MAE 106 Final Project Report
Team Number: 57
Team Members: Alex Amadeo-Ranch, Alex Ye, Trevor Ngo, Isaiah Lu

System Description

1. Two main design objectives were to create a rigid, structurally stable chassis that could securely support the steering and propulsion systems, and to develop a control system that enabled reliable autonomous navigation through straight-line correction and a consistent 90-degree turn.

   The stability of our robot was one of the clear, defining features of our design. While other robots caved under their own weight from PLA plastic used in their steering design, we decided to secure parts of our steering system to the chassis itself, so the steering system could not cave unless the wooden chassis itself broke. Also, the stability of our chassis was much better than that of others. While other robots used L-brackets to secure smaller pieces of wood between larger plates, this made the entire robot wobbly due to poor manufacturing and assembly. Since we added slots to secure our full support, from the bottom of our chassis to the point that secured the tire plate, our robot was not unstable at all. These strategies proved very successful, as our robot didn't need to be fixed after assembly at all.
2. Figures 1-3 show three views of the final robot, including a side rear, and front view. These photographs document the completed physical system and illustrate the robot's overall mechanical layout. The images are annotated to identify the robot's major components. These annotated components include the chassis plates, wheels, servo motor, pneumatic cylinder, vertical support members, electronics/wiring, and the mounted upper structure.

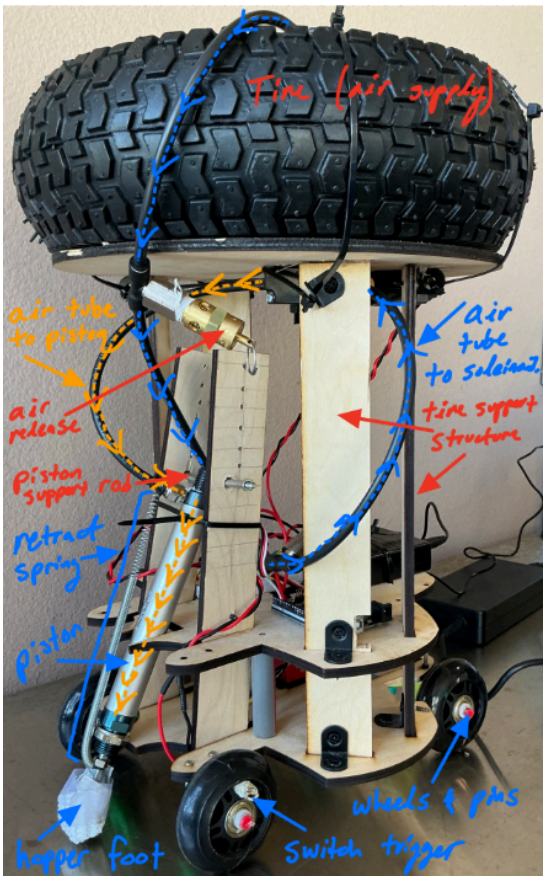

Figure 1: Rear View

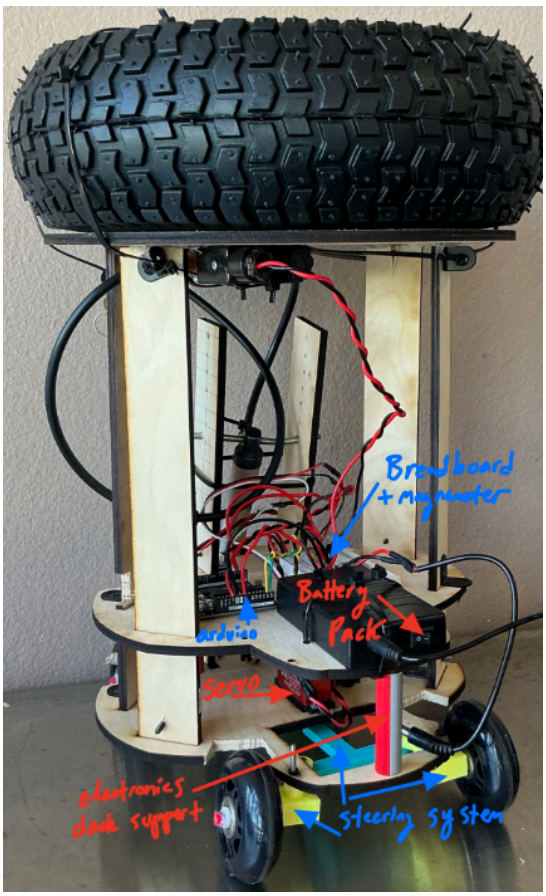

Figure 2: Front View

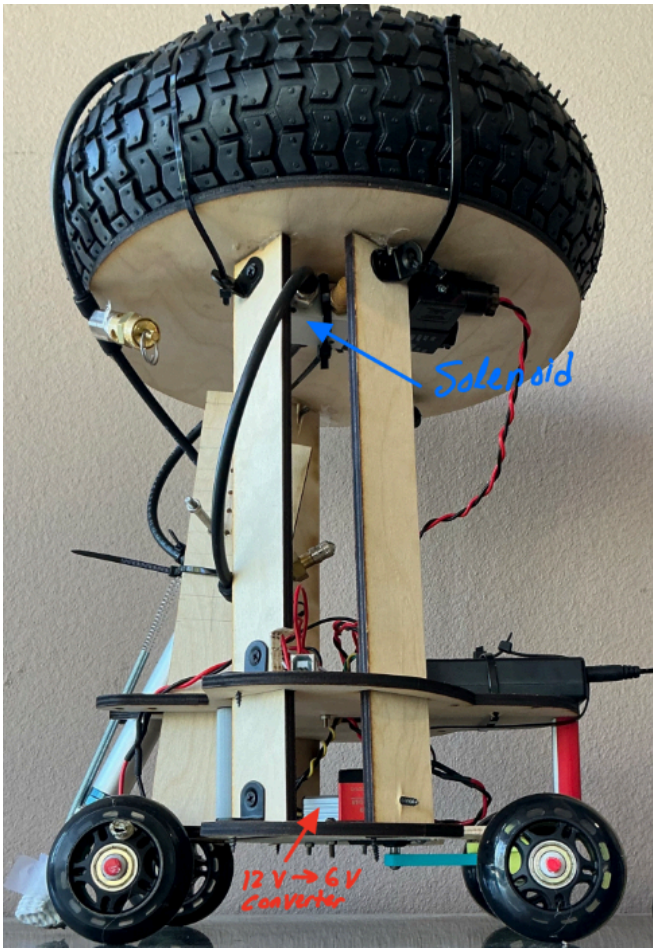

Figure 3: Side View

3. Figures 1-3 show three views of the final robot, including a side rear, and front view inside SOLIDWORKS. These depict the full chassis and rough overview of the electronic components, however, excluding the pneumatic tubing, components, breadboard, converter, and wiring. The images are annotated to identify the robot's major components. Figures 4-5 represent better views that accurately depict the validity of the dimensioning constraints including the height from the tire to the ground and the full chassis fitting within a 12” diameter circle

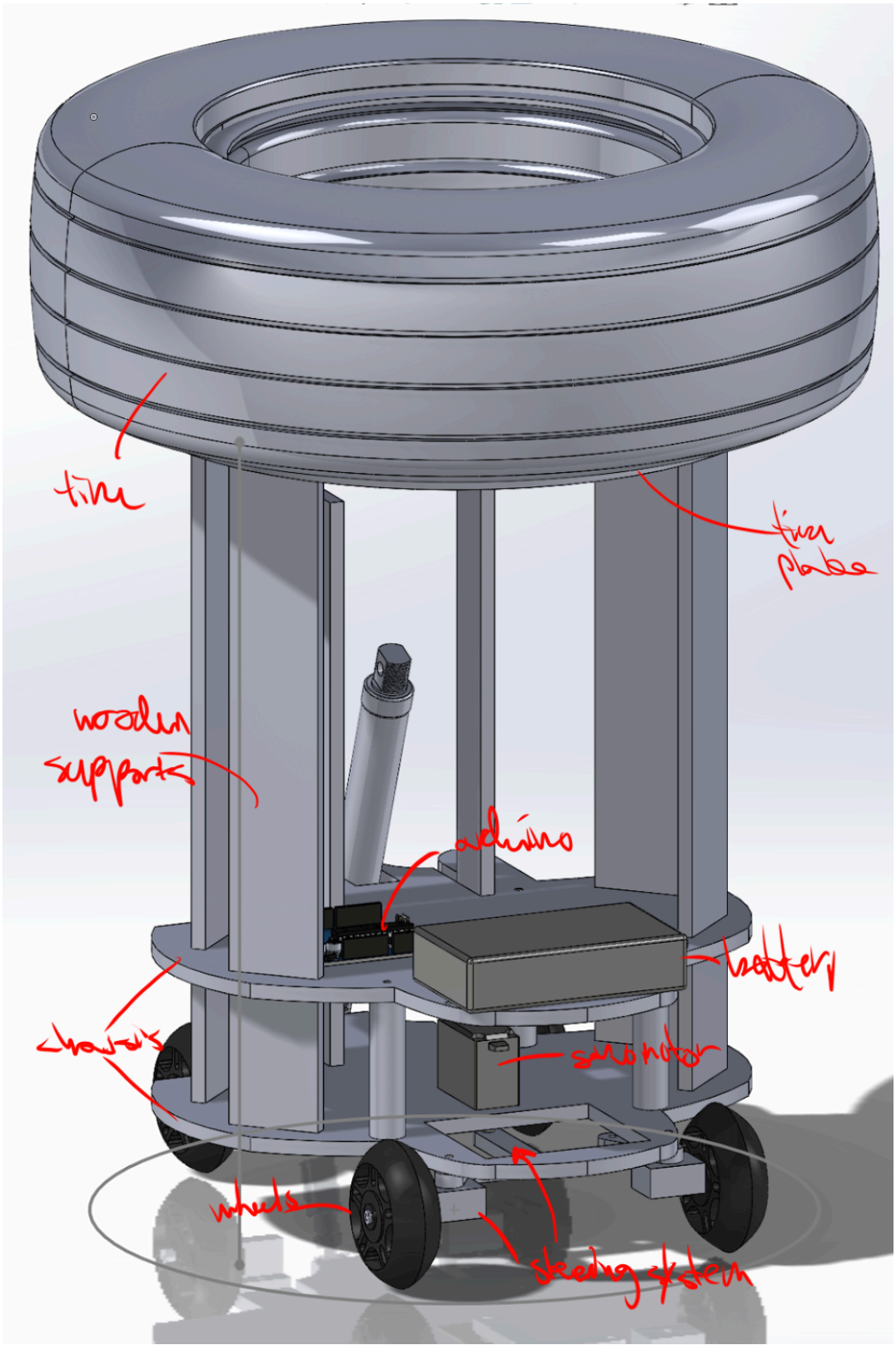

Figure 4: Front View

Figure 5: Side View

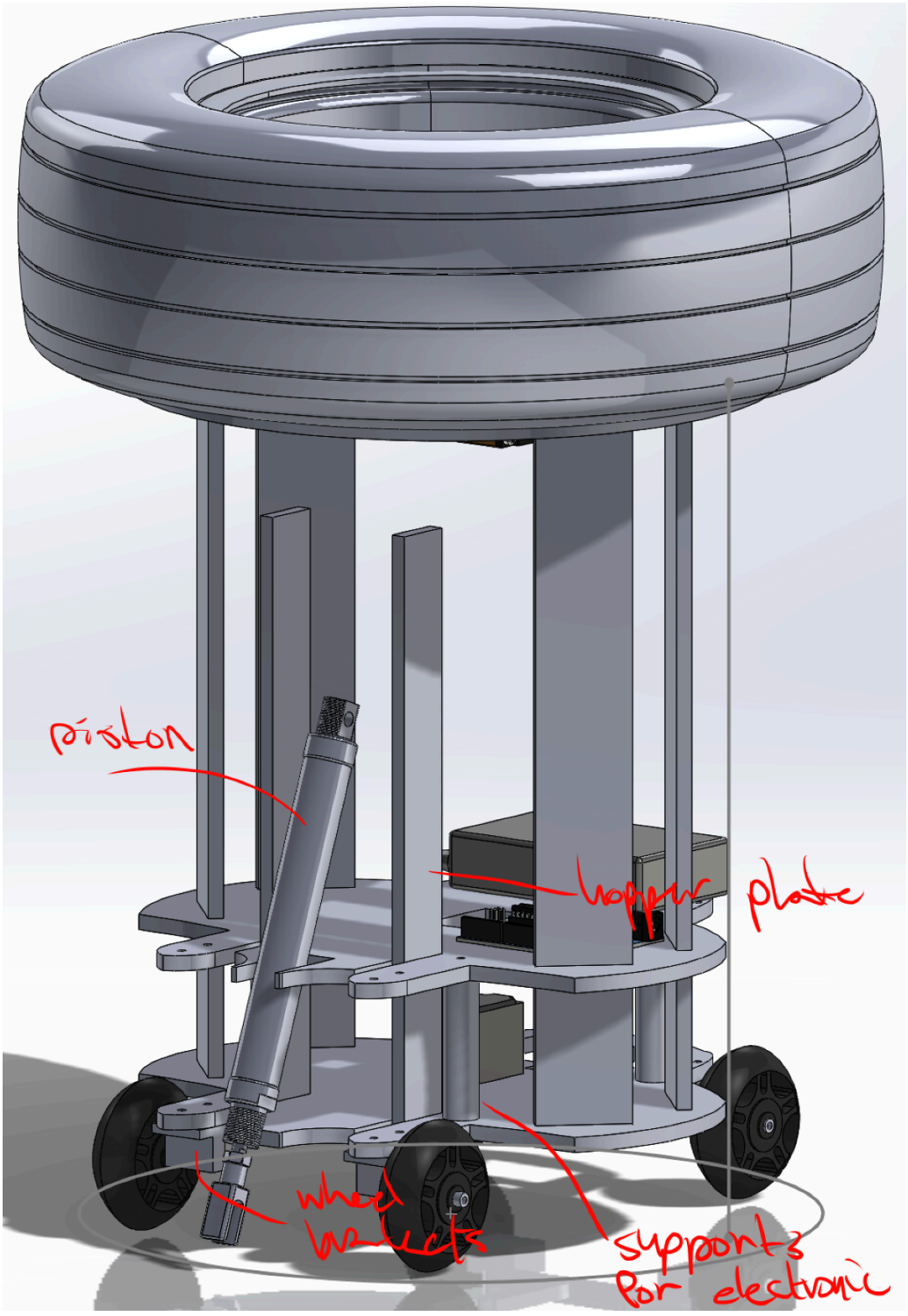

Figure 6: Rear View

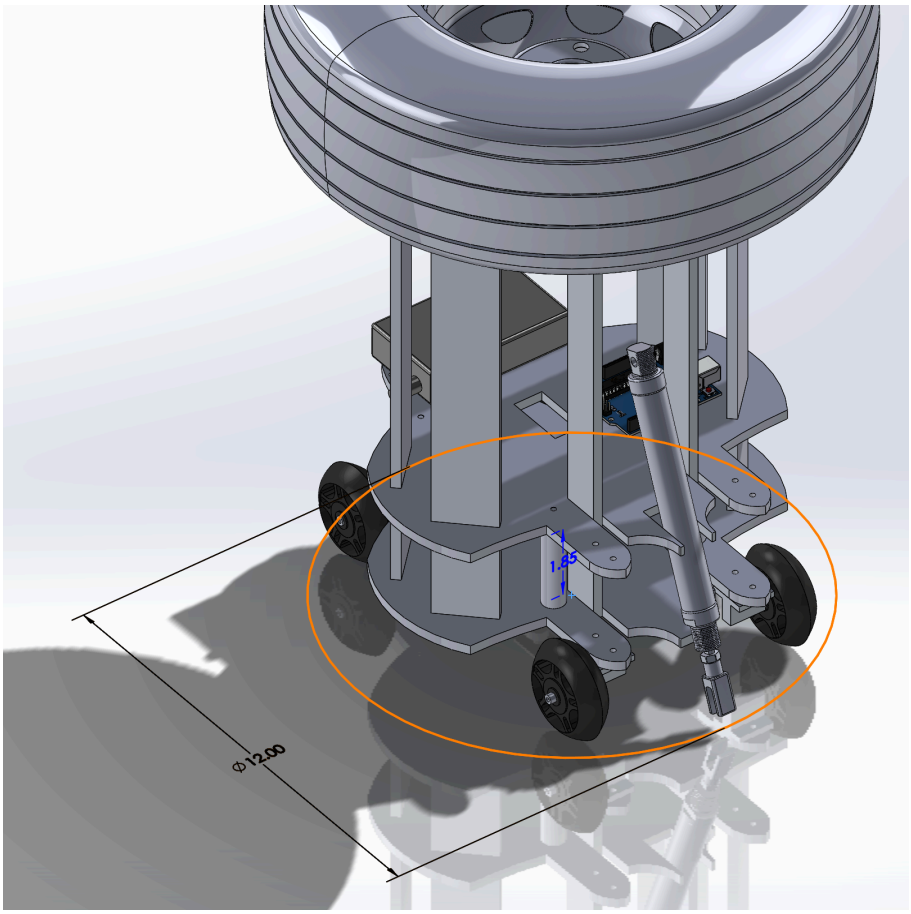

Figure 7: Dimensioned view showing that whole chassis fits into 12” diameter circle

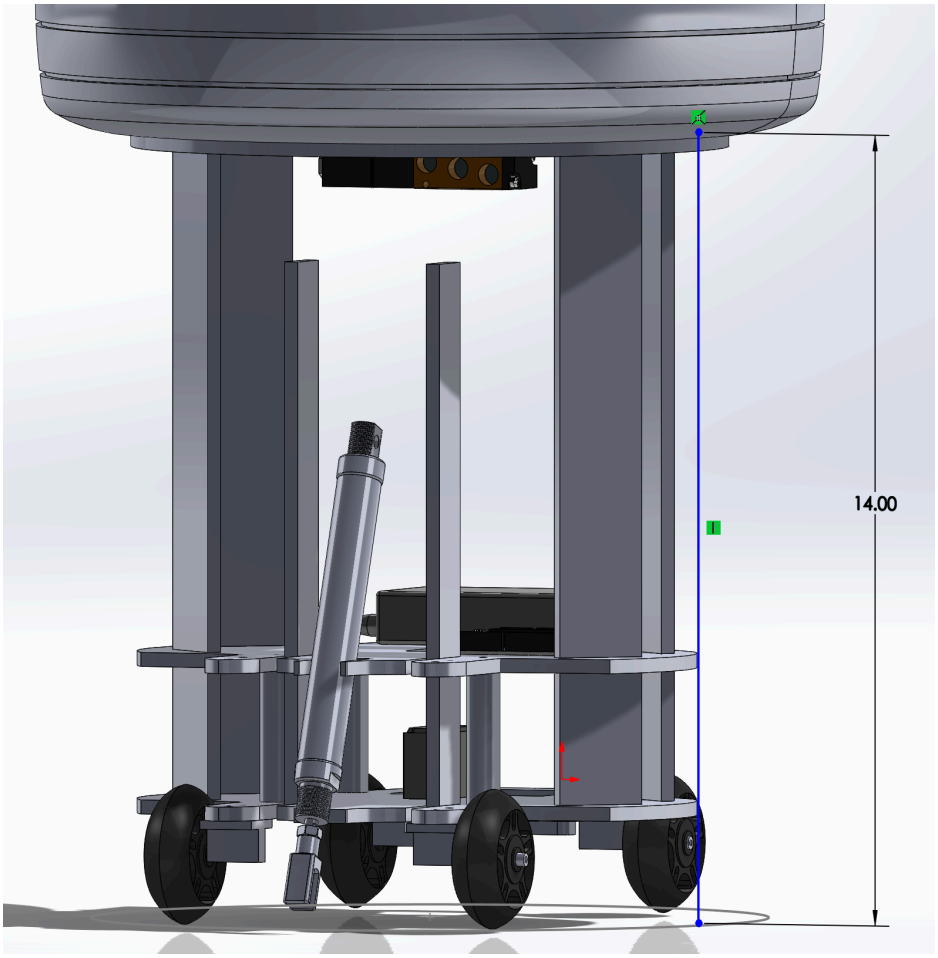

Figure 8: Dimensioned view showing height between top of tire plate and bottom is 14”

4. **Electrical Circuit Diagram**

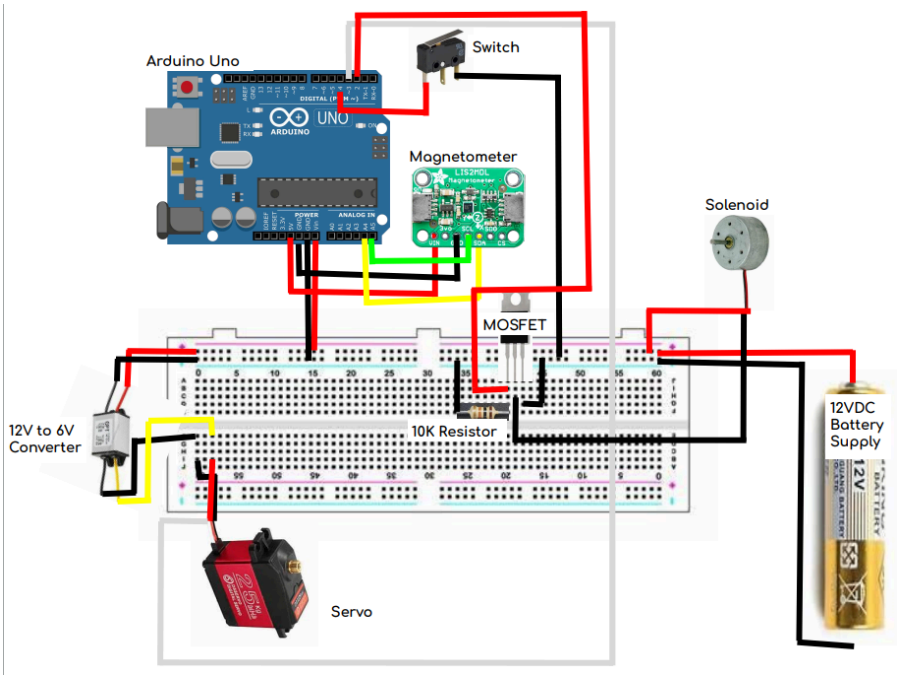

Figure 9: Circuit Diagram

5. The verification 2 code attached to the appendix below commands the robot to detect the bearing it is facing and continue on that path while adjusting for any potential steering errors that may occur. To accomplish this, the code uses a steering mechanism that corrects the steering angle when it detects a difference between the initial heading and the current heading. The piston will also fire simultaneously while this steering system is active, propelling the robot forward.

   The impulse response mechanism uses a similar idea to verification 2, using the same steering mechanism but capping the piston's propulsion to fire only 10 instances. The impulse response code also includes a switch detection mechanism that detects a full wheel revolution and adds the wheel's circumference to the total distance traveled. With this, we can determine the robot's position and velocity using mathematical equations. The code is also included in the appendix below.

Testing and Development

1. **Experimental Testing**:

   One performance criterion experimentally optimized for the robot was forward speed while maintaining stable motion. This criterion was related to the

pneumatic cylinder, since the solenoid timing directly affected how quickly and consistently the robot moved.

The experimental parameter varied was the pneumatic cylinder duty cycle, defined by the solenoid ON/OFF timing. At least two control parameters were kept constant during all experiments to ensure a fair comparison. These included the same floor surface, the same starting location, similar battery charge, and a constant pneumatic pressure of 30 psi. In all trials, the ON time was held constant at 100 ms. The OFF time was varied across five values: 400 ms, 650 ms, 900 ms, 1150 ms, and 1400 ms. These corresponded to duty cycles of 20%, 13.33%, 10%, 8%, and 6.67%, respectively. For each of the five duty-cycle settings, 10 runs were performed to characterize robot performance. The raw data for all trials are attached as Appendix [3]. The mean speed and standard deviation for each duty-cycle setting are summarized in Table [x].

G	H	I	J	K
On_ms	Off_ms	DutyCycle_pct	MeanSpeed_mps	SD_mps
100	400	20	2.4	0.183
100	650	13.33	2.124	0.101
100	900	10	1.79	0.071
100	1150	8	1.466	0.063
100	1400	6.67	1.2	0.047

Table [1]: Robot Speed vs. Solenoid Duty Cycle

The experimental results are shown in Figure [x], which plots the mean robot speed for each duty-cycle setting with standard deviation error bars. The data show that higher duty cycles generally produced greater average speed. The highest mean speed, approximately 2.40 m/s, occurred at the 100 ms ON / 400 ms OFF setting. As the OFF time increased and the duty cycle decreased, the robot's average speed decreased steadily.

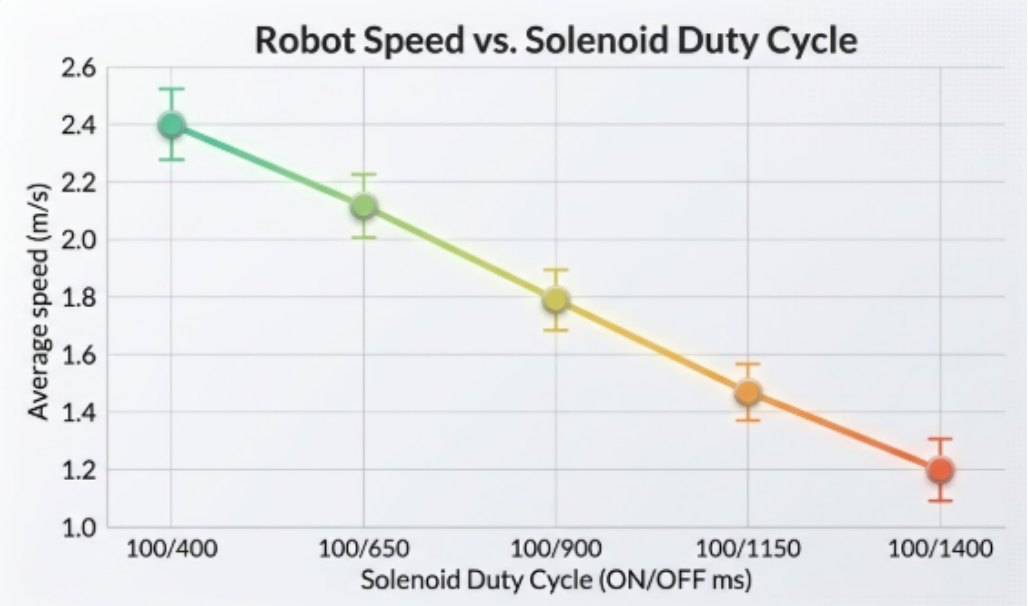

Figure [10]: Robot Speed vs. Solenoid Duty Cycle

Referring to Figure [x], the 100 ms ON / 900 ms OFF setting was chosen as the optimized parameter because it provided the best balance between speed and stability. Although the 100/400 setting produced the highest average speed, it also showed the largest standard deviation, indicating less consistent performance. In contrast, the 100/900 setting maintained a relatively high average speed while showing a smaller standard deviation, meaning the robot moved more consistently across repeated trials. Based on this tradeoff, 100 ms ON / 900 ms OFF was identified as the optimal duty cycle for achieving both quick and stable robot motion.

2. **Comparison with Mathematical Model:**

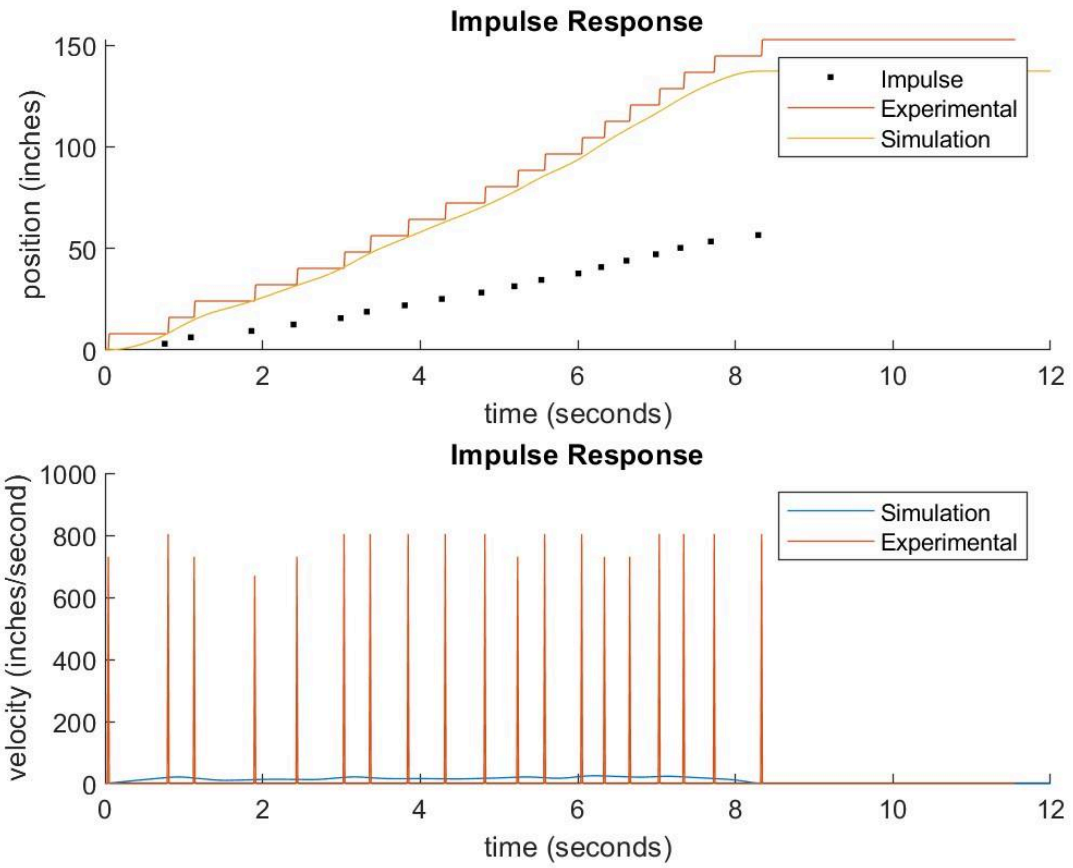

Figure 11: Impulse response plot

Our robot's actual performance is similar to the simulations, though the graph geometries differ. Our robot has also traveled a longer distance compared to the simulator's expected distance. Furthermore, our velocity graphs show spikes, with only one velocity value rather than a continuous curve as the simulator predicts.

The discrepancy between the simulator and the robot can be explained by the switch's function. Our robot can only detect when it has traveled a full revolution of the wheel, creating a sudden jump of distance traveled rather than a continuous mathematical model that the simulation operates on. This explains why our position-over-time graph has a stair-like geometry: our robot can only update its position in discrete increments, whereas the continuous model does not. The velocity graph follows a similar pattern, because the robot only detects a sudden change in position, the velocity suddenly spikes to a high value due to the change in time

of the distance traveled being so small. The mathematical model, on the other hand, does not have this limitation and can produce a continuous graph that more accurately represents the speed at which the robot is traveling, rather than detecting a sudden change when the switch is activated based on the rotation of the wheel.

Experimental Validation

The linked video demonstrates the robot completing the required motion sequence using a single Arduino sketch. In the video, the robot first travels forward at least 3 ft, then executes an approximately 90-degree turn, and then continues moving straight for at least another 3 ft before stopping actuation. Although the robot traveled slightly more than the minimum straight-line distances, this still satisfies the requirement because the objective was to achieve at least 3 ft before and after the turn. The video, therefore, verifies that the robot met the prescribed sequence of forward motion, turning, forward motion again, and final actuation termination under autonomous control. The script is shown in Appendix [4].

Video:
https://drive.google.com/file/d/1Xl7OVN0_UpDYF5JjkKur1msD2cRHLW9Z/view?usp=sharing

Summary of Contributions

All team members contributed meaningfully to the design, construction, testing, and improvement of the robot. Although each person had a primary role, the project required collaboration across mechanical design, controls, fabrication, wiring, and experimental testing. This shared effort helped the team solve problems more efficiently and ensured that the final robot functioned as an integrated system.

- Alex Amadeo-Ranch [Experimental]:
    - Determined the optimal pneumatic cylinder duty cycle by comparing robot speed and stability across repeated trials.
    - Collected, organized, and analyzed speed data, including calculating mean values and standard deviations for each test condition.
    - Calibrated the magnetometer and servo to improve the robot's sensing and steering performance.
    - Assisted with robot assembly and part manufacturing during the build process.
    - Helped troubleshoot circuit and hardware issues during integration and testing.
- Alex Ye [Controls]:
    - Planned structure and created Arduino code for robot by creating an impulse-response detector and counter-steering mechanism, resulting in autonomous steering correction in real time without user input.

    - Learned how to integrate physical electronic components to the Arduino board and utilize it in Arduino IDE such as pin initialization and optimizing parameters to ensure proper robot propulsion and steering.
    - Debugged and optimized magnetometer readings to help assist in direction detection, necessary in Verification #2 for counter steering and the impulse response test by detecting the robot moving 3ft and turning 90 degrees to continue another 3ft.
    - Assisted with Mechanical in design of the robot and learning how to utilize important tools such as 3D printing software (orcaslicer, Bambu labs, laser cutting, sawing). Also learned the mechanical steering mechanism to provide inputs to optimize steering consistency with the robot, and weight distribution (analyzing common areas of weight concentration to install support beams to the robot to ensure maximum stabilization).
    - Assisted with electrical in optimizing electronic placements on the robot in order to avoid as much interference as possible. Also optimized space and accessibility to electronic components to allow for ease of access when experimenting with the robot.
- Isaiah Lu [Electrical]:
    - Designed and created an electrical wiring diagram, clearly labeling all components and connections.
    - Created a comprehensive debugging manual for all electrical components, detailing symptoms of failure, testing procedures, and electrical diagrams.
    - Mounted and secured all electrical components onto the final robot, ensuring reliable connections and stress-relieved wiring for organized assembly.
    - Assisted with robot assembly and part manufacturing during the build process.
    - Assisted with Arduino programming, debugging code and iterating through tests to improve the robot's overall autonomous functionality.
- Trevor Ngo [Mechanical]:
    - Designed and optimized the chassis based on manufacturing, assembly, and dimensioning restrictions and constraints
    - Designed and optimized the propulsion and steering systems to create a robot capable of navigating through a complex maze and perform basic right turns and corrections along a straight line
    - Operated using power tools and CNC machinery for PLA plastic and wood to create a working chassis
    - Led robot assembly and part manufacturing during the build and optimization process
    - Assisted with experimental to help debug and calibrate the robot’s systems
    - Assisted with electrical in helping to place electronic components along the robot’s chassis including the reed switch system and wiring system to keep it efficient and clean

Two strategies that help ensure an engineering design team works well together and remains productive are clear role definition and regular communication, along with early testing with shared accountability.

First, a clear role definition combined with regular communication helps each team member understand their responsibilities while keeping the group coordinated. When tasks are divided based on team members' strengths, work can be completed more efficiently and with less confusion. Regular check-ins also allow the team to share progress, identify problems early, and ensure that the mechanical, electrical, and programming subsystems remain compatible.

Second, early testing with shared accountability improves both productivity and teamwork. Testing subsystems early and often helps the team detect design flaws before they become major problems late in the project. Shared accountability ensures that all members remain involved in the project as a whole rather than focusing only on their own part. This encourages collaboration, makes it easier to support one another when issues arise, and helps the team stay focused on meeting the overall design goals.

Appendix

[1] Verification 2 Code

```
#include <Servo.h>
#include <Wire.h>
#include <LIS3MDL.h>
#include <LSM6.h>

LIS3MDL mag; LSM6 imu; Servo myservo;

// --- Calibration ---
const float MAG_OFFSET_X = -838.5000;
const float MAG_OFFSET_Y = 3478.0;
const float MAG_SCALE_X  = 1883.5;
const float MAG_SCALE_Y  = 2060.0;

// --- Hardware Pins ---
const int servoPin = 3, solenoidPin = 2, reedPin = 4;
const int STEER_CENTER = 58, SERVO_MIN = 51, SERVO_MAX = 65;

// --- Piston & Distance ---
const unsigned long HOP_PULSE_MS = 100, RECHARGE_MS = 1000;
unsigned long lastHopTime = 0;
bool isHopping = false;

const float DIST_PER_TICK = 0.2042; // [cite: 10]
float distanceTravelled = 0;
int lastReedState = HIGH;
unsigned long lastClickTime = 0;
const int DEBOUNCE_DELAY = 200; // [cite: 15]

// --- Navigation States ---
enum State { IDLE, STRAIGHT_1, TURN_90, STRAIGHT_2, FINISHED };
State robotState = IDLE;
```

```
float desiredHeading = 0; // Set when 'g' is pressed
float turnStartHeading = 0;
float Kp = 0.55;
bool missionActive = false;

void setup() {
  Serial.begin(115200); // Higher baud rate for high-speed data logging
  Wire.begin();
  myservo.attach(servoPin);
  pinMode(solenoidPin, OUTPUT);
  digitalWrite(solenoidPin, LOW);
  pinMode(reedPin, INPUT_PULLUP); // [cite: 19]

  if (!mag.init() || !imu.init()) {
    while (1) { Serial.println(F("SENSOR ERROR")); delay(1000); }
  }
  mag.enableDefault(); imu.enableDefault();
  myservo.write(STEER_CENTER);
}

void loop() {
  // 1. Serial Commands
  if (Serial.available() > 0) {
    char cmd = Serial.read();
    if (cmd == 'g') {
      missionActive = true;
      distanceTravelled = 0;
      robotState = STRAIGHT_1;
      mag.read(); // Capture initial heading
      desiredHeading = calculateHeading();
      Serial.println(F("TIME(ms),DIST(m),STATE")); // Header for CSV data
    } else if (cmd == 'x') {
```

```
    stopMission();
  }
}

if (!missionActive) return;

// 2. Continuous Sensor Processing
float currentHeading = calculateHeading();
updateDistance();

// 3. State Machine Logic
unsigned long now = millis();

switch (robotState) {
  case STRAIGHT_1: // Move 3ft (~0.914m)
    runNavigation(desiredHeading);
    if (distanceTravelled >= 0.914) {
      robotState = TURN_90;
      turnStartHeading = currentHeading;
      distanceTravelled = 0; // Reset for next segment
      Serial.println(F(">> STARTING TURN"));
    }
    break;

  case TURN_90: // Turn at max radius until 90 degrees relative
    myservo.write(SERVO_MAX); // Hard right
    float angleTurned = abs(currentHeading - turnStartHeading);
    if (angleTurned > 180) angleTurned = 360 - angleTurned;

    if (angleTurned >= 90.0) { // Condition met
      robotState = STRAIGHT_2;
      desiredHeading = currentHeading; // Lock new heading
```

```
      distanceTravelled = 0;
      Serial.println(F(">> TURN COMPLETE"));
    }
    break;

  case STRAIGHT_2: // Move another 3ft
    runNavigation(desiredHeading);
    if (distanceTravelled >= 0.914) {
      robotState = FINISHED;
      stopMission(); //
    }
    break;
  }

  // 4. Actuation (Piston)
  if (missionActive && robotState != FINISHED) {
    if (!isHopping && (now - lastHopTime >= RECHARGE_MS)) {
      digitalWrite(solenoidPin, HIGH);
      lastHopTime = now;
      isHopping = true;
    } else if (isHopping && (now - lastHopTime >= HOP_PULSE_MS)) {
      digitalWrite(solenoidPin, LOW);
      isHopping = false;
    }
  }

  // 5. Data Logging for Impulse Response
  // Print values every loop for maximum resolution in velocity plots
  Serial.print(now); Serial.print(",");
  Serial.print(distanceTravelled, 4); Serial.print(",");
  Serial.println(robotState);
}
```

```
float calculateHeading() {
  mag.read();
  float mx = ((float)mag.m.x - MAG_OFFSET_X) / MAG_SCALE_X;
  float my = ((float)mag.m.y - MAG_OFFSET_Y) / MAG_SCALE_Y;
  float h = atan2f(mx, my) * 180.0f / PI;
  if (h < 0) h += 360.0f; //
  return h;
}

void updateDistance() {
  int currentReed = digitalRead(reedPin);
  unsigned long now = millis();
  if (currentReed == LOW && lastReedState == HIGH) {
    if (now - lastClickTime > DEBOUNCE_DELAY) {
      distanceTravelled += DIST_PER_TICK;
      lastClickTime = now;
    }
  }
  lastReedState = currentReed;
}

void runNavigation(float target) {
  float current = calculateHeading();
  float error = target - current;
  if (error > 180) error -= 360;
  if (error < -180) error += 360;

  int steerAdj = (int)(error * Kp) * -1; // Inverted
  myservo.write(constrain(STEER_CENTER + steerAdj, SERVO_MIN, SERVO_MAX));
}

void stopMission() {
  missionActive = false; robotState = IDLE;
  digitalWrite(solenoidPin, LOW);
  myservo.write(STEER_CENTER);
}
```

[2] Impulse Code

```
// --- MECHANICAL SWITCH DISTANCE COUNTER ---
int currentReedState = digitalRead(reedPin);
unsigned long now = millis();

// Check for "Falling Edge" (Switch just pressed)
if (trackingStarted && currentReedState == LOW && lastReedState == HIGH) {

  // DEBOUNCE: Only count if enough time has passed since the last click
  if (now - lastClickTime > DEBOUNCE_DELAY) {
    distanceTravelled += DIST_PER_TICK;
    lastClickTime = now; // Reset debounce timer

    Serial.print(F("PROGRESS: "));
    Serial.print(distanceTravelled, 3);
    Serial.print(F("m / "));
    Serial.print(TARGET_DIST);
    Serial.println(F("m"));

    if (distanceTravelled >= TARGET_DIST) {
      Serial.println(F("\n*** TARGET DISTANCE REACHED ***"));
      stopMission();
    }
  }
}
lastReedState = currentReedState;

// Steering Adjustment
if (abs(error) < DEADZONE) {
  myservo.write(STEER_CENTER);
} else {
  int steerAdj = (int)(error * Kp);
  if (invertSteering) steerAdj *= -1;
```

```
      myservo.write(constrain(STEER_CENTER + steerAdj, SERVO_MIN, SERVO_MAX));
    }

    // Solenoid Firing
    if (!isHopping && (now - lastHopTime >= RECHARGE_MS)) {
      digitalWrite(solenoidPin, HIGH);
      lastHopTime = now;
      isHopping = true;
    }
    else if (isHopping && (now - lastHopTime >= HOP_PULSE_MS)) {
      digitalWrite(solenoidPin, LOW);
      isHopping = false;
    }
  } else {
    stopMission();
  }

  // 4. Periodic Telemetry (Every 800ms)
  static unsigned long lastDebug = 0;
  if (millis() - lastDebug > 800) {
    Serial.print(F("HDG: ")); Serial.print(heading, 1);
    Serial.print(F(" | ERR: ")); Serial.print(error, 1);
    Serial.print(F(" | TOTAL DIST: ")); Serial.print(distanceTravelled, 2);
    Serial.print(F("m"));
    Serial.println(missionActive ? (trackingStarted ? F(" [RUN-LOCK]") : F(" [ALIGNING]")) : F(" [IDLE]"));
    lastDebug = millis();
  }
}

void stopMission() {
  missionActive = false;
  isHopping = false;

  digitalWrite(solenoidPin, LOW);
  myservo.write(STEER_CENTER);
}
```

[3] Experimental Testing Script

```
#include <Servo.h>

Servo steeringServo;

// ----------------------------- Pins -----------------------------
const uint8_t SOLENOID_PIN = 2;
const uint8_t SERVO_PIN    = 3;

// ------------------------ Steering setup ------------------------
const int STRAIGHT_SERVO_DEG   = 58;
const int SAFE_SERVO_LEFT_DEG  = 53;
const int SAFE_SERVO_RIGHT_DEG = 64;
const bool TURN_RIGHT          = true;
const int TURN_SERVO_DEG       = TURN_RIGHT ? SAFE_SERVO_RIGHT_DEG : SAFE_SERVO_LEFT_DEG;

// ------------------------ Pulse timing --------------------------
const unsigned long START_DELAY_MS            = 5000;
const unsigned long SOLENOID_ON_MS            = 100;
const unsigned long SOLENOID_OFF_MS           = 900;
const unsigned long TURN_SOLENOID_ON_MS       = 160;
const unsigned long TURN_SOLENOID_OFF_MS      = 700;
```

```
const unsigned long STRAIGHT_SERVO_SETTLE_MS  = 300;
const unsigned long TURN_SERVO_SETTLE_MS      = 150;
const unsigned long TURN_PULSE_HOLD_MS        = 40;
const unsigned long TURN_RECENTER_MS          = 700;
const unsigned long BETWEEN_PHASE_PAUSE_MS    = 250;

// ------------------------- Mission layout -------------------------
const int FIRST_STRAIGHT_PULSES = 2;
const int TURN_PULSES           = 1;

int safeServoAngle(int angleDeg) {
  return constrain(angleDeg, SAFE_SERVO_LEFT_DEG, SAFE_SERVO_RIGHT_DEG);
}

void holdSteering(int angleDeg, unsigned long holdMs) {
  unsigned long startMs = millis();
  int safeAngle = safeServoAngle(angleDeg);

  while (millis() - startMs < holdMs) {
    steeringServo.write(safeAngle);
    delay(20);
  }
}

void fireOnePulse(unsigned long onMs, unsigned long offMs) {
  digitalWrite(SOLENOID_PIN, HIGH);
  delay(onMs);
  digitalWrite(SOLENOID_PIN, LOW);
  delay(offMs);
}

void runStraightSegment(int pulseCount) {
  Serial.println(F("STRAIGHT SEGMENT"));
  holdSteering(STRAIGHT_SERVO_DEG, STRAIGHT_SERVO_SETTLE_MS);

  for (int i = 0; i < pulseCount; i++) {
    Serial.print(F("STRAIGHT_PULSE,"));
    Serial.println(i + 1);
    holdSteering(STRAIGHT_SERVO_DEG, 100);
    fireOnePulse(SOLENOID_ON_MS, SOLENOID_OFF_MS);
  }

  delay(BETWEEN_PHASE_PAUSE_MS);
}

void runTurnSegment() {
  Serial.println(F("TURN SEGMENT"));
  holdSteering(TURN_SERVO_DEG, TURN_SERVO_SETTLE_MS);

  for (int i = 0; i < TURN_PULSES; i++) {
    Serial.print(F("TURN_PULSE,"));
    Serial.println(i + 1);
    holdSteering(TURN_SERVO_DEG, TURN_PULSE_HOLD_MS);
    fireOnePulse(TURN_SOLENOID_ON_MS, TURN_SOLENOID_OFF_MS);
  }
}

void stopAllMotion() {
  digitalWrite(SOLENOID_PIN, LOW);
  steeringServo.write(safeServoAngle(STRAIGHT_SERVO_DEG));
}

void runMission() {
  Serial.println(F("MISSION START"));
  runStraightSegment(FIRST_STRAIGHT_PULSES);
```

```
  runTurnSegment();
  holdSteering(STRAIGHT_SERVO_DEG, TURN_RECENTER_MS);
  stopAllMotion();
  Serial.println(F("MISSION COMPLETE"));
}

void setup() {
  Serial.begin(115200);

  pinMode(SOLENOID_PIN, OUTPUT);
  digitalWrite(SOLENOID_PIN, LOW);

  steeringServo.attach(SERVO_PIN);
  steeringServo.write(safeServoAngle(STRAIGHT_SERVO_DEG));

  Serial.println(F("Validation video sketch ready."));
  Serial.print(F("Auto start in "));
  Serial.print(START_DELAY_MS);
  Serial.println(F(" ms"));

  delay(START_DELAY_MS);
  runMission();
}

void loop() {
  stopAllMotion();
  delay(1000);
}
```

[4] Experimental Validation Script

```
#include <Servo.h>

/*
  - solenoid MOSFET gate = D2
  - servo signal         = D3
  - wheel / limit switch = D4

// ------------------------------ Pins ------------------------------
const uint8_t SOLENOID_PIN     = 2;
const uint8_t SERVO_PIN        = 3;
const uint8_t LIMIT_SWITCH_PIN = 4;

// ---------------------- Main test settings ----------------------
// Change these for each new experimental case.
const unsigned long SOLENOID_ON_MS  = 100;
const unsigned long SOLENOID_OFF_MS = 1400;
const uint8_t NUMBER_OF_IMPULSES    = 10;

const unsigned long START_DELAY_MS = 5000;

// ---------------------- Control parameters ----------------------
// Keep these constant for the whole experimental test series.
const int STRAIGHT_STEERING_DEG = 58;
const int SAFE_SERVO_MIN_DEG    = 53;
const int SAFE_SERVO_MAX_DEG    = 64;

// -------------------- Wheel / speed settings --------------------
const float WHEEL_DIAMETER_M    = 0.070f;
const float DISTANCE_PER_TICK_M = PI * WHEEL_DIAMETER_M; // one click = one wheel revolution

// ------------------------ Switch filtering ------------------------
const unsigned long SWITCH_DEBOUNCE_MS = 80;
```

```
const unsigned long SAMPLE_PERIOD_MS   = 100;
const unsigned long MOTION_SETTLE_MS   = 1200;

Servo steeringServo;

bool testFinished = false;
bool solenoidIsOn = false;
uint8_t currentImpulse = 0;

long tickCount = 0;
int lastSwitchState = HIGH;
unsigned long lastDebounceMs = 0;
unsigned long runStartMs = 0;
unsigned long lastSampleMs = 0;
unsigned long previousTickMs = 0;
unsigned long lastTickMs = 0;

float positionM = 0.0f;
float lastTickSpeedMps = 0.0f;
float maxTickSpeedMps = 0.0f;

float dutyCyclePercent() {
  return 100.0f * ((float)SOLENOID_ON_MS / (float)(SOLENOID_ON_MS + SOLENOID_OFF_MS));
}

int safeStraightAngle(int angleDeg) {
  return constrain(angleDeg, SAFE_SERVO_MIN_DEG, SAFE_SERVO_MAX_DEG);
}

void stopAllMotion() {
  digitalWrite(SOLENOID_PIN, LOW);
  solenoidIsOn = false;
  steeringServo.write(safeStraightAngle(STRAIGHT_STEERING_DEG));
}

float currentVelocityEstimate() {
  unsigned long now = millis();

  if (lastTickMs == 0 || previousTickMs == 0) {
    return 0.0f;
  }

  if (now - lastTickMs > MOTION_SETTLE_MS) {
    return 0.0f;
  }

  return lastTickSpeedMps;
}

void printSettings() {
  Serial.println(F("---- EXPERIMENT SETTINGS ----"));
  Serial.print(F("Solenoid ON time (ms): "));
  Serial.println(SOLENOID_ON_MS);
  Serial.print(F("Solenoid OFF time (ms): "));
  Serial.println(SOLENOID_OFF_MS);
  Serial.print(F("Duty cycle (%): "));
  Serial.println(dutyCyclePercent(), 2);
  Serial.print(F("Number of impulses: "));
  Serial.println(NUMBER_OF_IMPULSES);
  Serial.print(F("Straight steering angle (deg): "));
  Serial.println(safeStraightAngle(STRAIGHT_STEERING_DEG));
  Serial.print(F("Distance per tick (m): "));
  Serial.println(DISTANCE_PER_TICK_M, 4);
  Serial.println(F("-------------------------------"));
}
```

```
void printCsvHeader() {
  Serial.println(F("event,time_ms,position_m,velocity_mps,tick_count,impulse_number,solenoid_on"));
}

void logRow(const char* eventLabel) {
  unsigned long elapsedMs = millis() - runStartMs;

  Serial.print(eventLabel);
  Serial.print(',');
  Serial.print(elapsedMs);
  Serial.print(',');
  Serial.print(positionM, 4);
  Serial.print(',');
  Serial.print(currentVelocityEstimate(), 4);
  Serial.print(',');
  Serial.print(tickCount);
  Serial.print(',');
  Serial.print(currentImpulse);
  Serial.print(',');
  Serial.println(solenoidIsOn ? 1 : 0);
}

void resetRunVariables() {
  tickCount = 0;
  positionM = 0.0f;
  previousTickMs = 0;
  lastTickMs = 0;
  lastTickSpeedMps = 0.0f;
  maxTickSpeedMps = 0.0f;
  lastDebounceMs = 0;
  lastSwitchState = digitalRead(LIMIT_SWITCH_PIN);
  lastSampleMs = 0;
  currentImpulse = 0;
}

void updateSwitchState() {
  unsigned long now = millis();
  int currentSwitchState = digitalRead(LIMIT_SWITCH_PIN);

  if (currentSwitchState == LOW && lastSwitchState == HIGH) {
    if (now - lastDebounceMs >= SWITCH_DEBOUNCE_MS) {
      tickCount++;
      positionM = tickCount * DISTANCE_PER_TICK_M;

      previousTickMs = lastTickMs;
      lastTickMs = now;

      if (previousTickMs > 0) {
        float dtSeconds = (lastTickMs - previousTickMs) / 1000.0f;
        if (dtSeconds > 0.0f) {
          lastTickSpeedMps = DISTANCE_PER_TICK_M / dtSeconds;
          if (lastTickSpeedMps > maxTickSpeedMps) {
            maxTickSpeedMps = lastTickSpeedMps;
          }
        }
      }

      lastDebounceMs = now;
      logRow("TICK");
    }
  }

  lastSwitchState = currentSwitchState;
```

```
  if (now - lastSampleMs >= SAMPLE_PERIOD_MS) {
    lastSampleMs = now;
    logRow("SAMPLE");
  }
}

void waitWithLogging(unsigned long waitMs) {
  unsigned long waitStartMs = millis();
  while (millis() - waitStartMs < waitMs) {
    updateSwitchState();
  }
}

void setSolenoid(bool turnOn) {
  solenoidIsOn = turnOn;
  digitalWrite(SOLENOID_PIN, turnOn ? HIGH : LOW);
  logRow(turnOn ? "SOLENOID_ON" : "SOLENOID_OFF");
}

void waitUntilRobotStops() {
  unsigned long quietReferenceMs = millis();

  if (lastTickMs > 0) {
    quietReferenceMs = lastTickMs;
  }

  while (true) {
    updateSwitchState();

    unsigned long referenceMs = (lastTickMs > 0) ? lastTickMs : quietReferenceMs;
    if (millis() - referenceMs >= MOTION_SETTLE_MS) {
      break;
    }
  }
}

void printSummary() {
  unsigned long totalTimeMs = millis() - runStartMs;
  float totalDistanceM = positionM;
  float averageSpeedMps = 0.0f;

  if (totalTimeMs > 0) {
    averageSpeedMps = totalDistanceM / (totalTimeMs / 1000.0f);
  }

  Serial.print(F("SUMMARY,"));
  Serial.print(SOLENOID_ON_MS);
  Serial.print(',');
  Serial.print(SOLENOID_OFF_MS);
  Serial.print(',');
  Serial.print(dutyCyclePercent(), 2);
  Serial.print(',');
  Serial.print(NUMBER_OF_IMPULSES);
  Serial.print(',');
  Serial.print(totalTimeMs);
  Serial.print(',');
  Serial.print(totalDistanceM, 4);
  Serial.print(',');
  Serial.print(averageSpeedMps, 4);
  Serial.print(',');
  Serial.print(maxTickSpeedMps, 4);
  Serial.print(',');
  Serial.println(tickCount);
}
```

```
void runExperimentalTest() {
  resetRunVariables();
  runStartMs = millis();
  lastSampleMs = runStartMs;

  printSettings();
  printCsvHeader();
  logRow("RUN_START");

  for (currentImpulse = 1; currentImpulse <= NUMBER_OF_IMPULSES; currentImpulse++) {
    setSolenoid(true);
    waitWithLogging(SOLENOID_ON_MS);

    setSolenoid(false);

    if (currentImpulse < NUMBER_OF_IMPULSES) {
      waitWithLogging(SOLENOID_OFF_MS);
    }
  }

  waitUntilRobotStops();
  logRow("RUN_END");
  printSummary();
  stopAllMotion();
}

void setup() {
  Serial.begin(115200);

  pinMode(SOLENOID_PIN, OUTPUT);
  digitalWrite(SOLENOID_PIN, LOW);

  pinMode(LIMIT_SWITCH_PIN, INPUT_PULLUP);
  lastSwitchState = digitalRead(LIMIT_SWITCH_PIN);

  steeringServo.attach(SERVO_PIN);
  steeringServo.write(safeStraightAngle(STRAIGHT_STEERING_DEG));

  delay(1000);

  Serial.println(F("Auto-run experimental testing sketch ready."));
  Serial.print(F("Starting in "));
  Serial.print(START_DELAY_MS);
  Serial.println(F(" ms..."));

  delay(START_DELAY_MS);
}

void loop() {
  if (!testFinished) {
    runExperimentalTest();
    testFinished = true;
  }

  stopAllMotion();
}
```