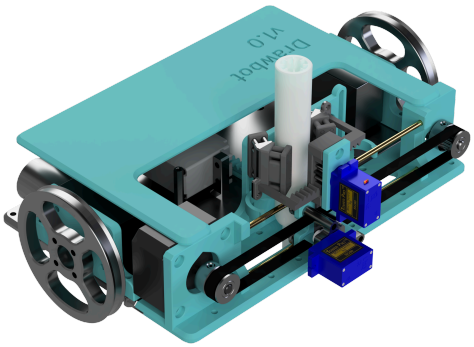
DrawBot

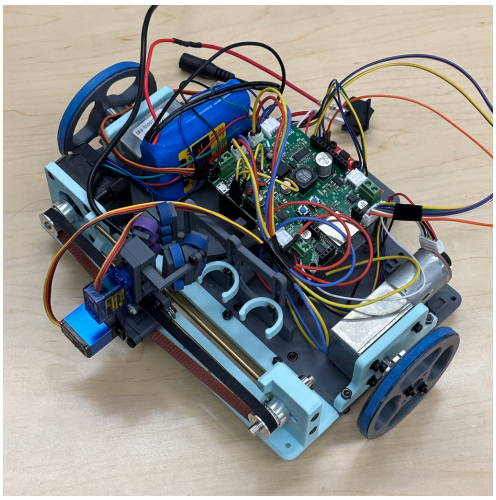
EECS 473
Fall 2025

Vivien Lin, Colin Riker, Shruti Shah, Aryan Thomare, Cindy Yang, & Prisca Yim

I. Introduction & Overview

Our project, Drawbot, is designed to help instructors and educators more efficiently present their class materials. DrawBot allows instructors to offload a part of their lecture preparation to an automated robot by simply uploading an image via a web app and sticking the robot to a magnetic whiteboard surface. Drawbot then reproduces the given image onto the whiteboard surface.





(fig. 1, & 2, The Complete CAD Render & Fully Assembled DrawBot)

We believe Drawbot would give instructors more time and capacity to focus directly on their students as the process of writing lecture material is offloaded to a whiteboard robot while also increasing readability, accuracy and detail of more complicated diagrams and notes for students. The concept of Drawbot is similar to a whiteboard drawing/erasing robot developed by HTX Studio.

Overall the project had seventeen design goals, of which all core aspects of the robot were achieved in some form. Several of the project's stretch goals like tool changing received partial implementations. There were some limited implementation issues left in g-code interpretation for driving commands and issues in robot performance, resulting in a nearly feature complete device and nearly completing the final project deliverables we set out to achieve.

II. Project Details & Descriptions

DrawBot High Level Overview

a. Goal

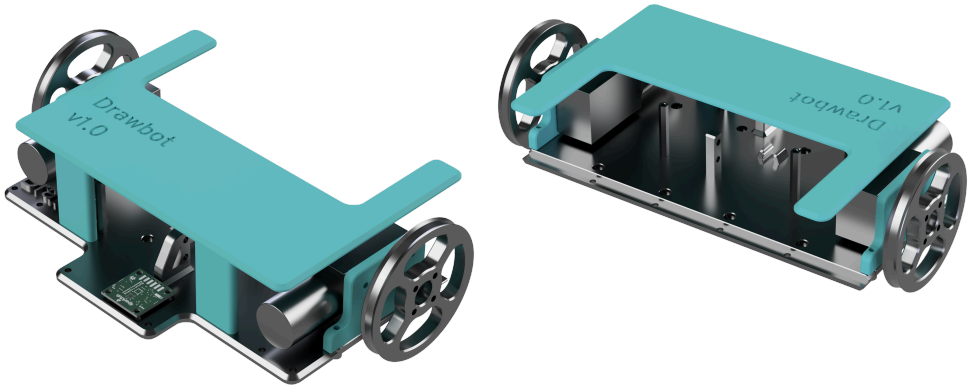
The primary goal of this project is to design and build an autonomous wall-climbing robot capable of taking a digital image as input and reproducing a scaled replica on a whiteboard. The robot should operate with minimal human intervention, handling both navigation and drawing tasks.

b. System Concept

The system integrates four major subsystems:

i. Drive System

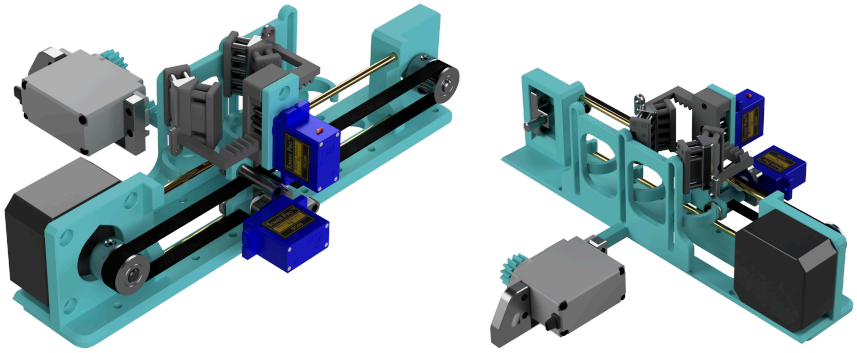
1. A motorized chassis designed to move smoothly across the vertical surface of a whiteboard.
2. Precise control of wheel speed and direction allows the robot to follow planned paths and maintain alignment.



(fig. 3 & 4, The Drive Base)

ii. Drawing Mechanism

1. A marker holder that can grip standard whiteboard markers securely.
2. Gantry to move the marker perpendicular to the chassis, enabling faster drawing.
3. A lift-and-lower mechanism to control when the marker contacts the board, ensuring clean starts and stops.
4. A tool-change feature that allows the robot to swap markers, enabling multi-color drawings.



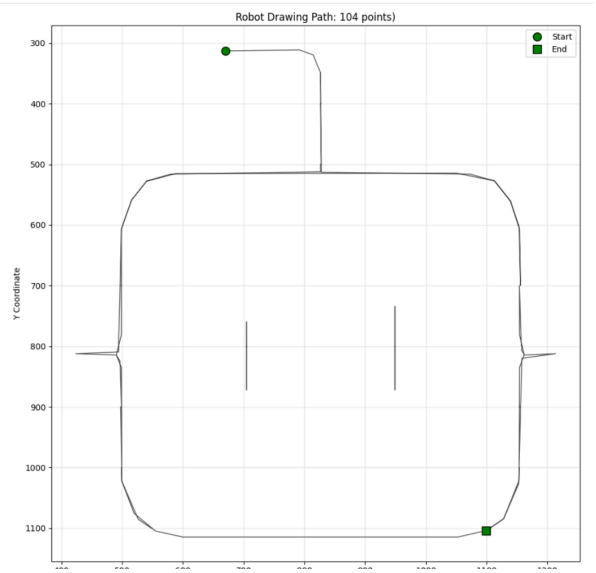
(fig. 5 & 6, the drawing subsystem including the draw gantry, tool head, and tool changer)

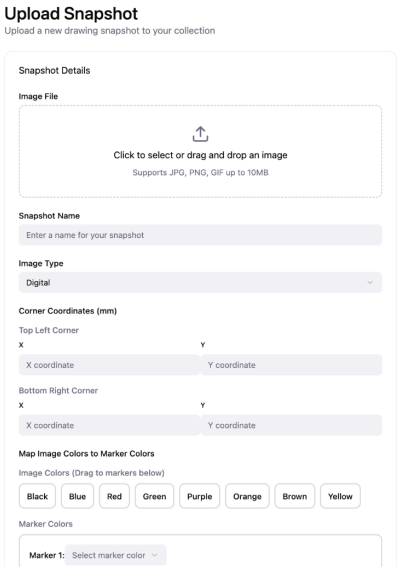
iii. Localization and Sensing

1. Sensors (e.g., encoders, webcam, and optical sensor) to track the robot's position on the board.
1. Localization algorithms and sensor fusion to correct drift and maintain accuracy over long paths.
2. Integration of CV feedback with peripheral device input to guide the robot to exact coordinates for drawing.

iv. Web UI

1. User interface to upload images and initiate robot drawing
2. Computation of image to g-code
3. Optional OCR to optimize handwriting
4. Process CV to localize the robot's absolute position
5. System orchestration





(fig. 7 & 8, The web application)

Together, these subsystems allow the robot to intake an image, process it into a set of drawing instructions, and execute those instructions on the board with precision.

c. Feasibility

The project's feasibility was questionable, as it requires solving several complex mechanical and control challenges simultaneously. Mechanically, for instance, the robot had to be lightweight yet strong enough to attach securely to a vertical whiteboard, while also incorporating a precise drawing mechanism and a tool-changing system. On the control side, achieving accurate localization is difficult because the robot must track its position in real time and compensate for drift, slippage, and gravity. These demands increased design complexity and reduced feasibility compared to simpler ground-based robots. However, with careful integration of lightweight materials, robust localization strategies, and modular subsystems, the project remained achievable.

d. Design Constraints

1. Time

A functional system was required to be demonstrated by December 4th, which gives a total timeline of around 2 months to complete the project. However, integration of PCB, drawing mechanism, controls, and software pipeline was delayed until the week of the demo. The team exceeded the deadline and demonstrated two days after the expo. This

means our time exceeded the time constraint allotted.

2. Budget

With a budget of $1500, our team stayed well below this, with a total accumulated cost of $778.62. The budget constraint was well met, staying ~65% below the limit. Budget savings were achieved by assembling PCBs in-house, reusing lab motors, and supplying personal materials.

3. Weight

Our design needed to be lightweight and compact. Creating a wall-climbing robot means that we were going against gravity, and our attachment system should be able to handle the weight of the robot to accurately maneuver it. Using the permanent magnet pulling force as a benchmark, we wanted the entire weight of the robot to exceed no more than 10 lbs. Our robot met these requirements, as it had a final weight of **3.25 lbs.**

4. Size

Like the weight constraint, size must be limited to create a lower center of gravity relative to the whiteboard to ensure attachment. This limited the size of text we could reproduce, as the gantry must be sized appropriately. The final size of the robot base was **170x210x60mm** (260x210x120mm including wheel).

5. Unit Cost

A lower unit cost is ideal, as it makes our robot competitive. Our DrawBot is a unique solution to the wall-drawing domain. However, existing listings for vertical drawing gantry systems or wall climbing robots cost $2500-$5000. Our unit cost is **$325.33**, which is significantly below that threshold, making our design cost-effective.

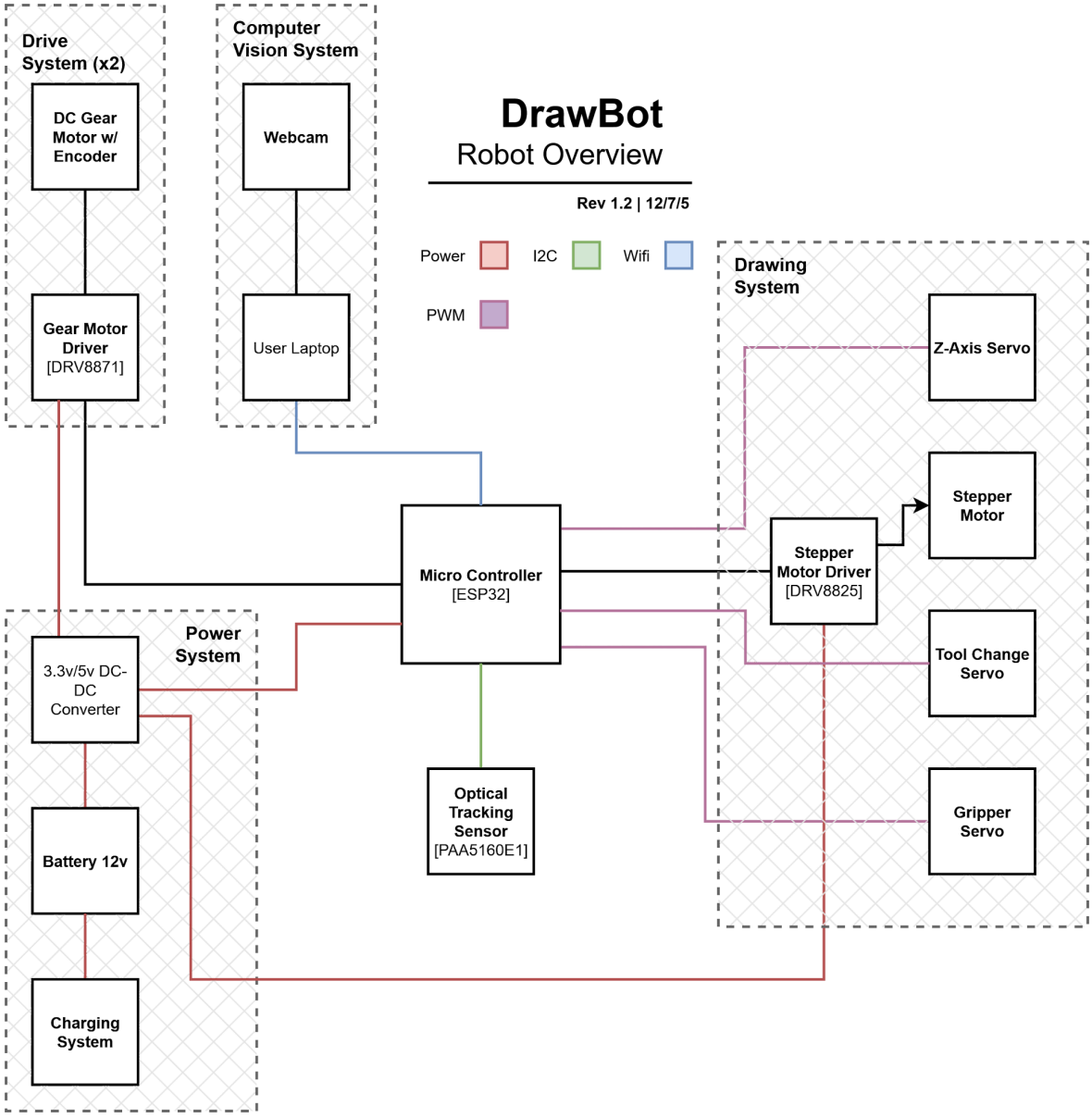
6. Functionality

Our primary focus is to make our product easy to use for instructors. If the device requires more time to set up and program than to simply draw the figures, instructors would avoid using our product. Therefore, one major design requirement is that our product should be designed to work “right out of the box”. This means that our design should be fairly self-contained with minimal modifications to the existing whiteboard surface. In terms of programming the robot, users should be able to upload their images and press a button to begin drawing. The drawing itself should not take very long to output, as this would reduce the functionality of our system.

Several functionalities were not achieved, as our image-to-whiteboard drawing pipeline is not operational, our speed is fairly slow, and the time for setup exceeds the time required to draw the image yourself. However, we did not modify the whiteboard, and our robot is self-contained and portable such that the system would be easily transported onto other magnetic whiteboards.

iii. System Architecture

An overview of the drawbots technical systems and a detailed breakdown of their function



(*fig. 9 The DrawBot Block Diagram*)

Component	Usage
ESP32 S3 WROOM 1 Microcontroller	The ESP32 S3 microcontroller was used to communicate with the user laptop wirelessly using its on board wifi capability. It was also used for communicating with peripheral devices, such as reading the encoder data, optical angle, and battery life, and writing to the stepper and dc drivers for drawing movement and driving respectively.
DC Gear Motors with Encoders	High torque motors were used to counter stalling resulting from gravity while the robot is traversing upwards. Encoders were used to calculate speed of individual motors.
DC Motor Driver (DRV8825)	A DC motor driver was used to control the gear motor speed and direction using PWM.
Webcam	The webcam was used to provide live video feed of the robot. It is primarily used for computer vision to accurately localize the absolute position of the robot and correct robot position from slippage.
AruCo Tags	These were used to define whiteboard boundaries, help localize the robot using known sizes and distances. A single AruCo tag on the robot likewise shows the absolute position of the robot.
User Laptop	The user laptop was used to interface with our robot. It performs the computation of the image to g-code and CV for localization.
Optical Tracking Sensor	The optical sensor was used for angular localization.
Stepper Motor Driver	The stepper motor driver was used to control direction and speed of the drawing gantry.
Stepper Motor	The stepper motor was used to move the marker perpendicular to the chassis for enhanced drawing speed and efficiency.
Z-Axis Servo	The Z-axis servo was used to lift the marker on and off the board between transmit and drawing periods.
Gripper Servo	The gripper servo was used to grab markers from the tool-change rack.
Tool-Change Servo	The tool-change servo extended and retracted a rack and pinion tool-change rack.
Charging System	The charging system enabled on-pcb charging via a micro USB cable.
12V Battery	A 12V battery was used to power the stepper and DC motor drivers. It was

	also used to power the PCB.
12V to 5V Buck Converter	A 12V to 5V buck converter was used to convert the battery voltage to a lower voltage without losing significant power dissipation.
5V to 3.3V LDO	5V to 3.3V LDO was used to convert the noisier 5V to a less noisy 3.3V for the ESP32, optical sensor, and USB to UART controller to use.
USB to UART Controller	The USB to UART controller was used to flash the ESP32 microcontroller directly from a micro USB cable.

III. Milestones, schedule, and budget

Over the course of the project every main milestone and budgetary constraint was met while our schedule was constantly changing and updating to better reflect the current blockers.

a. Milestones

The project milestones (see *fig. 10*) did serve as a useful metric and held us accountable but could have been more specific and should have been adjusted as time moved on. We achieved every core requirement of our milestones but they were often far too easy or didn't represent the actual challenges we needed to have addressed by that point. It likely would have benefitted us to submit an addendum or correction after milestone 1 for milestone 2.

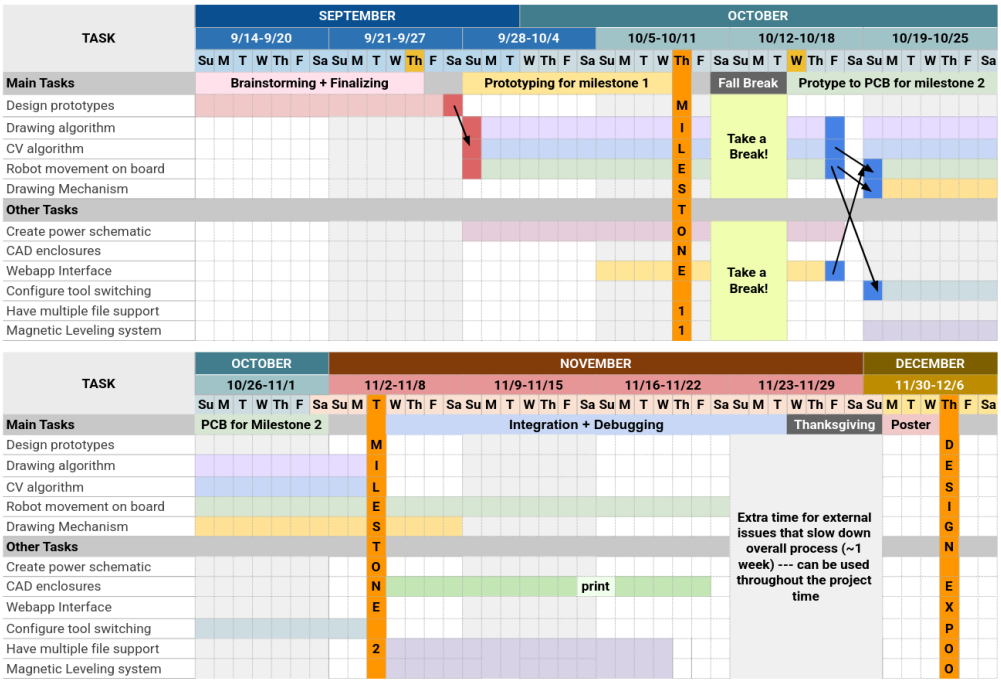
Milestone 1:
Prototype robot moves on whiteboard given predefined path or via remote control
Drawing algorithm can read in image and output robot's path
CV or sensor can locate robot's position, and display x,y coordinates on console
Robot can erase a small drawing on whiteboard
Milestone 2:
Robot PCB should be designed, ordered, and on its way
Laptop can take an image and output a path that the robot follows
Robot PCB fully assembled
CV algorithm can correct robot's erroneous movement

(fig. 10 Milestone 1 & 2, Bolded items were required, the rest were stretch goals)

b. Schedule

The initially proposed schedule for our project was wildly different from how the project

unfolded and represents one of the largest areas of improvement. Throughout development the biggest deviation from our schedule came from the fact much of our mechanical and electrical design required iterations and debugging well after initial prototypes were done. These both represented areas where blockers in testing and development for our software stack arose. There were three primary blockers however that set project timelines back more than the rest: the drawing systems mechanics, incorrect IC footprints, and poor, slippery tires.



(fig. 11 The Initially Proposed Schedule)

Mechanically, the drawing system represented a highly coupled system especially once tool changing became a goal for the project. Designing grippers, gantries, tool clips and fitting all the servos and steppers motors to make it work took nearly a month of effort and required a complete redesign over thanksgiving break.

PCB design was also a time consuming process due to the lead time required for manufacturing. In our case, a singular bad footprint for our LDO alongside switched RX and TX pins caused us to respin our design, adding additional manufacturing, debugging and assembly time into our schedule.

One of the most reoccurring issues was tire slippage, this directly delayed robot tuning and testing many times. This frequently added hours or days into any given test while we attempted

to resolve our traction issues. This added random delays across the board throughout the latter half of the semester.



(fig. 12 A stack of failed robot tire designs)

Overall,while many little issues arose that impacted scheduling, the drawing system design, the PCB respin, and continuous tire issues pushed the project the furthest off our timelines and schedules.

c. Budget

Our team had a budget of $1500. Given that existing whiteboard or wall-climbing robots are either nonexistent or expensive (>$2500), we had a lot of room to spare for expenses. Additionally, we did not need to purchase many speciality ICs or devices, keeping our cost very manageable. We had a final expenditure of $778.62, keeping us well below the budget.

IV. Lessons learned

Overall, the project taught us many valuable lessons, both technical and soft skills.

a. What Went Well

Because of the technical rigor of the project, we learned many new and valuable skills, notably in developing a PCB, PID controller, mechanical design, computer vision, g-code generation, routing algorithms, websockets, FreeRTOS tasks and scheduling, and web interface. From these skills, we also learned interactive testing, integration, and redesign.

b. What Went Poorly

We believe that delayed integration, prioritizing tasks, and overall timeline planning went poorly. We initially prioritized robot design and development, and most of our initial meetings were co-designing the robot. However, as the time went on, each team member

specialized in their own subproject, resulting in each of us being highly knowledgeable about our own progress, and less of each other. This led to slower integration, miscommunication of design goals, and less team collaboration.
Since this project is built such that testing our software is reliant on an operational robot– meaning that the robot must be mechanically and electrically operable, this set us back quite a bit as there were several hurdles in their development. We believe that better task allocation to these high priority tasks would have been optimal.

c. Memo

"You will learn a lot, but choose a more feasible project. If you really want to do this, prioritize having a functional robot early on for early software integration."

V. Contributions of each member of team

Team member	Contribution	Effort
Mango (Robot)	Provided ***limited*** artistic services (he tried his best).	1.0%
Colin Riker:	Colin handled designing and manufacturing of our custom robot. Creating CAD for every mechanical system such as drive system, drawing gantry, tool head and tool swapping systems. Finally taking each design and printing them using FDM printing. In total completing nine different iterations of the DrawBots design. He helped explore many of the different tire solutions, from rubber bands, to rc racing tires. He also helped create some diagrams/images and copy used for the final report.	16.5%
Vivien Lin:	Vivien worked on image processing to extract coordinates and detect colors for G-code generation, with separate commands for the stepper motor that controls the gantry and gear motor for driving. She optimized the robot path by filtering out unnecessary points and contours and sorting the contours. She worked on webcam detection of Aruco tags for the absolute position of the robot. She also helped with the software integration for the full-pipeline in FreeRTOS.	16.5%
Shruti Shah:	Shruti worked on the web interface, handling the user interface and the robot communication system. She handled setting up the message requests, and constructing a pseudo-API that the robot can use to request its webcam location, battery percentage, and G-Code files. She also assisted in assembling and soldering the PCBs. She also helped with overall integration tasks in FreeRTOS. Additionally, she helped design the poster and its graphics.	16.5%
Aryan Thomare:	Aryan worked on the robot's navigation and control systems, including the motor PID loops, encoder integration, and odometry sensor fusion for navigation. He implemented the servo and marker control logic used in the final demo and helped integrate the webcam data into the navigation pipeline. He developed the web-based debugging interface for manual control, telemetry, and testing. Aryan also contributed to software integration in FreeRTOS and collaborated with the team on robot assembly and mechanical modifications throughout the project.	16.5%
Cindy Yang:	Cindy led the design of the custom PCB, handling both schematic capture and board layout. She assembled the PCBs, performed testing and debugging, and managed wiring. She contributed to PCB integration and power management. She provided ongoing hands-on	16.5%

	support to bring mechanical and electrical subsystems together. She worked on calibrating the webcam and lightly assisted in debugging the FreeRTOS pipeline. She also worked on the final report.	
Prisca Yim:	Prisca worked on the drawing mechanism, mainly the software side and integrating it with the robot's main system. She created the gantry.h library for the gantry and tool changing mechanisms that interfaced with the robot's main code. She helped with the prototype CAD design of the marker holder and gantry. She also wrote the milestone 1 and 2 reports and helped with the demo poster.	16.5%

VI. Cost of Manufacture

Robot Quote

Part	Quantity	Price	Tax	Shipping	Total
Neodymium Bar Magnets	1.00	$15.99	$0.96	$0.00	$16.95
SparkFun Optical Tracking Odometry Sensor - PAA5160E1	1.00	$82.50	$4.95	$0.00	$87.45
3D Printer Filament, Baby Blue and Gray	2.00	$4.99	$0.30	$0.00	$10.28
600 pcs M3 Screw Assortment Kit	1.00	$7.99	$0.48	$0.00	$8.47
Gear Motor 20 RPM Low Noise for 12mm Diameter Planetary Motor	2.00	$38.74	$2.32	$0.00	$41.06
190mNm High Torque Stepper Motor 1.8 Deg Standard Step Angle	1.00	$11.69	$1.40	$0.00	$13.09
Pulley Rail Ball Bearings	1.00	$6.79	$0.36	$0.00	$7.15
3 mm Diameter Brass rods	1.00	$4.99	$0.35	$0.00	$5.34
12V 3Ah battery	1.00	$21.99	$0.13	$0.00	$23.31
SG90 Microservo	1.00	$1.50	$0.09	$0.00	$1.59
HI422 Servo	1.00	$20.99	$1.26	$0.00	$22.25
MG90D Servo	1.00	$7.00	$0.42	$0.00	$7.42
Logitech 720p Webcam	1.00	$14.00	$0.84	$0.00	$14.84
Magnetic Tape	1.00	$4.19	$0.25	$0.00	$4.44
Total Cost					$259.20

JLCPCB Quote Per PCB:

Section	Price
JLCPCB Available Components	$33.74
JLCPCB Unavailable Components	$6.02
Assembly	$26.17
Print	$0.40
Total Board Cost	$66.33

The total cost of manufacturing a DrawBot, came to **$325.53**. This is a reasonable price for a fairly populated design that incorporates higher-value parts such as the ESP32 microcontroller, charging ICs, a USB-to-UART controller, and motor drivers. While this board is more expensive than a standard ESP32 development board, it has been purpose-built for our system, offering greater reliability and integration tailored to our specific requirements. Additionally, the robot itself is highly specialized, and has several features with specific sensors such as the odometry, webcam, and gear motors that drive the cost up. We can also expect this to be a one time cost, and that the robot should not require consistent upkeep.

VII. Parts and Budget

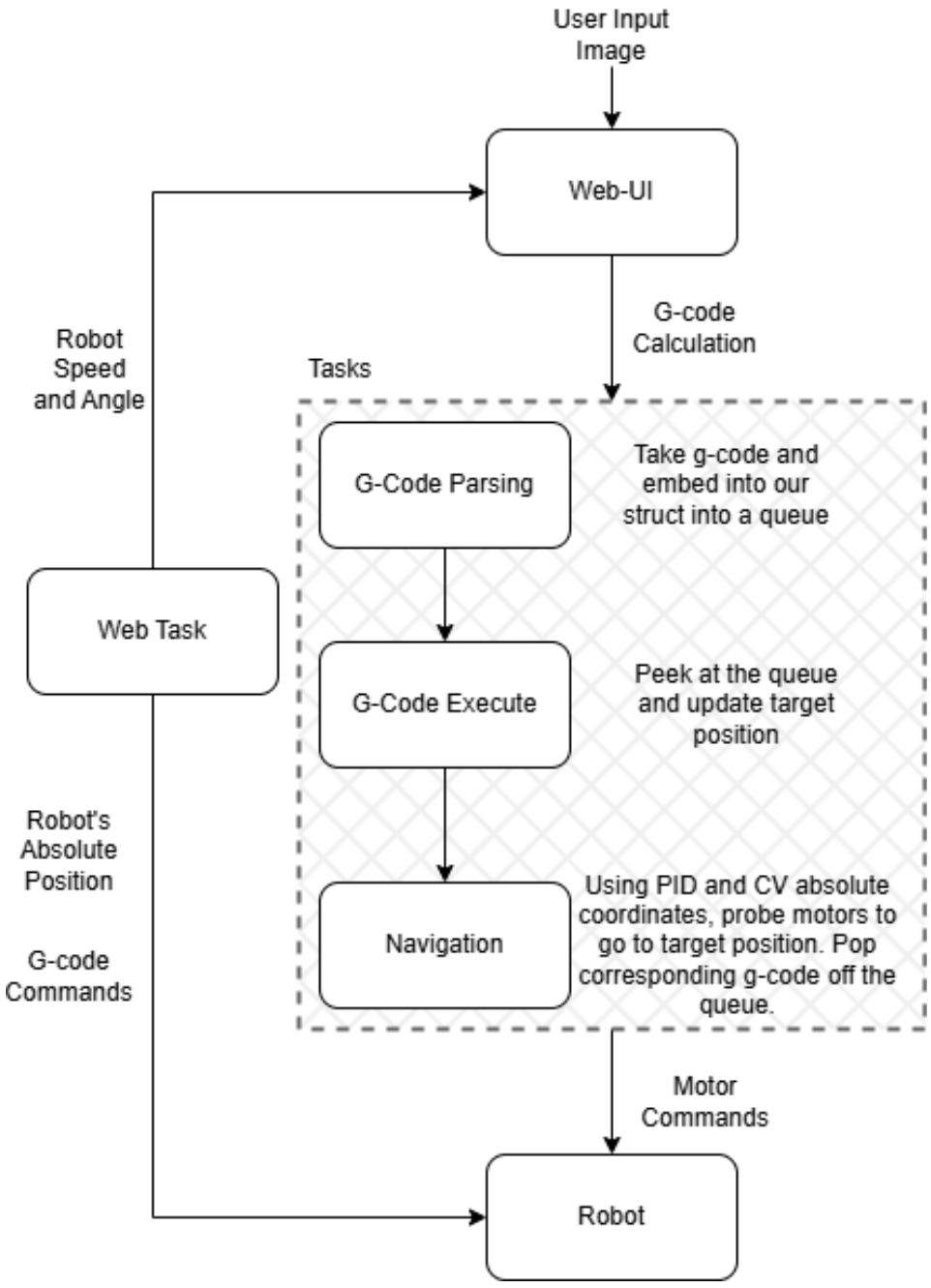
Part	Quantity	Price	Tax	Shipping	Total
Neodymium Bar Magnets	1.00	$15.99	$0.96	$0.00	$16.95
Optical Tracking sensor with 6-axis IMU - 1x1in	1.00	$82.50	$4.95	$0.00	$87.45
3 ESP32 Boards with a Type C Cable Set	1.00	$19.82	$1.19	$0.00	$21.01
10-Piece Set of DRV8825 Stepper Motor Driver Modules	1.00	$20.29	$1.22	$0.00	$21.51
Pack of four MOTOU Nema17 Stepper Motors	1.00	$29.99	$1.80	$0.00	$31.79
3D Printer Filament 1.75 mm, 1kg Spool, Baby Blue and Gray	2.00	$19.99	$2.40	$0.00	$42.38
Pack of 12 EXPO Dry Erase Markers, Assorted Colors	1.00	$10.49	$0.63	$0.00	$11.12

5-Spoke RC 12 mm Hex Hub Wheel Rims & Rubber Tires (Black)	1.00	$11.39	$0.68	$0.00	$12.07
600 pcs M3 Screw Assortment Kit	1.00	$7.99	$0.48	$0.00	$8.47
Self Locking Motor 20 RPM Low Noise 12mm Diameter Planetary Motor	2.00	$38.74	$2.32	$0.00	$41.06
190mNm High Torque 1.8 Deg Standard Step Angle 1A/3.7V 22.5mm	2.00	$11.69	$1.40	$0.00	$24.78
PCB Parts	1.00	$205.31	$12.32	$6.99	$224.62
Alliance Rubber 54625 Advantage Rubber Bands Size #62, 1 lb Box Contains Approx. 450 Bands (2 1/2" x 1/4", Blue)	1.00	$8.05	$0.48	$0.00	$8.53
Pulley Rail Ball Bearings 3mmx10mmx3mm Carbon Steel Miniature Bearings 10pcs, Nine Pieces 3 mm Diameter Brass rods	1.00	$11.78	$0.71	$0.00	$12.49
Transistors, Resistors, Capacitors	1.00	$7.44	$0.98	$6.99	$15.41
CN3303 10 pieces	10.00	$7.85	$0.47	$40.00	$48.32
PCB (Stencil & 5 PCBS)	1.00	$10.20	$2.69	$32.64	$45.53
JST Connectors	1.00	$16.19	$2.44	$6.99	$25.62
12V 3Ah battery	1.00	$21.99	$0.13	$0	$23.31
Magnetic Tape	1.00	$4.19	$0.25	$2.99	$7.39
PCB (Stencil & 5 PCBS)	1.00	$10.20	$10.07	$28.54	$48.81
Total Cost					$778.62

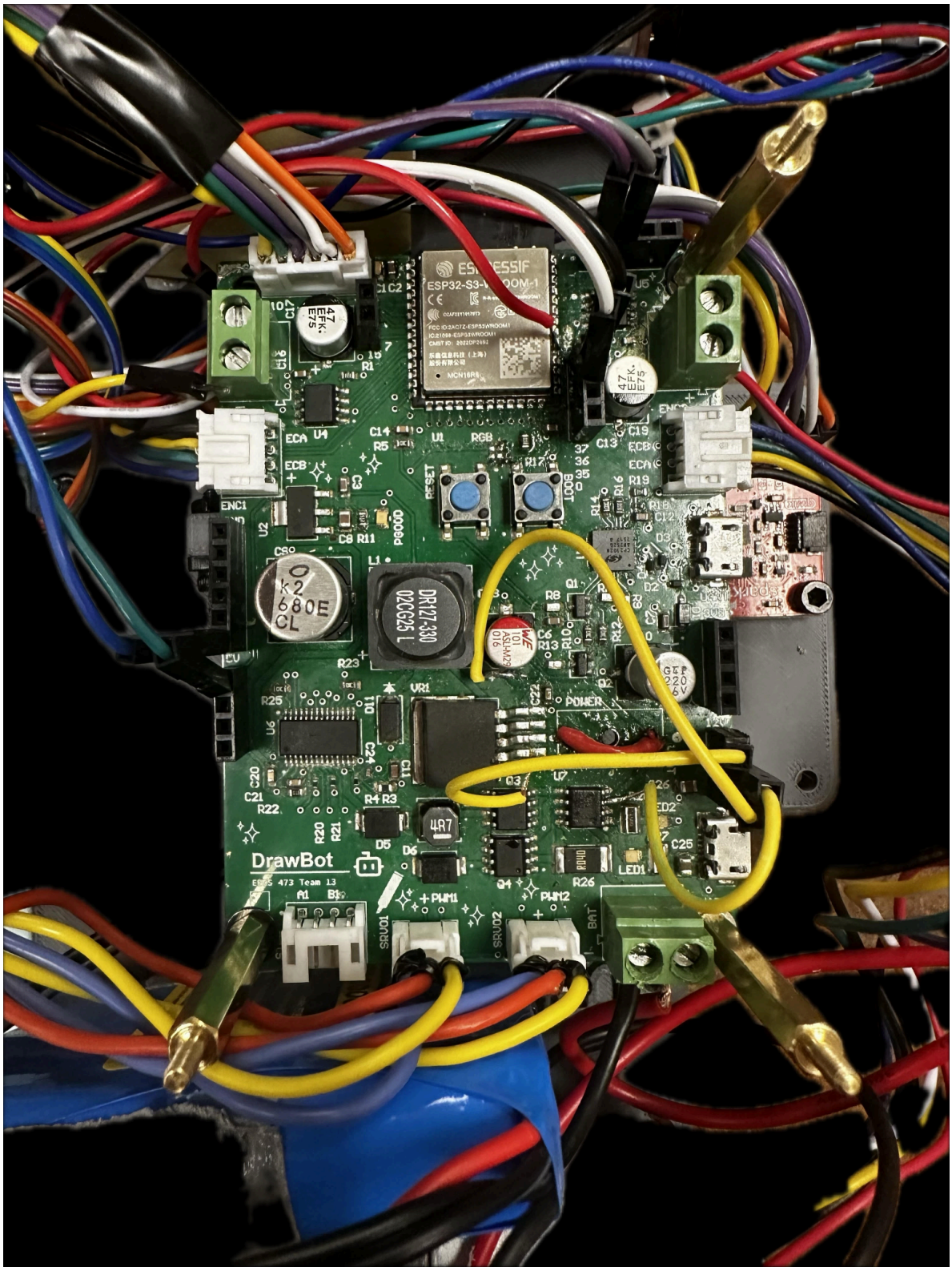
Total Budget: $1500
Remaining Budget: $721.38

VIII. References and Citations

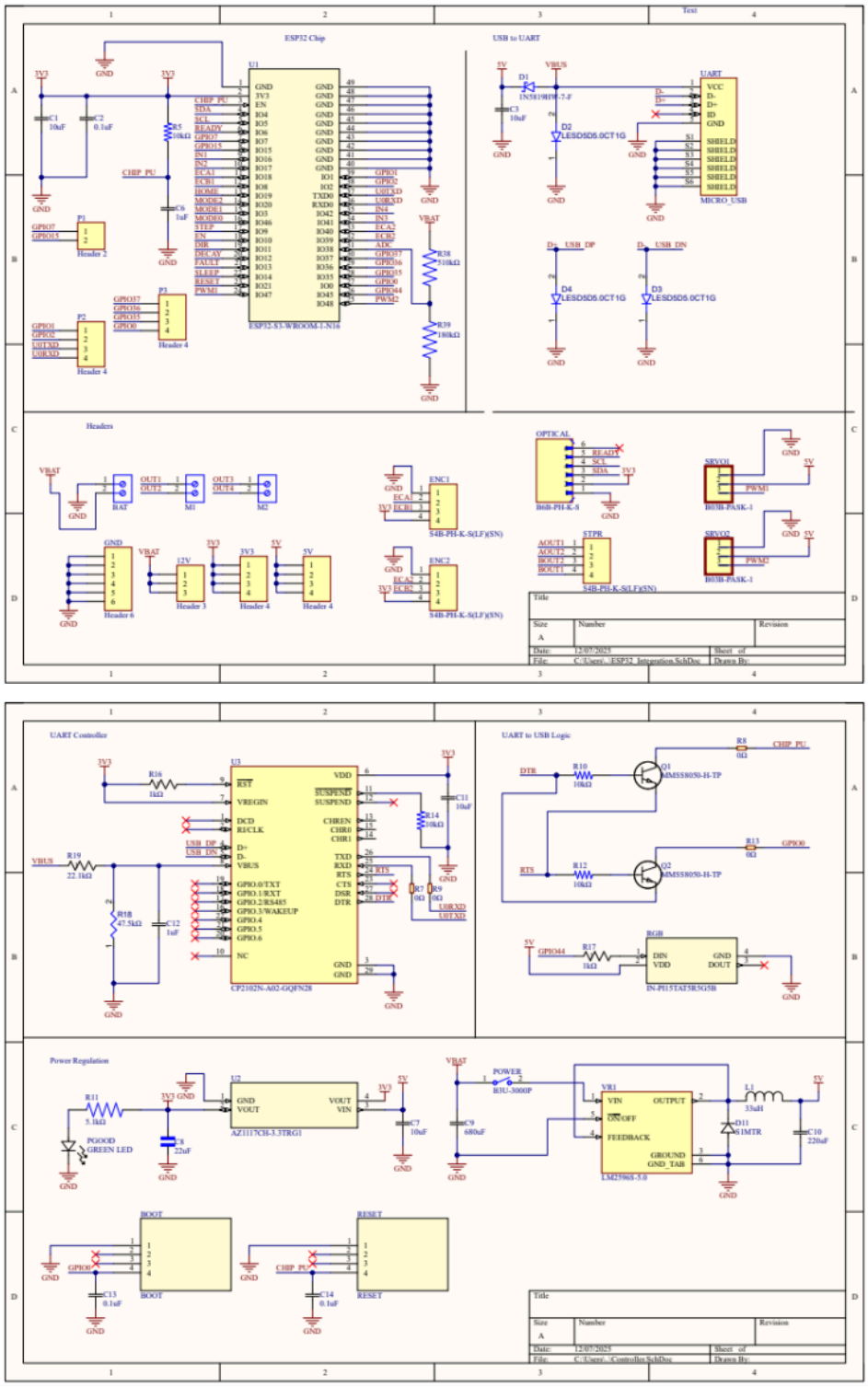
Full Code:
https://docs.google.com/document/d/1n2CT1IxTI4pEKCZMNLRU5KBHEzsIVs3C1V_g6McKcDA/edit?tab=t.0

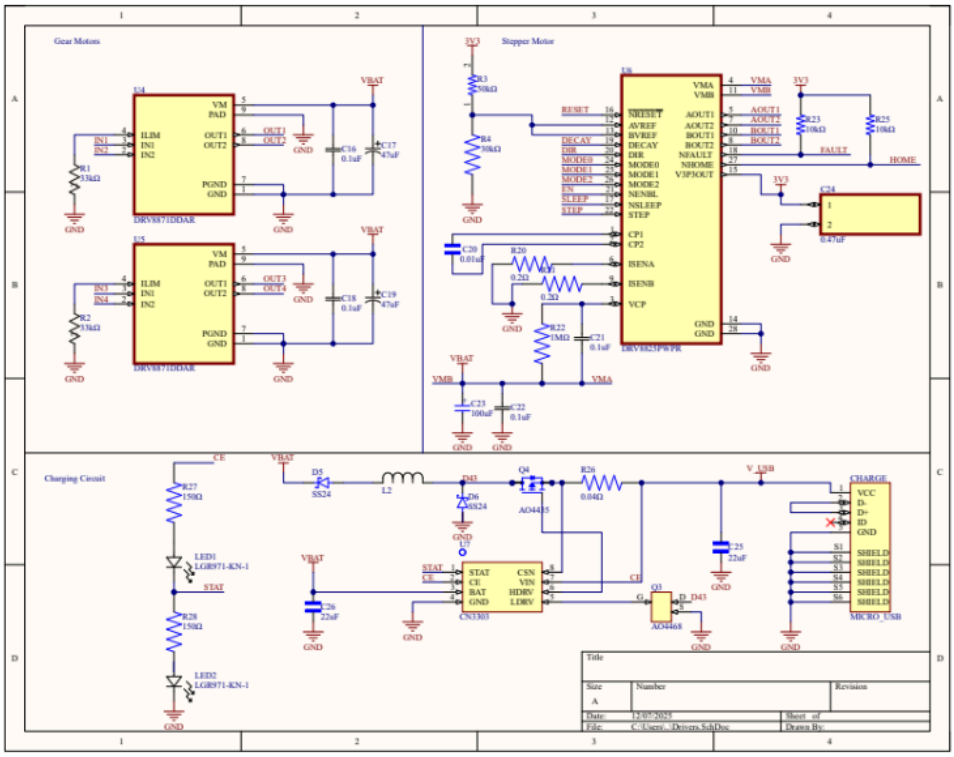


(fig. 13 High Level Software Pipeline)


DrawBot
EECS 473 Team 13
ESP32-S3-WROOM-1
DR127-330
02CG25 L
RESET
BOOT
POWER
PWM1
PWM2
SRV01
SRV02
BAT
ENC1
ECA
ECB
RGB
LED1

(fig. 14 final PCB design)





(fig. 15, 16, & 17 schematics for PCB)

PCB Design: Much of the schematic design was based on the ESP32 S3 WROOM 1 Dev Board. Specifically the bootloader logic and USB to UART Controller was directly copied into our design. Additionally, we'd like to express our gratitude to the Blind Barnacles team for giving us an ESP32 Microcontroller!

Figma UI: The front end web interface was designed with Figma UI. This allowed us to create a detailed specification of how we wanted the frontend interface to look like, and what kind of user parameters we wanted to include.

Rob Tillart: Used his DRV8825 Arduino Library from Github for gantry.h code for modeling the stepper motor controls similarly.

John K. Bennet (madhephaestus): Used his ESP32Servo Library from Github for gantry.h code for implementing the marker gripping and lifting servo controls.

Optical Sensor: Used Sparkfun Qwiic OTOS Arduino Library to calibrate the optical sensor and receive angle of the robot.

G-code: Followed g-code format using this CNC Hanging Plotter example.

========== Hardware Interfaces ==========

main.ino (Overall Task Scheduling)

```
//===== Structs =====
struct Stepper {
  float currentX, currentY;
  float nextX, nextY;
  bool markerDown;
  int color;
};
Stepper stepper;

struct GcodeCommand {
  char command[8];
  float nextX, nextY;
  bool hasX, hasY;
  float angle;
  int color;
};

// ====== Class Instances ======
ESP32Encoder encLeft, encRight;
QwiicOTOS myOtos;
GANTRY gantry;
WifiClient wifi;

// ==== Semaphores & Mutexes ====
SemaphoreHandle_t ExecuteGearSemaphore;
SemaphoreHandle_t ExecuteStepperSemaphore;
SemaphoreHandle_t poseMutex;
SemaphoreHandle_t printMutex;

// ========== Queues ==========
QueueHandle_t queStepper;
QueueHandle_t queGear;

// ========== Tasks ==========
```

```
/**
 * @brief Parses incoming G-code commands from WebSocket Buffer.
 * Commands are stored in a queue (separated by robot's travel points
 * and drawing points).
 */
void TaskGcodeParse(void *pvParameters) {
}

/**
 * @brief Peeks at the top of stepper motor G-code queue and sets
 * the target position of the stepper motor. ExecuteStepperSemaphore
 * is used to determine if new positions should be set.
*/
void TaskStepperGcodeExec(void *pvParameters) {
}

/**
 * @brief Peeks at the top of the gear motor G-code queue and sets
 * the target position for robot navigation. If command is G0, the
 * marker is lifted up. If G1, the marker is pressed down. This task
 * waits for ExecuteGearSemaphore to execute, and semaphore will
 * be given when the robot reached the goal in the navigationTask.

 */
void TaskGearGcodeExec(void *pvParameters) {
}

/**
 * @brief Calculates the motor velocities, reads the heading angle
 * from the optical sensor, and the robot's x & y coordinate from
 * the camera.
 */
void encoderTask(void *pv) {
}

/**
 * @brief PID task that continuously adjusts left and right motor
 * speeds by comparing target velocities to measured velocities,
 * applying PID control with anti-windup,and sending the resulting
PWM
 * commands to the motors at a fixed update rate.
```

```
 */
void pidTask(void *pv) {
}

/**
 * @brief Configures the speed and direction of the stepper motor
 * given the current robot's position, target position, and traveling
 * velocity. Pops the stepper motor command off the queue when the
 * position of marker is reached.
 */
void stepperTask(void *pvParameters) {
}
/**
 * @brief Implements a finite state machine with the following
states:
   enum State {
        ROTATE_TO_GOAL = 0,
        DRIVE_TO_GOAL = 1,
        ROTATE_TO_FINAL = 2,
        GOAL_REACHED = 3
    };
 * Given a target point (x,y), the robot first orients itself to face
 * the target in ROTATE_TO_GOAL within an angle tolerance.
Afterwards,
 * the state transitions to DRIVE_TO_GOAL, where it monitors the
 * position and adjust the motor's speed accordingly. In
 * ROTATE_TO_FINAL, the robot rotates to its desired orientation. In
 * GOAL_REACHED, the g-code pops off the queue and signals ready to
 * execute the next command.
 */
void navigationTask(void *pv) {
}

/**
 * @brief Continuously cleans up disconnected websocket clients.
 */
void TaskWebServer(void *pvParameters) {
}

// ======== WebSocket Methods ========
/**
```

```
 * @brief Handles all incoming messages from WebSocket.
 * Key message types include:
 * "ROBOT_POS": the position of the robot from the camera.
 * "GCODE":     indicates the start and end of G-Code files.
 * "TELEMETRY": general robot info such as encoder status &
coordinate
 */
void handleWebSocketMessage(void *arg, uint8_t *data, size_t len) {
}

/**
 * @brief Sends all robot information through websocket to our custom
 * debugging interface. Information includes robot's current
position,
 * current navigation state, encoder data, heading angle, and current
 * target point.
 */
void sendTelemetry() {
}

// ======== Setup Function ========
void setup() {
  Serial.begin(115200);

  // Initialize Gantry
  gantry.init();

  // Connect to Wifi
  Serial.print("ESP IP: ");
  Serial.println(wifi.connectWiFi(ssid, password, incommon_ca));

  // Initialize Optical Sensor
  Wire.begin(CUSTOM_SDA, CUSTOM_SCL);
  myOtos.begin();
  myOtos.calibrateImu();
  myOtos.resetTracking();
  myOtos.setAngularScalar(0.9985437903);

  // Create synchronization objects
  poseMutex = xSemaphoreCreateMutex();
  printMutex = xSemaphoreCreateMutex();
```

```
  ExecuteGearSemaphore = xSemaphoreCreateBinary();
  ExecuteStepperSemaphore = xSemaphoreCreateBinary();

  // Initialize G-Code Queues
  queStepper = xQueueCreate(50, sizeof(GcodeCommand));
  queGear = xQueueCreate(50, sizeof(GcodeCommand));

  // Initialize Gear Motors
  setupMotors();

  // Initialize Encoders
  ESP32Encoder::useInternalWeakPullResistors = puType::up;
  encLeft.attachHalfQuad(ENC_L_A, ENC_L_B);
  encRight.attachHalfQuad(ENC_R_A, ENC_R_B);
  encLeft.clearCount();
  encRight.clearCount();

  // Create Tasks
  xTaskCreatePinnedToCore(encoderTask, "Encoders", 4096, NULL, 3, NULL, 1);
  xTaskCreatePinnedToCore(pidTask, "PID", 4096, NULL, 2, NULL, 1);
  xTaskCreatePinnedToCore(navigationTask, "Nav", 4096, NULL, 2, NULL, 1);
  xTaskCreatePinnedToCore(TaskGcodeParse, "GcodeParse", 4096, NULL,2,NULL, 1);
  xTaskCreatePinnedToCore(TaskGcodeExec, "GcodeExec", 4096, NULL, 2, NULL, 1);
  xTaskCreatePinnedToCore(TaskWebServer, "WebServer", 8192, NULL, 3, NULL, 1);
}
```

gantry.h

```
class GANTRY {
  private:
    Servo z_axis;       // servo for z axis movement
    Servo gripper;      // servo for grippers
    Servo tool_change;  // servo for tool change rack
    DRV8825 gantry;     // STEPPER MOTOR FOR GANTRY

    /**
     * @brief Puts marker back into tool changing rack. Specify
     * position num to put it back to.
     */
    void putMarkerBack(int numPos);

    /**
```

```
     * @brief Takes marker out of tool changing rack. Specify
position
     * to take it out of.
     */
    void takeMarkerFrom(int numPos);


    /**
     * @brief Move marker to the right
     * @param distance Travelling distance for marker (mm)
     * @param speed    Speed at which the marker should travel (mm/s)
     */
    void moveRight(float distance, float speed);

    /**
     * @brief Move marker to the left
     * @param distance Travelling distance for marker
     * @param speed    Speed at which the marker should travel
     */
    void moveLeft(float distance, float speed);

    /**
     * @brief Grab marker (assuming positions are already aligned)
     */
    void grab();

    /**
     * @brief Release marker (assuming positions are already aligned)
     */
    void release();

public:
    float position = 0.0; // position of gantry belt

    // *** FUNCTIONS! ***
    // NOTE: with the current stepper motor resolution (1.8˚) and the
       radius of the pulley wheel (6mm),
    // each step is equivalent to moving 0.188 mm
    // this can change with microstepping

    /**
```

```
     * @brief initialize servos and driver
     */
    void init();
    /**
     * @brief Move gripper fully to the top of the gantry.
     * @param orientation 0: robot is facing to right.
     *                    1: robot is facing to left.
     */
    void resetPos(int orientation);

    /**
     * @brief gantry moves to an "absolute" position
     */
    void move(float pos, float speed);


    // ===== MARKER COMMANDS =====
    void liftMarker();
    void lowerMarker();
    /**
     * @brief Gantry travels to colorPos1 and releases current
marker.
     * Then it travels to colorPos2 and grabs the new marker.
     */
    void switchMarker(int colorPos1, int colorPos2);
};
```

pid.h

```
/**
 * @brief Initialize motor pins and pwm channels
 */
void setupMotors();
/**
 * @brief Remap PID output to skip deadzone
 */
int remapPID(double pidOutput, bool rotatingInPlace);
/**
 * @brief Write PWM signal to motors.
 * @param left True for left motor & False for right
 * @param pwmVal desired pwm value for motor
 */
```

```
void motorWrite(bool left, int pwmVal);
/**
 * @brief Write 0 PWM for both motors
 */
void stopMotors();
```

wifiClient.h

```
class WifiClient {
private:
    const char* ntpServer = "pool.ntp.org";
    const char* timezoneEST = "EST5EDT,M3.2.0/2,M11.1.0";
    unsigned long previousMillisWiFi = 0;
    char timeStringBuff[50];
public:
    WifiClient();
    /**
     * @brief connect the robot to a wifi server
     * @param username WiFi username
     * @param password WiFi password
     * @param certificate If connecting to enterprise WiFi, include
     * certificate
     */
    uint32_t connectWiFi(const char* username, const char* password,
const char* certificate = "");
};
====================== End of Hardware Interfaces
====================
========================= Software Interfaces ======================
```

app.py (flask server)

```
"""
Process the image - replace this with your actual processing logic.
Args:    image_path (str): Path to the uploaded image file
         name (str): Name of the snapshot
         width_mm(float): User-defined width of drawing space
         height_mm(float): User-defined height of drawing space
Returns:
        dict: Processing result from G-Code convertor
"""
def process_image(image_path, name, image_type, width_mm, height_mm):
@app.route('/process/<image_type>/<width_mm>/<height_mm>',
methods=['POST'])
```

```
"""Handle image processing requests"""
def process(image_type, width_mm, height_mm):
@app.route('/video_feed')
"""
Video streaming route that streams processed video frames with ArUco
detection.
Returns MJPEG stream that can be used in an HTML img tag or video
element.
"""
def video_feed():
@app.route('/robot_pos')
"""
Return current robot position {'x': x, 'y': y} (Directly sent to
robot)
"""
def get_robo_pos():
    return get_robo_loc()
@app.route('/configure_aruco', methods=['POST'])
"""
Configure ArUco whiteboard size in millimeters.
Expects JSON: {"width_mm": <number>, "height_mm": <number>}
"""
def configure_aruco():
```

RobotControl.tsx (brief snippets of robot communication protocol)

```
// Declare websocket to communicate with robot
const wsRef = useRef<WebSocket | null>(null);
// Initialize robot on user-provided IP address
const ws = new WebSocket(`ws://${robotIpAddress.trim()}/ws`);
…
// Send Robot Location from Camera
useEffect(() => {
    const interval = setInterval(() => {
      const fetchAndSend = async () => {
        try {
          const response = await
          fetch("http://localhost:500/robot_pos"); // API Call

          if (response.ok) {
            const { x, y } = await response.json();
            wsRef.current?.send(`RobotPos: x=${x} y=${y}`);//Send to
```

```
Robot
          }};
      fetchAndSend();
    }, 250); // Send every 0.25 seconds
…
// Fetch processed G-Code from the backend
const gearGCodeText = await response_gear.text();
const stepperGCodeText = await response_stepper.text();
// Helper function to split text into chunks of N lines
const splitIntoChunks = (text: string, linesPerChunk: number): {};
// Helper function to send chunks with delay
const sendChunks = async (chunks: string[], chunkType: string){};
// Split into chunks of 40 lines
const CHUNK_SIZE = 80;
const gearChunks = splitIntoChunks(gearGCodeText, CHUNK_SIZE);
const stepperChunks = splitIntoChunks(stepperGCodeText, CHUNK_SIZE);
await sendChunks(gearChunks, "gear");
await sendChunks(stepperChunks, "stepper");
```

detect_aruco.py

```
# [Upper Left ID, Bottom Left ID, Top Right ID, Bottom Right ID]
BOARD_IDS = [5,1,6,3]
ROBOT_ID = 7 # Robot Tag ID
robot_position = [None, None] # Global RobotPos from Camera

class Aruco:
    def __init__(self):
     # dimensions of board determined by distance between the tags
       self.width_mm = 0
       self.height_mm = 0
    """
    Separate detected ArUco markers into the robot tag and the 4
board
    tags.

    Parameters:
        frame: Current camera frame.
        corners: Detected marker corner coordinates.
        ids: Marker IDs.
        rvecs: Rotation vectors for detected markers.
```

```
        tvecs: Translation vectors for detected markers.

    Returns:
        robot_tag: Dict with robot tag data, or None.
        board_tags: List of dicts for detected board corner tags.
    """
    def detect_tags(frame, corners, ids, rvecs, tvecs):

    """
    Read a frame, detect ArUco tags, and compute the robot's global
    position.

    Parameters:
        aruco_dict: ArUco dictionary for detection.
        parameters: Detection parameters.
        cap: OpenCV video capture object.

    Returns:
        robot_pos: (x, y) in mm, or None if not all tags are found.
        frame: Processed frame with marker overlays.
    """
    def get_pos(aruco_dict, parameters, cap):
```

gcode.py

```
class GcodeConverter:
     # parameters are user-configurable variables on the webserver
     def __init__(self, width_mm, height_mm, image_type, color_map,
     top_left_x=0, top_left_y=0):

          self.RowHeight = 100 # gantry length in mm

          # set image dimensions for image processing
           self.WIDTH_PIXELS = 2000
           self.HEIGHT_PIXELS = 2000
```

```
            # true if input image is digital, false if photo
            self.isDigital = (image_type.lower() == 'digital')

           # drawing dimensions on the board
            self.WIDTH_MM =  WIDTH_MM
            self.HEIGHT_MM = HEIGHT_MM

            # default color for undefined colors
            self.DEFAULT_COLOR = 'black'

           # assign marker positions on the robot
            self.draw_positions = {
                 color1: 1,
                 color2: 2,
                 color3: 3
             }

"""
- Process image into list of contours.
- Filters out contours and points.
- Split the contours for drawing to be completed row by row
- Sort the paths for optimal gantry drawing

Input:
     File path of image
Returns:
     3 output files:
        -  file1 for stepper motor commands
        -  file2 for gear motor commands
        -  file3 for plotting
"""
     def image_to_gcode(self, image_path, output_file1,
output_file2,
     output_file3):
```

color.py

```
"""
Input: sized original image, list of 8 unique colors, default color
set for undetected colors
```

```
Output: list of contours with assigned colors for each contour

Creates color masks for 8 unique colors, overlaps them with the
contour mask, and assigns the color that has the most overlapping
pixels to the contour.
"""
def assignColors(org_img, contours, drawing_colors, default_color):
```

visual.py

```
"""
Input: file direction for g-code file, width and height of image
dimensions
Output: saves the plot in output file

Plots all the paths with designated colors and returns the visual
file
"""
def visualize_gcode(gcode_file_path, output_folder, output_file,
WIDTH, HEIGHT):
```

ocr.py (text recognition & processing)

```
ocr = PaddleOCR(
    use_doc_orientation_classify=False,
    use_doc_unwarping=False,
    use_textline_orientation=False)
 """
Detect text in an image using OCR and return the recognized words.
Parameters:
    image_path: Path to the input image.
    INPUT_WIDTH: Width to resize the image for OCR (pixels).
    INPUT_HEIGHT: Height to resize the image for OCR (pixels).
    confidence_threshold: Minimum OCR confidence to keep a word.
Returns:
    words: List of detected words w/ positions and confidence scores.
"""
def detect_text(image_path, INPUT_WIDTH=1500, INPUT_HEIGHT=1500,
confidence_threshold=0.8)
```

```
"""
Write detected OCR text back onto a copy of the image and save the
result.
Parameters:
    detected_words: List of words and their coordinates.
    image_path: Path to the original image.
Returns:
    output_path: File path to the saved annotated image.
"""
def transpose_print_text_to_image(detected_words, image_path)

====================== End of Software Interfaces
====================
```