they hit the OCR value, they would trigger an interrupt, which we configured to set a digital pin high, and reset the timer back to 0. We then set a second OCR value (let's call them OCRA and OCRB) to trigger an interrupt after 500µs which would set the pin to low. This OCRB value does not reset the timer, so the OCRA value does not have to take this second interrupt into account. We knew that the maximum frequency the stepper motors could run at was 500Hz through experimentation, so we always set the motor with the most steps to run at 500Hz, determine our total timescale, and calculate the other frequencies based on the global timescale. We then would set 4 different OCRA values on the 4 timer peripherals for the 4 stepper motors based on their desired frequency, and set the same OCRB values for a 500µs on-time. This entire calculation happened before beginning any of the motors, and the OCRA interrupts simply had the following lines: if the step count is below the desired step count, set the driven pin to high and increment the step count. Otherwise, set a motorXRunning flag to false. Once all motorRunning flags were false, the Arduino triggered another flag allowing it to read and parse further instructions. Recall that none of these interrupts were occurring within the main loop while the motors were running. This "behind the scenes" approach allowed us to focus on PID control of the DC motors within the main loop, which consistently called the function to update the DC motors.

This DC motor control was a basic open-loop proportional control algorithm based on the feedback from the magnetic encoders. Since the stepper motors did not require active monitoring or CPU power, the DC algorithm was able to take full advantage of the computational power of the Arduino while the motors were running. This was important because the function to update the DC motors, while not mathematically complex, had to constantly compare the current angle against the desired angle and determine the shortest path without going over the bounds of motion on every single call. It also had to make sure to properly increment/decrement the rotations counter if the current encoder reading was on the other side of 0° compared to the previous value, and use this information to determine the shortest path. The function also had to check whether the motors were within the acceptable range, and stop them if they were. Due to the backlash from the encoders and gear system, as well as friction from the gearboxes, the motors were only accurate to within about 8° total. In a professional setting, this would be utterly unacceptable, but since all error was due to a low cost of development, we considered this a relative success. Some improvements include implementing differentiation and integration into the control sequence as well as improving the backlash on the gears.

Overall, the control sequence was exceedingly complex and required months of development, constant testing, many lost hairs, and many sleepless nights. Implementing timer controls and directly manipulating register values for different modes was a new experience for everybody involved and definitely developed our skills and experience with embedded systems programming.

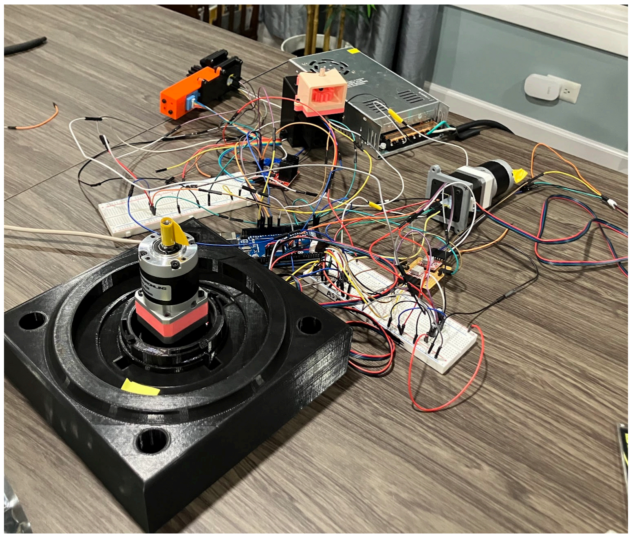

Figure 14. Test setup for combined stepper/DC controls

Programming

This section will go over all non-controls programming within the project. This primarily consists of motion planning, inverse kinematics, and instruction set generation and streaming. All algorithms written in this section were done in python and run on a PC, which then streamed instruction sets to the Arduino. It’s important to note that the Arduino did not have enough dynamic memory to even store one full instruction set, much less the code for the motion planning algorithms. For this reason, it was necessary to constantly stream the instructions to the Arduino instead of sending the data all at once.

Motion Planning

The primary motion planning algorithm we ran was based on the application of light painting. This is when you take a long-exposure photo and “paint” a picture with an LED. The concept was to upload an image, process it, and “paint” it using the robot. The first step in this process was processing the image and turning it into a set of spatial coordinates for the robot to travel to. This is what is encompassed in the motion planning section.

First, the image was run through a Canny edge-detection algorithm to identify the primary contours. These contours were then run through a Douglas-Peuker approximation algorithm to remove redundancies and reduce the number of points, which led to a significant reduction in the amount of time it took to recreate an image. Finally, all contours less than 100 square pixels in area were removed to reduce time even further. To determine the color of the LED for each contour, the image was blurred and inpainted along each contour, and a selection of RGB values was taken and averaged. Each contour was then assigned an RGB value along with the set of points in image coordinates.

The contours in image coordinates were then projected onto the 3D spatial plane with the origin at the base of the robot. It was ensured that the image bounds were projected onto a portion of the plane that was fully accessible to the robot. An example of the entire process is shown below. This set of coordinates marked the end of the motion planning portion of the python algorithm.

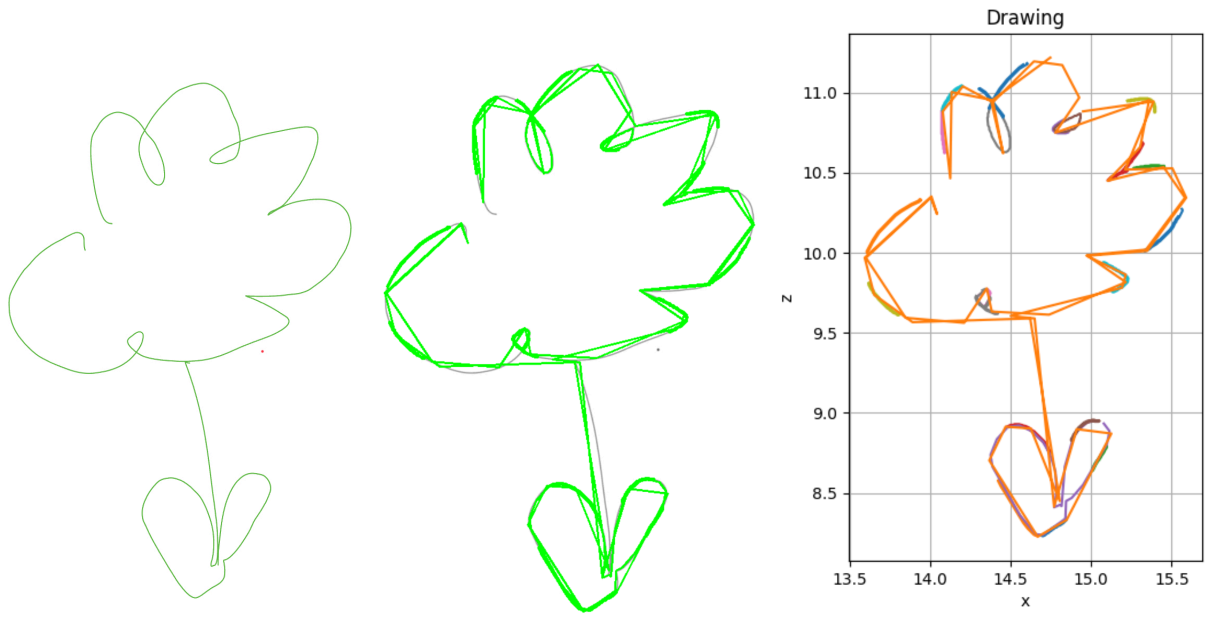

Figure 15. Original image (left), contour generation overlaid onto original image(middle), and projection into 3D space (right)

Inverse Kinematics

For the image algorithm specifically, the inverse kinematics were done by hand using simple trigonometry. Each of the points were run through a simple trigonometric algorithm that returned the angles of the motors that were required to place the end-effector at that point in space. However, for other algorithms that were not used in the final product, a Jacobian-based, matrix exponential, Newton-Raphson iterative approach was used to converge on a final solution.

The original pose of the end-effector in the home frame is given by a matrix M, consisting of a 3x3 rotation matrix, 3x1 position matrix, and 1x4 matrix to finish the square. The final pose of the end-effector is then represented by a transformation matrix T in the same form. A 6x6 matrix consisting of the screw axes for each joint is also given as an input, which are calculated using the axes of rotation as well as the position of the joint in the home orientation. To begin the Newton-Raphson method, an initial guess for the final angles is also input, as well as acceptable threshold values that signal the end of iteration. The final result is a set of angles that place the end-effector in the desired position and orientation.

Instruction Sets

This entire idea of streaming data in packages occurred due to the issue of memory on the Arduino. The control program took up about 60% of the Arduino’s dynamic memory, meaning there was no way that it could possibly store an instruction set of 10,000 characters. If we ever wanted to write an instruction set that could allow the robot to draw even a basic picture, we would need to have the data constantly streamed to the Arduino, so we came up with this custom streaming method and instruction set language to bypass the memory issue.

Before parsing the 3D spatial coordinates, the python script checked to ensure that there was proper communication between the Arduino and the PC. This involved a simple probe to see whether the device was recognized under a serial port. If it was, then the program moved on to the next step.

Once the 3D spatial points were all converted into sets of angles for the motors, these were encoded into a custom “instruction set” with delimiters denoting the beginnings and ends of various “instructions” including commands like: begin calibration, move, wait, change LED color, and end program. These were all merged into one large string, primarily consisting of move commands with LED commands sprinkled throughout.

This massive string was then broken up into sets of commands no longer than 1500 characters, which was then streamed along the USB cable to the Arduino serial port, which parsed it upon next availability. Only once the Arduino returned a flag signaling that the instructions were completed did the script then parse and send the next chunk of instructions.