A1 Group 8 — The Deskinator

Final Design Report

Team: Giacomo Cappelletto, Ava Johnson, Birinder Bachhal, Ceren Atalar
Course: EK210
Date: December 07, 2025

Contents

1. Executive Summary

This project focuses on developing a fully autonomous desktop-cleaning robot ("Deskinator") designed to maintain the EK131-210 classroom workspaces between student uses. The goal is to provide a reliable, debris-removing solution that reduces the need for manual cleaning, ensures a sanitary environment, and increases classroom efficiency. The device is engineered to operate safely on a 4 ft × 4 ft desk surface, navigate without falling off, and remove debris, represented by grains of rice in testing, using a custom-built vacuum system.

To achieve this goal, the Deskinator integrates two major subsystems: a motion platform capable of detecting desk boundaries using onboard proximity sensors, and a vacuum mechanism designed specifically for this project. These are orchestrated by software running on a Raspberry Pi 4B, which performs real-time localization via an Extended Kalman Filter, boundary discovery through a bump-and-turn state machine, and boustrophedon coverage planning, all visualizable remotely over SSH/VNC. The robot is powered entirely by an onboard battery and is triggered using a touchless activation method, such as a hand wave. Upon completing its cleaning cycle, it provides a clear end-of-task signal through an led.

Testing evaluated the robot's ability to safely move about the desk surface, successfully detect edges, collect debris from the safe operation area into a removable sealed container, operate wirelessly for more than five minutes, and complete its cleaning task within the required two minutes. Results demonstrated that the robot met the performance criteria: it navigated without falling, reliably responded to the touchless gesture start signal, collected an average of 99.40% of rice grains with the custom vacuum into the attached container, and indicated initiation and completion through the LED.

A high-level glass-box diagram summarizing the system's sensing, computation, and actuation is provided in Appendix Section 7.1.

2. Introduction

2.1. Problem Statement

The goal is to design a device that can clean and maintain shared EK131-210 desks in between uses, to ensure that students have a debris-free and sanitary workspace, without requiring a staff to manually wipe down each desk during classes.

2.2. Key Objectives and Metrics

Objective	Metric / Constraint
Reliable	Complete 99% of cleaning cycles without malfunction or human intervention
Accuracy	Remove 95% of rice from the 4 ft × 4 ft desk area

Objective	**Metric / Constraint**
Efficiency	Clean the full desk area in under 2 minutes
Automated	Operate entirely on its own after receiving the start signal
Navigation	Detect and avoid desk edges; never fall off
Durability	Withstand day-to-day use
Ease of use	A new user can start cleaning with minimal instruction
Touchless	Start and stop without physical contact
Storage capacity	Hold debris from three full cleaning cycles before emptying
Precision	Maintain cleaning paths within a 1.5 cm margin
Affordability	Material cost under $200
Portability	Weight under 750g
Ease of cleaning	Dustbin and brushes are removable and washable
Compact	Footprint no larger than 12 in × 12 in
Completion recognition	Signal task completion via light or sound

The relative importance of these objectives was compared using a pairwise comparison chart shown in Appendix Section 7.2.

3. Innovative Elements

Concept generation and selection for the Deskinator is documented in the morph chart in Appendix Section 7.3, which captures alternative options for sensing, motion, suction, and user interaction.

3.1. Software

3.1.1. Extended Kalman Filter (EKF) for Localization

The EKF in slam/ekf.py (see Appendix Section 7.9) is the foundation of the robot's self-awareness on the table. It maintains a probabilistic estimate of the robot's pose (x, y, θ) by fusing two complementary sensor streams, which are the wheel odometry from the stepper motors and yaw rate from the MPU-6050 gyroscope. The prediction step uses differential drive kinematics to propagate position based on wheel displacements, while the gyro update corrects angular drift that accumulates from wheel slip. The filter also implements tactile localization via update_line_constraint(). This happens when the robot detects a table edge during the coverage phase through the APDS9960 front facing sensors, it therefore knows it must be on the boundary of the table and snaps its position to that known wall. This closed-loop correction keeps cumulative drift bounded even during long cleaning sessions by enforcing known constraints (eg. table edges are at 90°) to the pose graph estimation model.

3.1.2. Wall Follower with Bump-and-Turn Logic

The WallFollower in control/wall_follower.py implements a boundary discovery algorithm using a simple but effective bump-and-turn state machine. The robot drives forward until the APDS-9960 proximity sensors detect an edge (raw value drops below threshold in config.py, meaning no surface reflection), then executes a choreographed sequence: back up a fixed distance, rotate a few degrees opposite to the detected side, and continue. By counting corners and tracking distance traveled, the algorithm can detect when it has completed a full lap around the table (4 corners + returning near the start pose). This approach is very tolerant of sensor noise, because rather than trying to maintain precise wall-following distance, it embraces contact with the boundary as the primary source of information. The corner poses collected during this phase are fed to RectangleFit to estimate the table's dimensions using an implementation of RANSAC, and a loop closure is added to the pose graph when the lap completes, globally correcting accumulated drift, as seen in Figure 1.

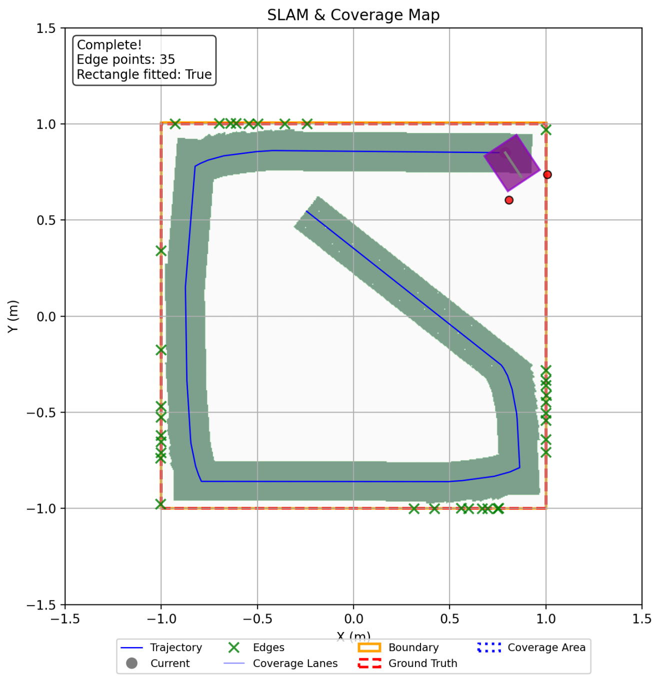

Figure 1: Wall follower bump-and-turn trace used to fit the desk boundary.

3.1.3. Boustrophedon Coverage Planning with Visualization

Once the boundary is known, the CoveragePlanner in planning/coverage.py (see Appendix Section 7.9) generates a boustrophedon (lawn-mower) pattern of parallel lanes that guarantees complete coverage. It computes lane spacing based on the vacuum width minus a configurable overlap (in config.py), ensuring no gaps between passes and a fully swept table. The lanes alternate direction ("back-and-forth") to minimize unnecessary turns. Meanwhile, the SweptMap tracks which cells have been visited, and the Visualizer in utils/viz.py renders both the SLAM map (showing trajectory, detected edges, loop closures, and tactile corrections) and a green heatmap of coverage in real-time using matplotlib (see Figure 2). This immediate feedback lets users verify the robot is working correctly and see exactly which areas have been cleaned.

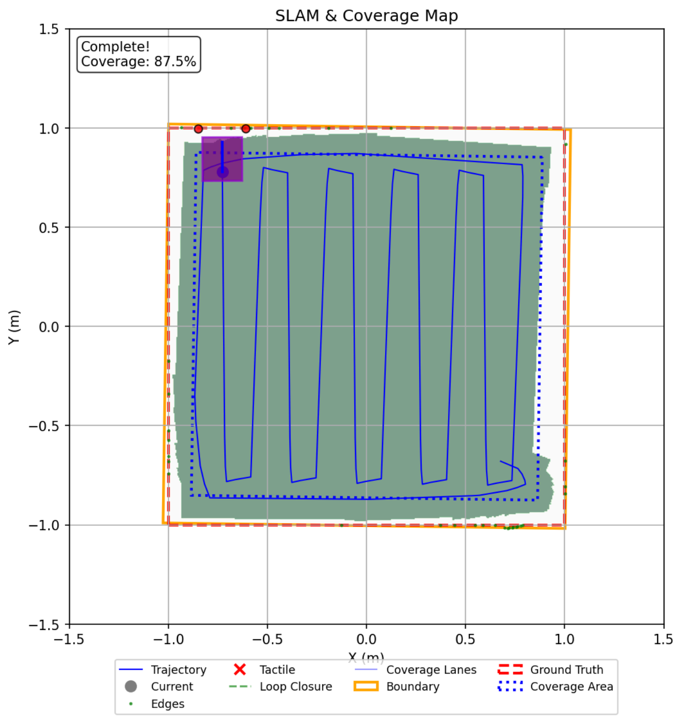

Figure 2: Boundary points collected during discovery feed lane generation.

3.2. Hardware

3.2.1. Raspberry Pi 4B

Instead of an Arduino-based microcontroller, the Deskinator uses a Raspberry Pi 4B for three reasons. First, its Linux environment enables SSH/VNC connectivity, streaming the real-time pose graph and coverage heatmap to a remote computer without onboard display hardware. Second, the Pi supports multiple software I2C buses, which is critical for the three APDS-9960 sensors that share the same fixed address (0x39), avoiding external multiplexers. Third, networked Python with asyncio allows a single workstation to monitor and control multiple robots simultaneously, enabling a future implementation of fleet-scale operation across several desks.

3.2.2. Modular Battery Pack

Because the robot is designed for repeated, extended cleaning cycles on frequently used tables, we implemented a fully external, hot-swappable battery system. The two Li-ion battery packs mounted on the top of the housing provide approximately 3 hours of continuous operation (see Appendix Section 7.8). This modular design allows one pack to power the robot while the other is charging, enabling uninterrupted use and minimizing downtime during long cleaning sessions.

3.2.3. IMU — MPU6050

To ensure accurate dead reckoning and therefore enhance precision in the loop closures of the SLAM pose graph estimation algorithm, the use of a 6-axis gyro and accelerometer was employed. The fusion of the MPU6050′s data (bearing, angular acceleration, linear acceleration) and the motor's calculated odometry ensures that errors linked to drift and slip are reduced to a minimum, and loop closures yield tighter data in the algorithm part.

A full bill of materials listing all electronic and mechanical components is provided in Appendix Section 7.4, while the detailed wiring and power distribution are shown in the electronics sketch in Appendix Section 7.5. Mechanical layout and vacuum geometry are captured in the CAD drawings in Appendix Section 7.7.

3.3. Final Product Integration

The final product is a fully autonomous desktop-cleaning robot that combines advanced software, precision hardware, and coordinated sensing actuation to deliver a cleaning system. Its software integrates a bump-and-turn wall-following algorithm to map out boundaries, a boustrophedon coverage planner that generates parallel lanes without gaps, and an Extended Kalman Filter SLAM framework that fuses wheel odometry from two NEMA17 motors with data from the MPU6050 and edge constraints from the APDS9960 sensors. Together these ensure drift bounded localization throughout both mapping and cleaning phases. The hardware system includes a dual-battery pack for extended operation, high-torque stepper motors for consistent differential drive motion, and a custom vacuum chute built using the continuity principle

$$V_1 A_1 = V_2 A_2$$

to amplify suction velocity at the intake. The APDS9960 sensors enable both reliable edge detection and touchless activation, while the Raspberry Pi 4B handles multi-bus I2C communication, real-time visualization, and remote SSH access.

The code supporting these subsystems is organized into modular components that define the robot's behavior. Low-level modules control the motors and vacuum, while sensor drivers manage IMU data, APDS9960 proximity readings, and gesture detection across separate software-I2C buses. SLAM and mapping modules implement the EKF prediction-update cycles, boundary fitting, and edge-based corrections, while the planning layer handles wall following, lane generation, and sweep-map coverage tracking. Testing and simulation tools support repeated trials, generate SLAM and coverage visualizations, and verify system performance. Altogether, this integrated software manages actuator behavior, sensor processing, boundary estimation, coverage execution and simulation-based testing, resulting in a platform capable of safely navigating a desktop and removing debris.

The overall execution sequence is summarized in the code flow chart in Appendix Section 7.6.

4. Results

4.1. Coverage Testing

The robot's cleaning performance was evaluated through a comprehensive testing framework implemented in robot_testing.py, which executed 50 independent trials of the complete cleaning cycle of a $2\text{m} \times 2\text{m}$ desk, of which 2 quit unexpectedly, leaving 48 full completed trials. This converts to a 96% cleaning trial efficacy, which is close to the Key Objective of 99% eff. Each trial consisted of two phases: boundary discovery, where the robot employed wall-following behavior to map the table perimeter and fit a rectangular boundary model, and coverage execution, where the robot followed a systematic boustrophedon path pattern to clean the identified region. During each trial, eight key performance metrics were collected, including edge points detected, rectangle fitting error (measured in parts per million), coverage percentage for both the full and inset rectangles (safety for edges), boundary discovery time, coverage time, total cleaning time, and total distance traveled.

All trial data was exported to Excel spreadsheets containing both raw measurements and summary statistics (mean, standard deviation, minimum, and maximum values). The results were subsequently analyzed using visualize_results.py, which generated Gaussian distribution plots for each performance metric by fitting normal distributions to the empirical data. This visualization script produced individual histograms overlaid with fitted Gaussian curves showing the mean and standard deviation for each metric, along with a comparative time analysis plot displaying the distributions of boundary discovery, coverage, and total cleaning times side-by-side.

4.2. Key Outcomes

4.2.1. Coverage of inset (safe) rectangle

This is the most crucial metric to demonstrate the efficacy of the deskinator, as it measures the swept 'safe' area by the fan. This area is calculated by applying a margin equal to the distance between the sensor line and the start of the fan line, and then used to calculate the boustrophedon path. This ensures that the robot doesn't attempt to step outside of the table area it has identified. Therefore, this is effectively the only area we can safely sweep. Testing shows a range of values (% of area swept) between 96,95% and 100%, with a mean of 99.40% and standard deviation of 0.70%, indicating highly effective sweeping of the reachable area of the table, therefore meeting the main Key Objective of the project.

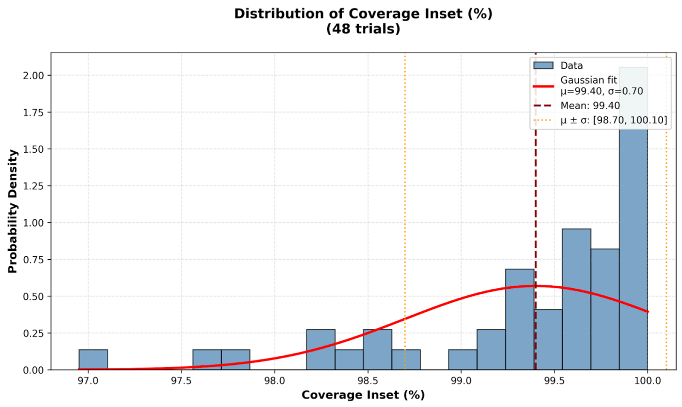

Figure 3: Distribution of swept inset coverage with Gaussian fit.

4.2.2. Time Metrics

Another crucial aspect of the successful and efficient implementation of the deskinator is its speed. One of the Key Objectives was for the full sweeping of the table to be completed in under 120 seconds. The conducted testing used conservative linear and angular velocity limits (see config.py) but still achieved a mean completion time of 137 seconds, a standard deviation of 3.57 seconds, and a maximum value of 146,25 seconds. Increasing the linear and angular velocity limits by just 21.875% would reduce maximum times to 120 seconds, as well as significantly decreasing the mean time. Testing with higher limits has been conducted and showed improved speed performance, but lower limits were chosen for repeated performance due to concerns linked to overheating of the A4988 motor drivers. The addition of a heatsink would solve these issues, effectively decreasing mean operating time.

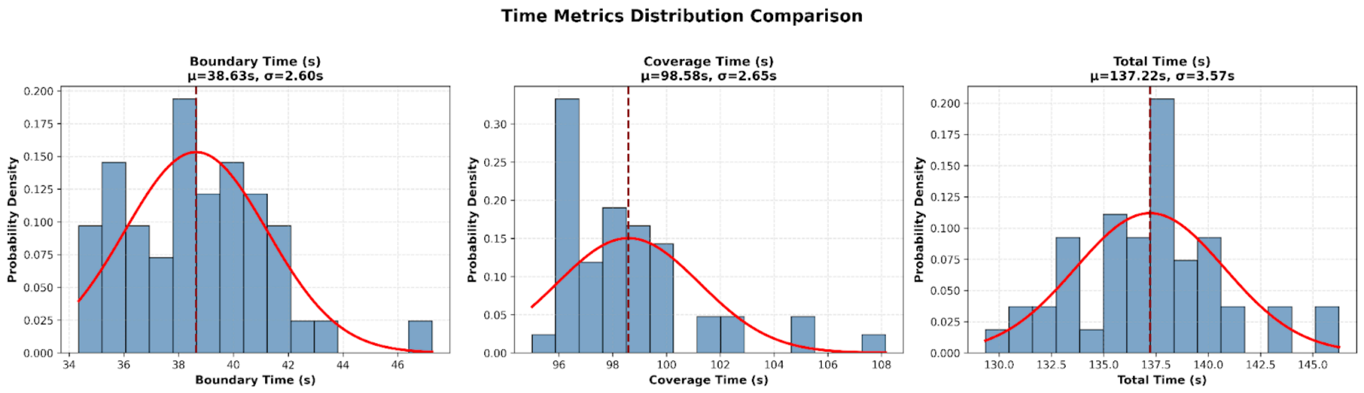

Figure 4: Boundary, coverage, and total time distributions across trials.

4.3. Additional Performance Indicators

Rectangular fit error and distance traveled remained tightly clustered, indicating consistent path tracking and coverage density.

image("images/distribution-distance.png", width: 50%) image("images/distribution-fitting.png", width: 50%)

Figure 5: Left: distribution of total distance traveled; right: rectangle fit error from edge points.

4.4. Means Testing

4.4.1. Sensor Characterization

APDS9960 proximity sensing was validated across distances to set reliable thresholds for edge detection and gesture activation.

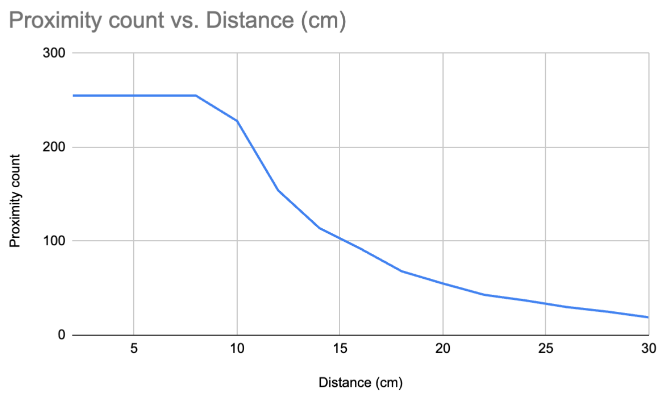

Figure 6: APDS9960 proximity response versus distance.

4.4.2. Motor Characterization

The motors where characterized by testing their anguar rotation accuracy against a rotary encoder in order to evaluate their error in odometry. Results showed highly accurate actuation, with a mean error of 0.01 degrees, a standard deviation of 0.03 degrees over 1190 steps. This is a very small error, and is therefore considered to be negligible in our odometry calculations.

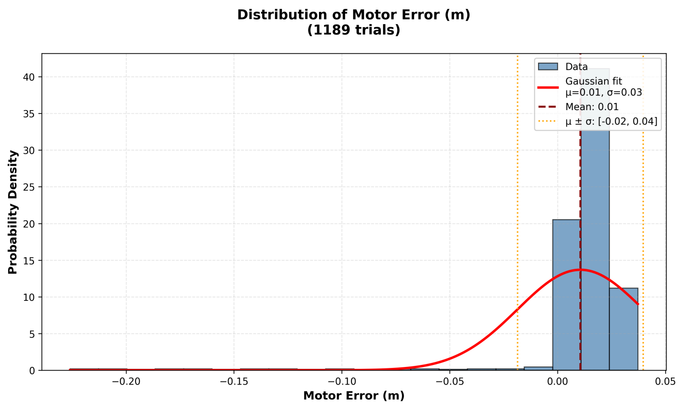

Figure 7: Distribution of motor error with Gaussian fit.

5. Lessons Learned

5.1. I2C Software-Hardware Timing

During the first prototype iteration, an I^{2}C expander MUX was used to manage what were initially four APDS9960 sensors for edge detection. The MUX also carried the IMU and an additional APDS9960 for gesture sensing, giving a total of six I^{2}C devices connected in parallel to a single hardware I^{2}C port (GPIO 2-3 on the Raspberry Pi 4B). This arrangement produced unusable data because of timing conflicts on the bus.

The Raspberry Pi's default hardware-I^{2}C clock rate is 100 kHz. Since the sensors were polled serially, the effective bus load rose significantly. This likely violated timing requirements, specifically when the effective clock period became smaller than the internal sensor timing constraints ($t_{\text{clk}} < t_{\text{clk-q}} + t_{\text{logic}} + t_{\text{setup}}$). As a result, the sensor frames arriving at the Pi were corrupted.

To address this, we attempted lowering the hardware I^{2}C clock in config.txt by adjusting the i2c_arm_baudrate of bus 1. Documentation and community reports reported that although the APDS9960 interface can technically support up to ≈ 200 kHz, the internal proximity engines update at only ≈ 100 Hz. Achieving stable communication required lowering the bus speed far below the Pi's default. Although this produced clean data, the resulting latency was too large, causing delayed edge detection and occasionally having the front ball caster falling off the table.

The successful solution was to transition to the Raspberry Pi's software-I^{2}C capability, which allows additional virtual I^{2}C buses to be instantiated on arbitrary GPIO pins. These software buses operate reliably around $10 - 50$ kHz depending on the bit-bang delay. We reduced the edge-detection system to two front-facing sensors (switching to a wall-following algorithm) and moved the remaining sensors onto three separate software I^{2}C buses. The gesture sensor remained on the hardware bus to preserve responsiveness and to have the two front sensors on similar hardware-software setups. The following overlays were added to boot/firmware/config.txt:

```
dtoverlay=i2c-gpio,bus=5,i2c_gpio_sda=6,i2c_gpio_scl=13,i2c_gpio_delay_us=50
dtoverlay=i2c-gpio,bus=7,i2c_gpio_sda=19,i2c_gpio_scl=26,i2c_gpio_delay_us=50
dtoverlay=i2c-gpio,bus=3,i2c_gpio_sda=24,i2c_gpio_scl=25,i2c_gpio_delay_us=50
```

With `i2c_gpio_delay_us=50`, the software I^{2}C bit-bang frequency is approximately: $f \approx \frac{1}{2\times\ \text{delay}} = \frac{1}{2\times 50\ \mu\text{s}} \approx$ 10 kHz. This configuration completely resolved the data-integrity issues and produced clean, stable measurements across the APDS9960s and the MPU6050, enabling reliable edge detection, gesture sensing, and inertial tracking.

5.2. Driver Current Limits and Microstepping Stability

In the first prototype we reused an Adafruit DC & Stepper Motor HAT (TB6612FNG). The TB6612 datasheet caps each H-bridge at 1.2A continuous, 3A peak, with no chopper current control. Our NEMA17s are rated $\approx 1.4 - 1.7 \frac{\text{A}}{\text{phase}}$ with low coil voltage ($\approx$ 3V), so driving them from a 12V supply depended on the HAT's PWM and coil inductance to self-limit. That worked for a few seconds, but as soon as we asked for "microstepping" the bridge hit its over-current/thermal limits: coil current drooped, the current vector collapsed toward full-step, and the motors audibly jittered and lost position. Scope traces on the coil showed the PWM duty shrinking after a short warm-up, matching the TB6612 thermal foldback behavior noted in the datasheet. The lesson was that a voltage-mode H-bridge without a per-phase current limit will not hold a sine profile on a motor that wants more current than the bridge can supply.

We replaced the HAT with two A4988 drivers. The A4988 datasheet gives 35V max V_{MOT} and up to $2\frac{\text{A}}{\text{phase}}$ with cooling, and—most importantly—a programmable current limit ($I_{\text{trip}} \approx \frac{V_{\text{ref}}}{8\cdot R_{\text{sense}}}$). We set V_{ref} for $\approx 1\frac{\text{A}}{\text{phase}}$

to stay under the NEMA17 rating while keeping the motors cool on a 12V rail. The A4988′s fixed 1/16-step microstep table keeps the phase currents sinusoidal regardless of supply voltage, and its built-in decay/chopper logic prevents the thermal collapse we saw on the TB6612. The board-level electrolytic caps also tame the VMOT spikes we observed when the HAT dumped current during stalls.

In software we leaned into the A4988′s requirements: we honor the 1 µs STEP pulse spec with a 10 µs pulse, cap the step frequency to 10 kHz, and ramp velocity with acceleration limits so we don't out-pace the current regulator. Microstepping is baked into the odometry, so we keep the resolution gains without losing steps. This stack (current-limited A4988 + timing- and acceleration-aware drive code) eliminated the mid-run jitter and gave us smooth, repeatable microstepping at the speeds needed for desk-safe motion.

6. Possible Future Improvements

6.1. Swarm Operation

The robot is currently designed to operate as a single unit, but it could be extended to operate as a swarm of robots by having multiple instances connected to the same network and operating on a request-based system instead of being individually controlled via ssh. This would allow multiple robots to clean different desks at the same time, and a central view of the entire fleet could be maintained to monitor their progress and status.

6.2. Battery Pack Unification

The robot is currently designed to use two separate battery packs, one for the motors/fan and one for the Raspberry Pi. This is because the motors require a higher voltage and current than the Raspberry Pi. However, this is not efficient as one system bottlenecks the other when it comes to operation time. A single larger battery pack that is shared between the motors and the Raspberry Pi through a voltage step-down regulator would allow the robot to operate for a longer time on a single charge, as well as simplifying the charging and attachment of the battery pack to the robot.

6.3. Vacuum System Improvement

The current vacuum system is not effective in collecting larger or denser debris than rice or small pieces of plastic. A more powerful vacuum fan paired with an additional reduction in the vacuum chute area would increase the suction velocity and therefore the efficacy of the vacuum system.

6.4. UX Simplification

The current UX is not very beginner and non-technical-user friendly. The need to ssh into the machine, manually download the code and run it makes it a tedious and technical process. Creating a simple web-app that can be downloaded as an executable and run locally would simplify the process and make it more accessible to a wider audience. The web-app would abstract the complexity of connecting to the robot by providing a simple interface to guide the users thorugh the process of connecting to the robot and running the cleaning process.

7. Appendix

7.1. Glass Box Diagram

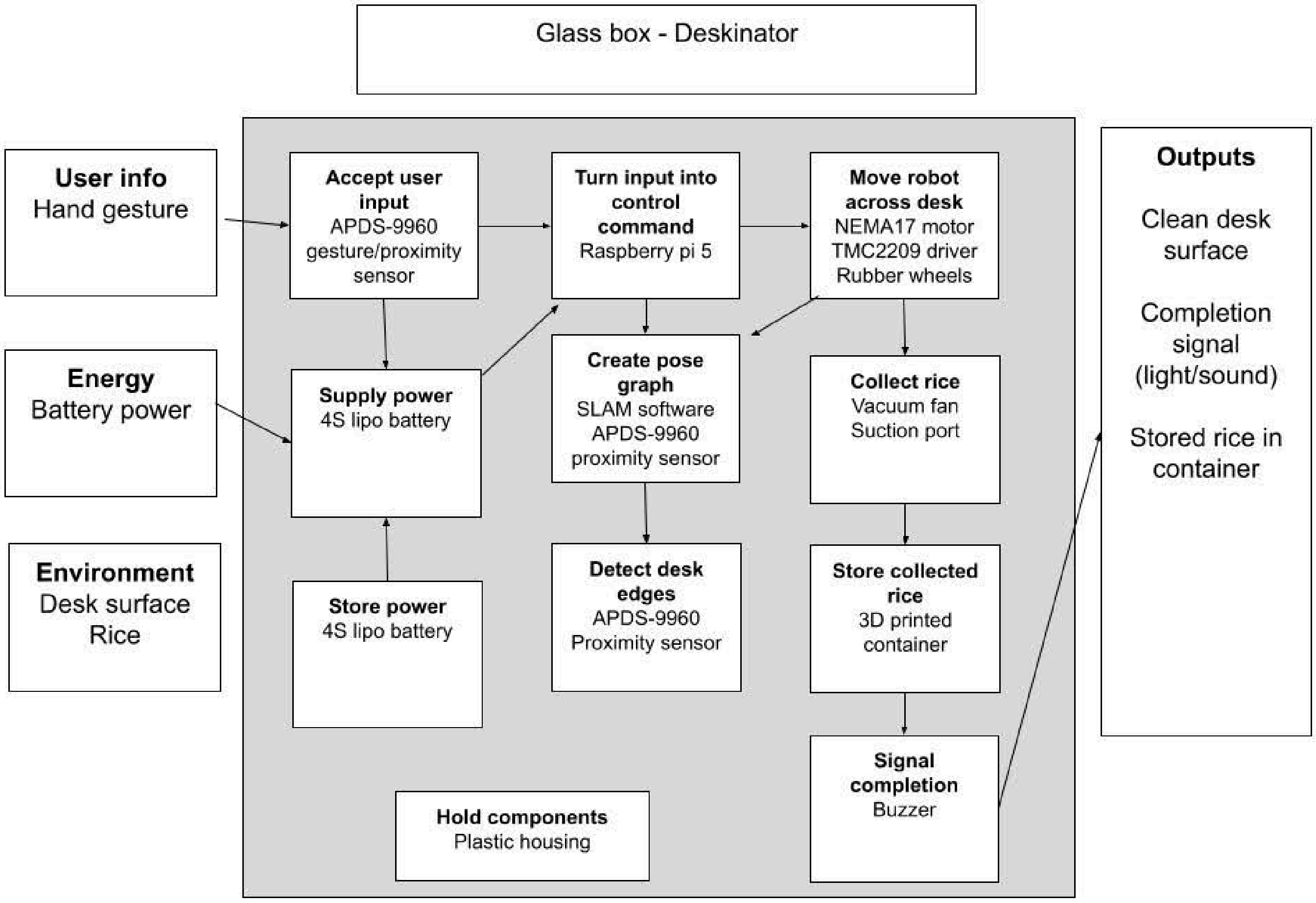

Figure 8: Glass box diagram of the Deskinator system.

7.2. PCC Table

Pairwise comparison of the design objectives used for the PCC. A value of 1 indicates the row objective dominated the column objective; totals are row sums.

Objectives	**A**	**P**	**A**	**E**	**D**	**M**	**P**	**E**	**R**	**N**	**C**	**T**	**S**	**A**	**C**	**E**	**TOTAL**
Accuracy	.	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	14
Precision	0	.	1	1	1	0	0	0	1	0	1	1	0	1	0	0	6
Affordability	0	0	.	1	1	0	0	0	0	0	0	1	0	0	1	0	4
Easy to clean	0	0	0	.	1	0	0	0	0	0	0	1	1	0	1	0	3
Durability	0	1	1	1	.	1	1	1	1	0	0	1	1	0	1	0	7
Mobility	0	1	1	1	1	.	1	1	1	0	0	1	1	1	0	1	10
Portable	0	1	1	1	0	0	.	1	0	0	0	0	0	1	0	0	4
Easy to use	0	0	1	0	1	0	0	.	1	0	1	1	1	1	1	0	7
Reliable	1	1	1	1	1	1	1	1	.	1	1	1	1	1	0	1	15
Navigation	0	0	0	1	0	1	0	0	0	.	1	1	1	1	0	1	10
Completion recognition	0	0	0	0	0	0	0	1	0	0	.	0	1	1	0	0	1
Touchless	0	1	1	1	1	1	0	1	1	0	0	.	1	1	1	0	7
Storage capacity	0	0	0	1	1	0	1	1	1	0	0	1	.	1	1	0	7

Objectives	A	P	A	E	D	M	P	E	R	N	C	T	S	A	C	E	TOTAL
Autonomy	0	0	0	1	1	1	0	1	1	0	1	1	1	.	1	0	10
Compactness	0	1	0	0	0	0	0	0	0	0	0	1	1	0	.	0	2
Efficiency	0	1	0	1	1	1	1	1	1	0	0	1	1	1	1	.	11

7.3. Morph Chart

Function	**Means 1**	**Means 2**	**Means 3**
Edge Detection	Downward APDS9960 proximity	Contact probes	Infrared distance sensors
Vacuum Suction	Small axial fan with funnel	Centrifugal impeller with dust box	Piezo microblower
Motion	2-wheel differential drive	4-wheel skid steer	Continuous track (tank)
Start Signal	Motion detection	Clap detection	Light signal
End Indicator	Buzzer	LED light	Screen display

7.4. Bill of Materials

Label	**Part Type**	**Price ($)**
100Ω-Res	100 Ω resistor	0.00
10kΩ-Res	10 kΩ resistor	0.00
12V-3A-10kmAh	12 V LiPo battery	18.48
5V-2.4A-5kmAh	5 V LiPo battery	18.48
DIODE	Rectifier diode	0.00
Driver-Left	Adafruit 6109 A4988	6.95
Driver-Right	Adafruit 6109 A4988	6.95
GESTURE	Adafruit APDS9960 proximity sensor	7.50
IMU	MPU6050	6.99
LED1	Red (633 nm) LED	0.00
LEFT	Adafruit APDS9960 proximity sensor	7.50
MOSFET	N-channel MOSFET	0.00
NEMA17-Left	NEMA 17 (17HS15-1504S-X1)	9.13
NEMA17-Right	NEMA 17 (17HS15-1504S-X1)	9.13
RBPi 4B	Raspberry Pi 4B	35.00
Resistor	220 Ω resistor	0.00
RIGHT	Adafruit APDS9960 proximity sensor	7.50
Vacuum-DC	12 V DC vacuum fan	25.00
Total	18 items	158.61

7.5. Electronics Sketch

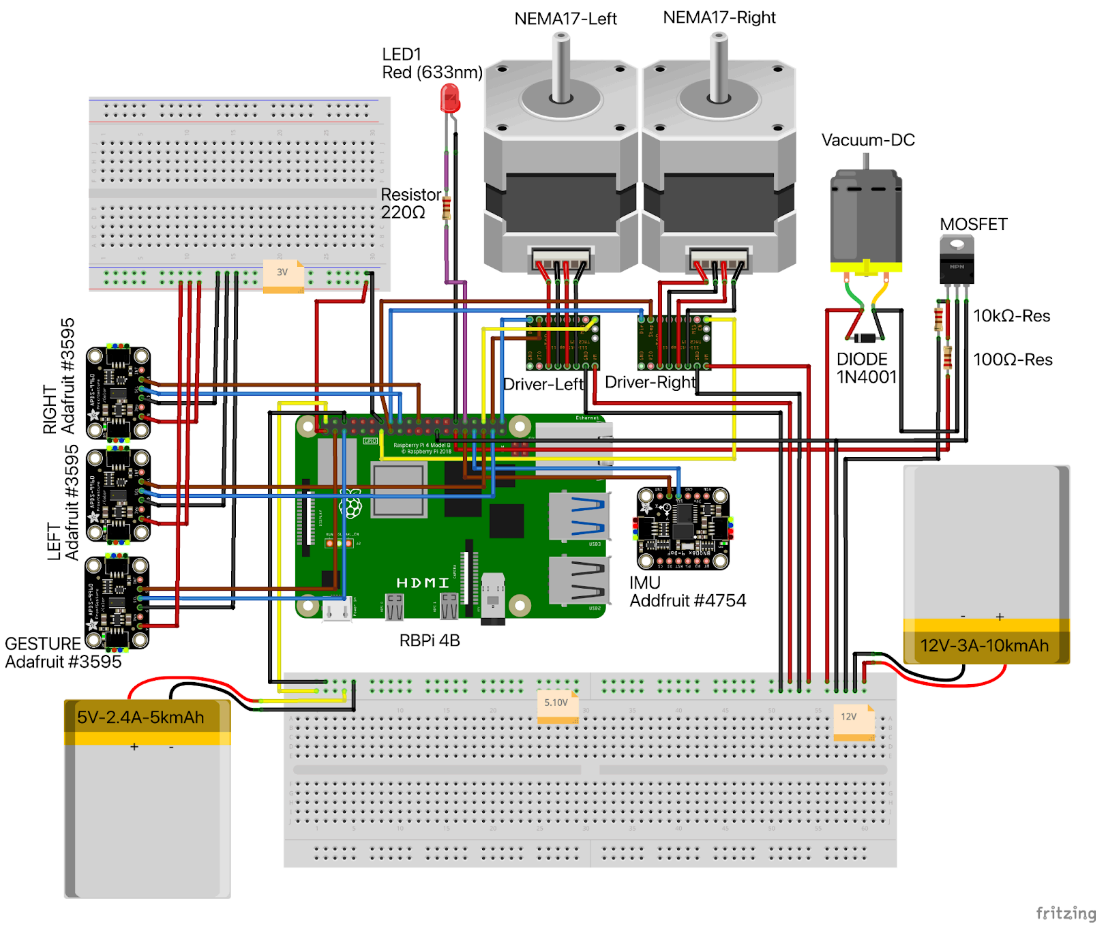

Figure 9: Electronics sketch showing power, drivers, sensors, and Pi interfaces.

7.6. Code Flow Chart

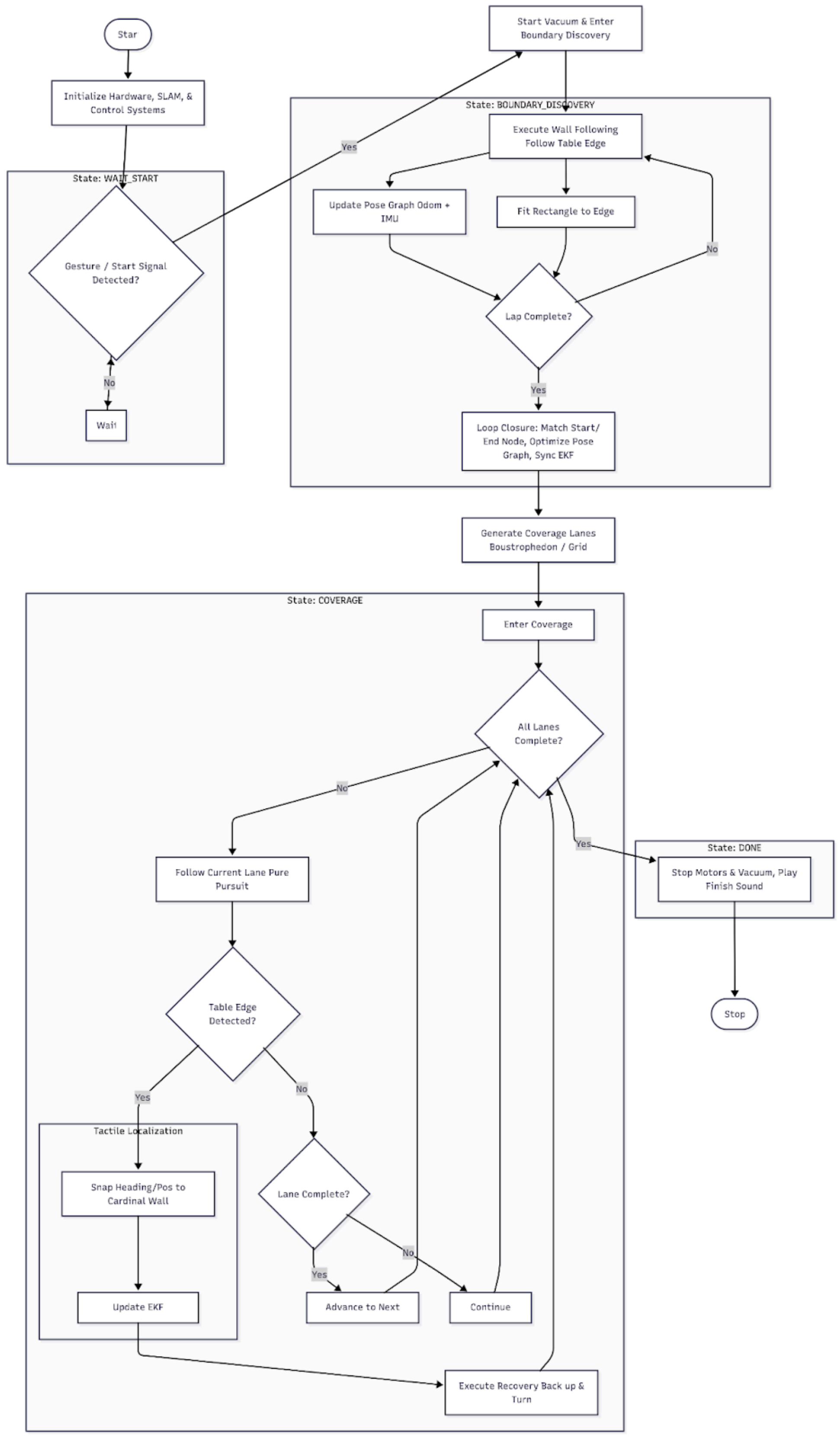
Star
Initialize Hardware, SLAM, & Control Systems
State: WAIT_START
Gesture / Start Signal Detected?
No
Wait
Yes
Start Vacuum & Enter Boundary Discovery
State: BOUNDARY_DISCOVERY
Execute Wall Following Follow Table Edge
Update Pose Graph Odom + IMU
Fit Rectangle to Edge
Lap Complete?
No
Yes
Loop Closure: Match Start/ End Node, Optimize Pose Graph, Sync EKF
Generate Coverage Lanes Boustrophedon / Grid
State: COVERAGE
Enter Coverage
All Lanes Complete?
No
Yes
State: DONE
Stop Motors & Vacuum, Play Finish Sound
Stop
Follow Current Lane Pure Pursuit
Table Edge Detected?
Yes
No
Tactile Localization
Snap Heading/Pos to Cardinal Wall
Update EKF
Lane Complete?
Yes
No
Advance to Next
Continue
Execute Recovery Back up & Turn

Figure 10: Software flow chart outlining sensing, localization, planning, and actuation.

7.7. CAD Drawings

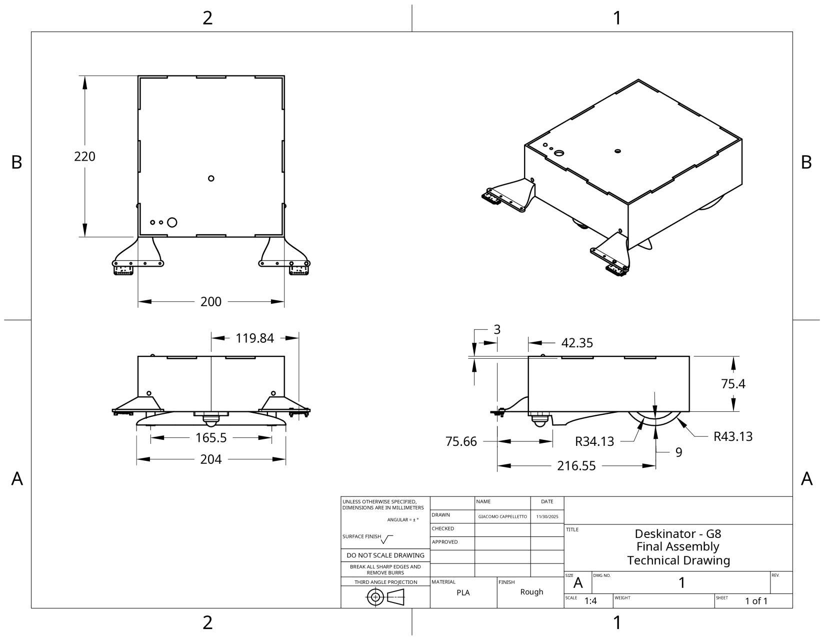

Figure 11: CAD drawing of the final assembly.

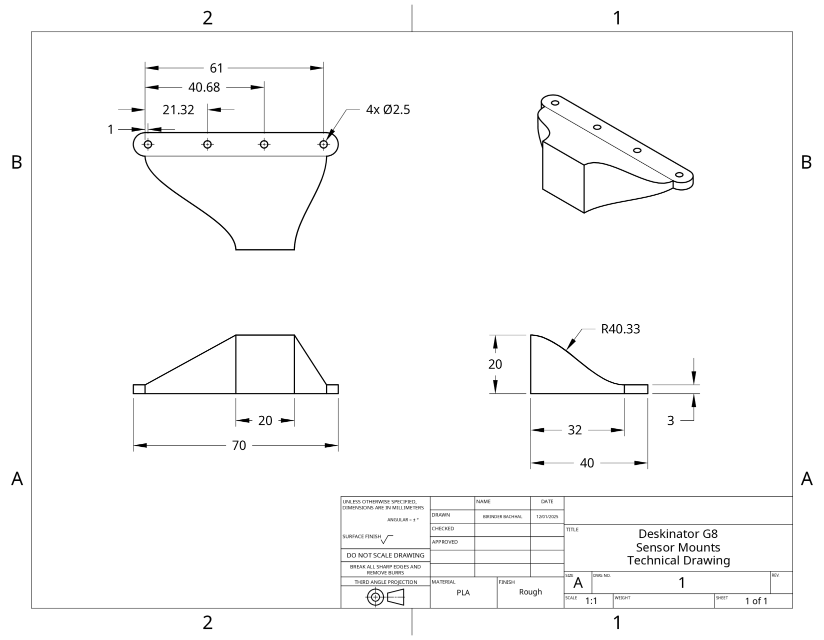

Figure 12: CAD drawing of the sensor mount.

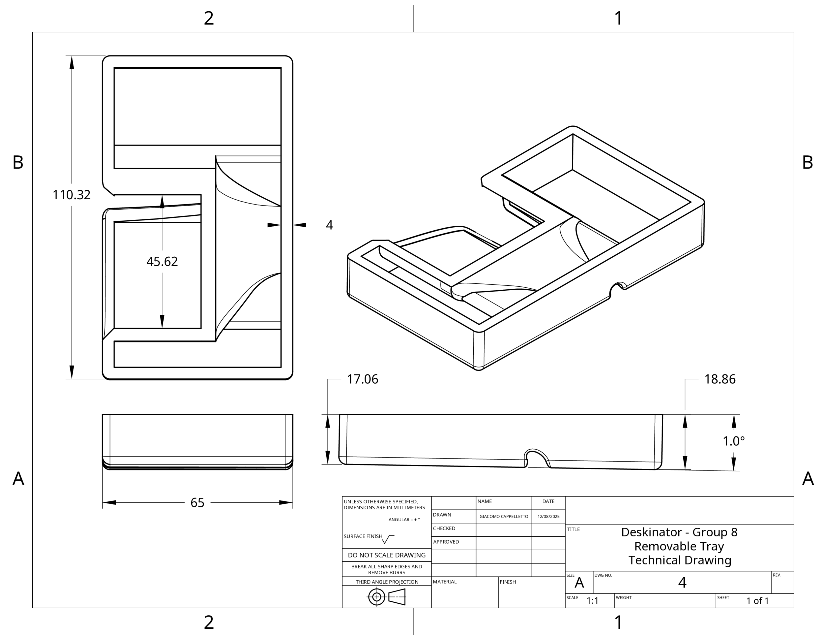

Figure 13: CAD drawing of the vacuum scoop.

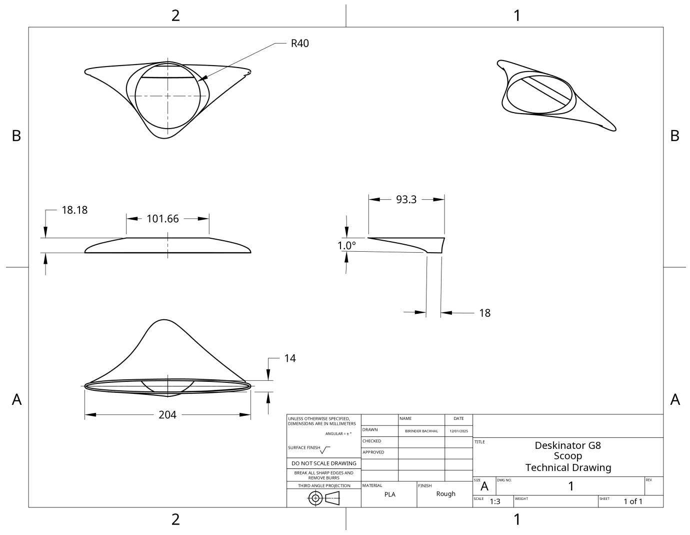

Figure 14: Vacuum scoop design based on continuity principle.

7.8. Power Budget

12 V system (motors + fan):

$$V_{\text{supply}} = 12\text{V}, \quad Q = 10\ 000 \text{ mAh} = 120 \text{ Wh}$$

$$P_{\text{fan}} \approx 12\text{W}, \quad P_{\text{motors}} \approx 20\text{W}, \quad P_{\text{total}} \approx 32\text{W}$$

$$t_{\text{ideal}} = \frac{Q}{P_{\text{total}}} = 120\frac{\text{Wh}}{32}\text{W} \approx 3.75\text{h}$$

$$t_{\text{realistic}(\approx 3.0\text{h})}$$

(after derating)

5 V system (Pi + sensors):

$$V_{\text{supply}} = 5\text{V}, \quad Q = 5000 \text{ mAh} = 25 \text{ Wh}$$

$$P_{\text{Pi}} \approx 5 - 8\text{W}, \quad P_{\text{sensors}} \approx 0.1\text{W}$$

$$t_{\text{runtime}} \approx 3.1 - 5.0\text{h}, \quad t_{\text{realistic}} \approx 3.5 - 4.0\text{h}$$

Overall:

$$\text{Limited by the 12V rail}$$

$$\text{Estimated continuous operation: } \approx 3 \text{ hours}$$

7.9. Code Repository

The full Python codebase (SLAM, planning, control, visualization, and tests) is in this git project repository.

https://github.com/JJCAPPE/deskinator.git