Team 309- ENGRMAE 106 Final Report

Gavin Fujimoto, Matthew Scott, Henry Nguyen, Erin Lee

Table of Contents

1. System Description

1.1. Design Concept

1.1.1. Mechanical and Software Design Concepts:

The Mechanical design concept for the robot is a cylindrical three layer design that incorporates the pneumatic piston as a hopper that will strike the ground as the propulsion device. The robot's steering is reliant on the servo that is mounted onto the bottom-most plate and serves as a parallel steering mechanism that works in conjunction with the piston to move the robot. Each layer serves a purpose, with the top-most layer carrying the weight of the pneumatic tire and the middle most layer is where most of the electrical components are stationed to not interfere and be safe from the rest of the robot.

The robot makes use of a hybrid control system, a combination of both open- and closed-loop operation. During the open-loop control, the robot is instructed to drive forward a set distance as gauged by the limit switch before beginning a turn into the channel. In the closed-loop phase, the robot uses the magnetometer to direct itself towards the center of the channel. The program compares current heading to a desired heading to correct steering and operates the piston based on asymmetric on-off timing.

1.1.1.1. **Design Goals**:

Mechanical	Software
- Work closely with the software role to closely integrate software and hardware; design with software capabilities/limitations in mind. - Ensure every design choice is within the	- Develop a flexible program structure to account for physical uncertainties and adjust for desired characteristics - Maintain legible code to facilitate use by other roles and

parameters set by the project's constraints and ensure each physical property is feasible to create and is structurally sound.	allow for quick changes to critical sections

1.1.1.2. Final Design Discussion

The final design of the robot was set apart from the other respective robots because of the simplified design. Many other group's robots were reliant on having the best looking robot or were over-engineered to complete a task that was otherwise straightforward and simple. Because more time was invested in the proper testing of the robot and making sure the robot was operational ever since it was built, a lot of headaches of redesigning the robot was prevented and overall making the robot functional by the end.

In terms of software, the robot features an organized and robust program with variables for all sensors and controls (servo, magnetometer, pistons) with careful attention to optimizing performance. For example, the limit switch is programmed to only register after being pressed and released again without relying on a debounce timer and the servo features both adjustments to account for heading overflow and to accurately map the input angle to the actual output. Code is sectioned by feature and commented; variables are self-descriptive. Outside of physical limitations, the program allows for a wide range of experimental adjustments contributing to consistent performance. The flexibility allowed the team to easily address issues including oversteer, tipping, and noise. Additionally, the structure of the code allowed for a single framework to be easily adjusted to satisfy multiple use cases including Verification #2, Hybrid and Waypoint control systems, experimental testing, and impulse response collection.

1.2. Final Design CAD Model and Photo:

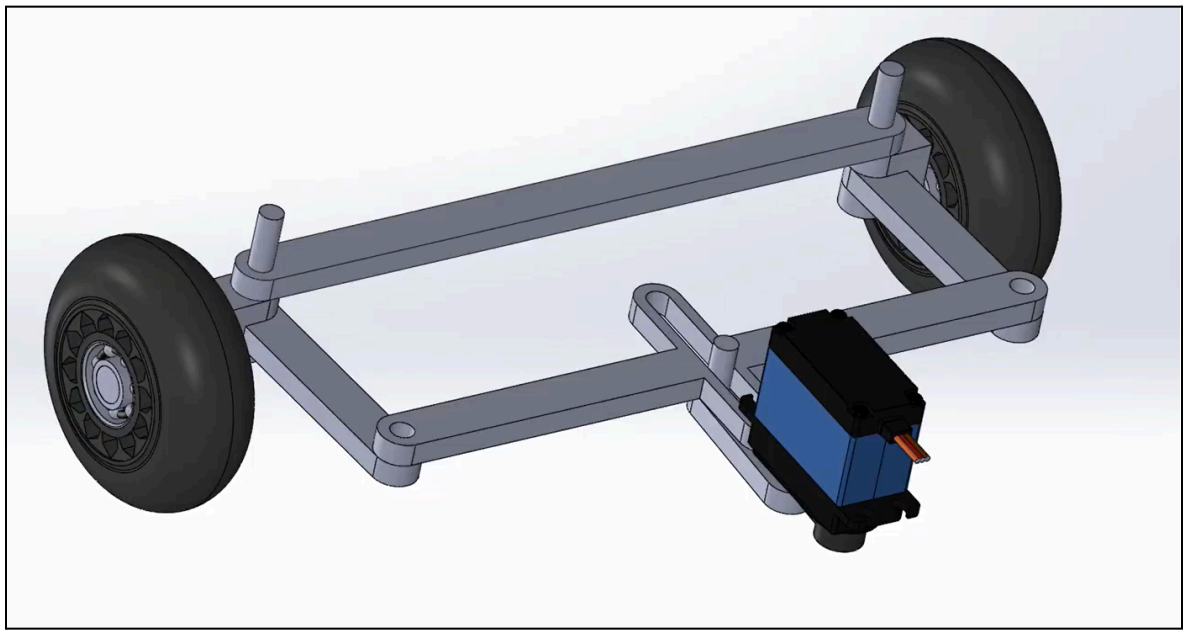

Figure 1: CAD Steering mechanism.

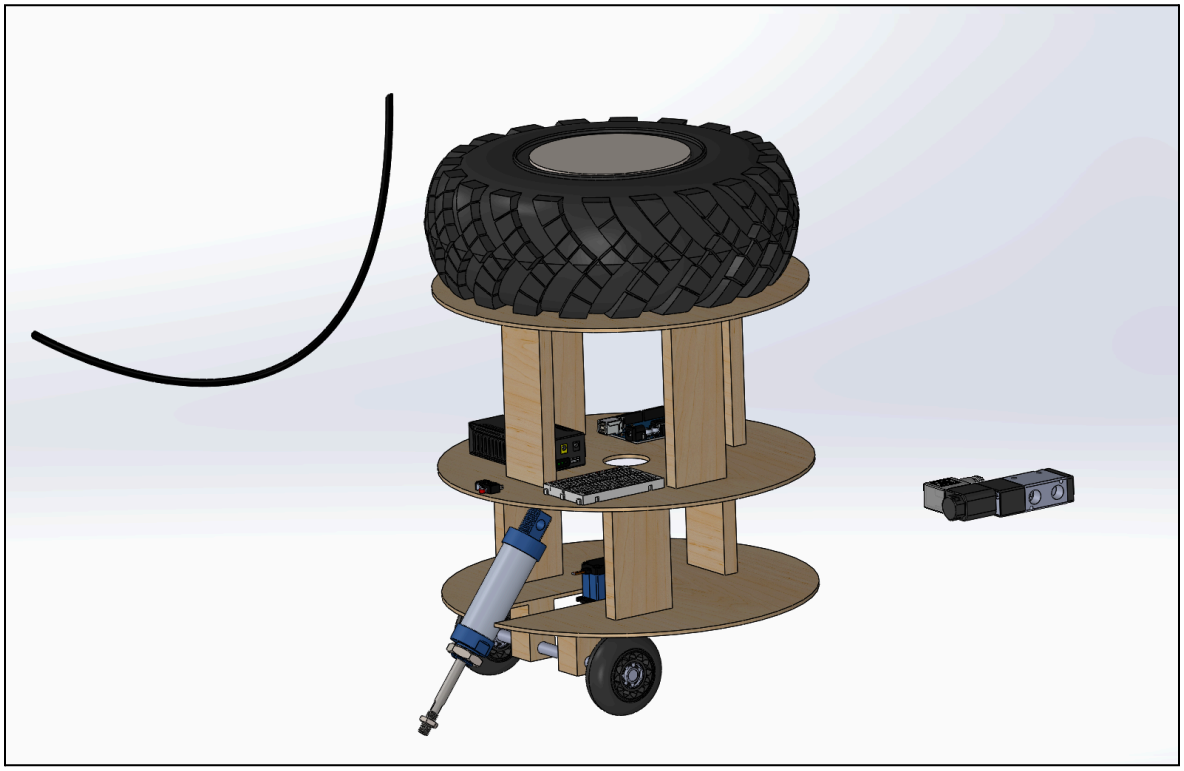

Figure 2: CAD Isometric view of wooden structure of the robot.

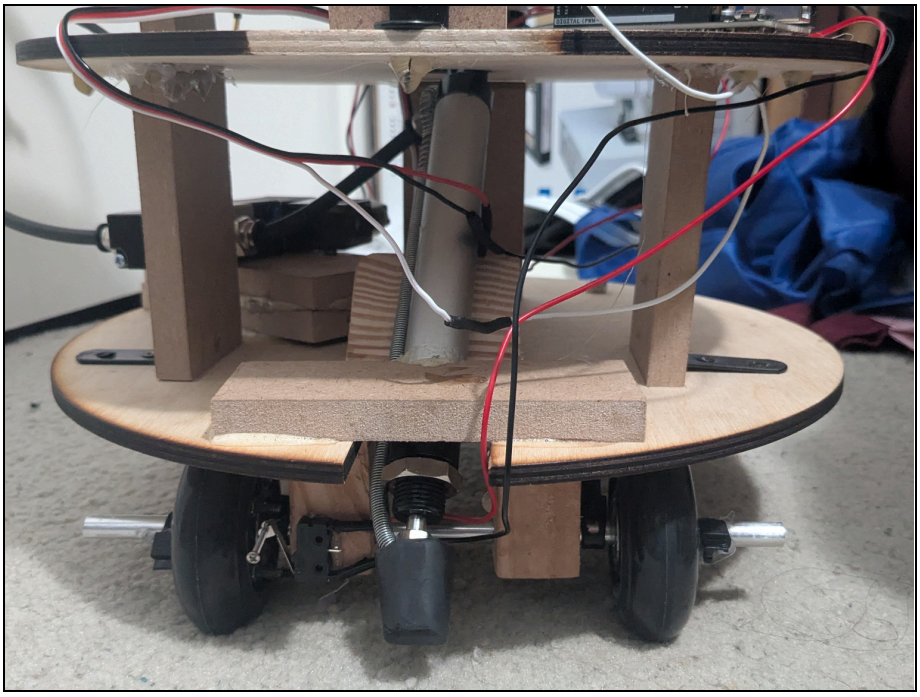

Figure 3: Propulsion system for the robot.

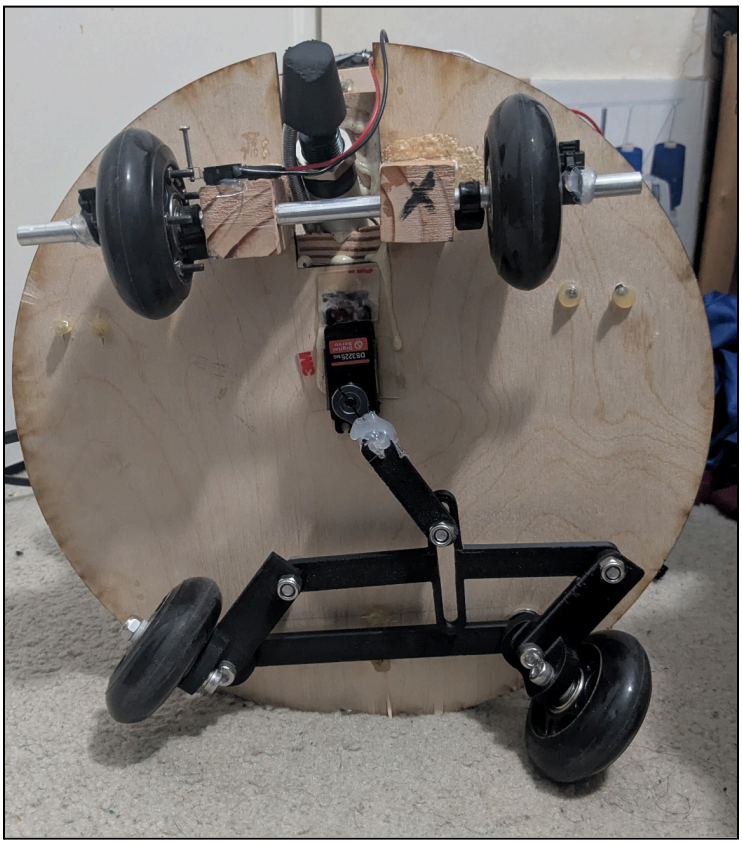

Figure 4: Bottom view of the physical robot, showcasing steering mechanism and rear axle.

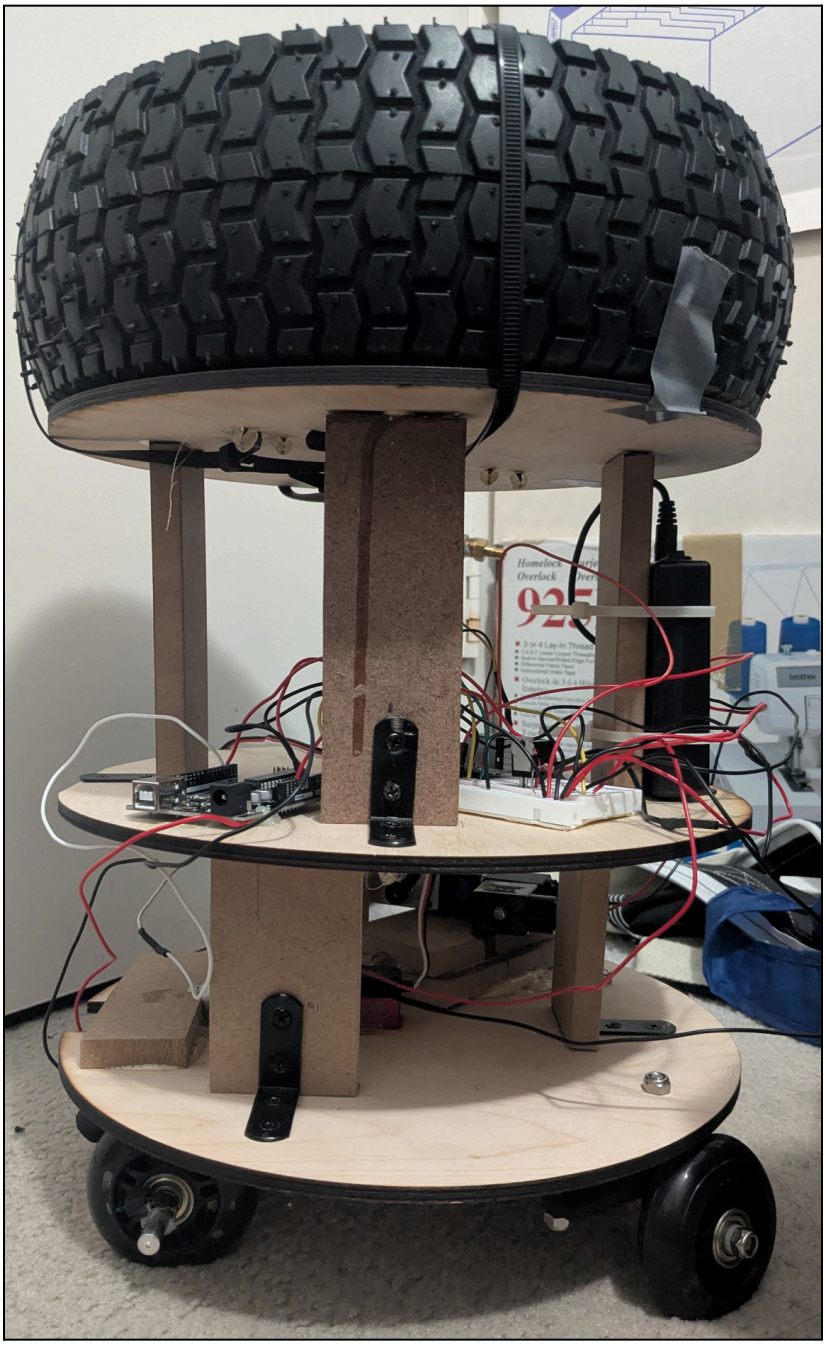

Figure 5: Right view of physical robot, showcasing the electrical components being mounted.

1.3. Final Electrical Circuit Diagram:

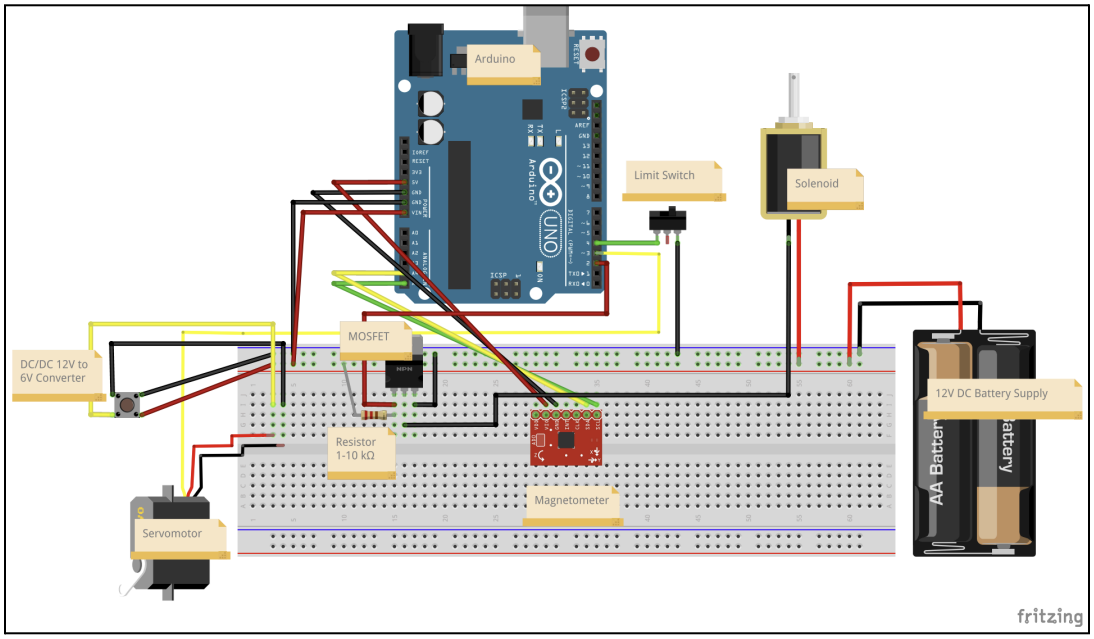

Figure 6

1.4. Software Code

Impulse response code included in Appendix A. Verification 2 code submitted to Verification 2 assignment.

2. Testing and Development

2.1. Experimental testing

2.1.1. Performance Criterion:

We sought to enhance the servo motor's accuracy during straight-line motion by minimizing any lateral deviation from the intended trajectory. We want to move quickly in a straight line, but also make sure we are following that straight line precisely.

2.1.2. Experimental Parameter:

In this investigation, the primary variable was the neutral servo angle, while the servo gain and piston on/off timing were held constant. The neutral servo angle defines the default position of the servo motor and is integral to the steering calculations.

2.1.3. Plot and Data:

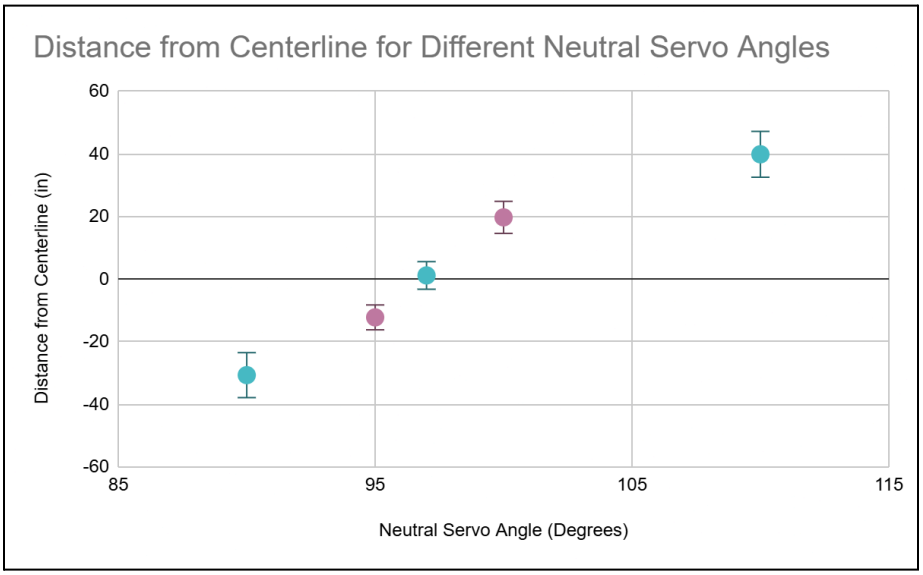

Figure 7

Neutral Servo Angle	Mean	Standard Deviation
90°	-30.65 in	7.20
95°	-12.2 in	3.97

97°	1.2 in	4.41
100°	19.75 in	5.12
110°	39.9 in	7.32

2.1.4. Parameter Optimization Explanation:

To refine and optimize this parameter, a 10-foot reference line was placed on the floor. The robot was placed at the beginning of the line, oriented precisely along the reference. Over a four-second interval, the robot's piston was actuated multiple times, and the lateral displacement from the centerline was measured after. This procedure was repeated 10 times for 5 different neutral servo angle values. The graph shows that as the neutral servo angle value went towards 97 degrees, that the accuracy improved and the standard deviation decreased.

2.2. Comparison with mathematical model

2.2.1. Robot Impulse Response Data:

Impulse response data was collected by running a program which fired the piston a single time. The limit switch was used to record the distance and time between ticks to calculate velocity. The amount of data collected was limited by the resolution of the encoder system, which was five screws installed in the hub (five ticks per revolution).

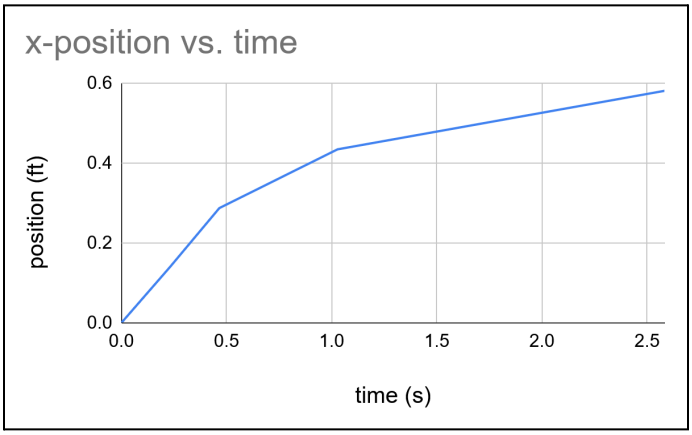

Figure 8

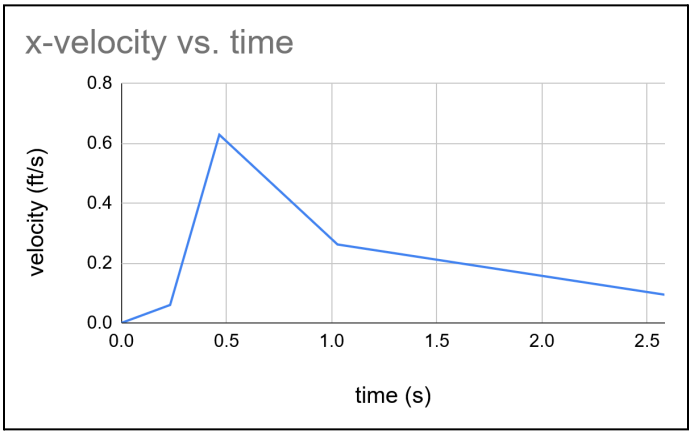

Figure 9

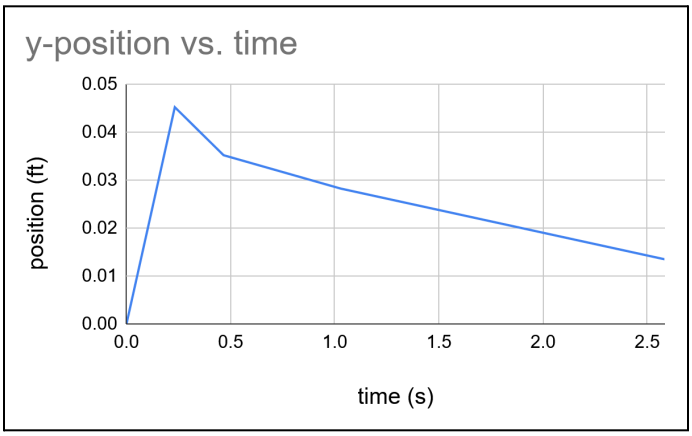

Figure 10

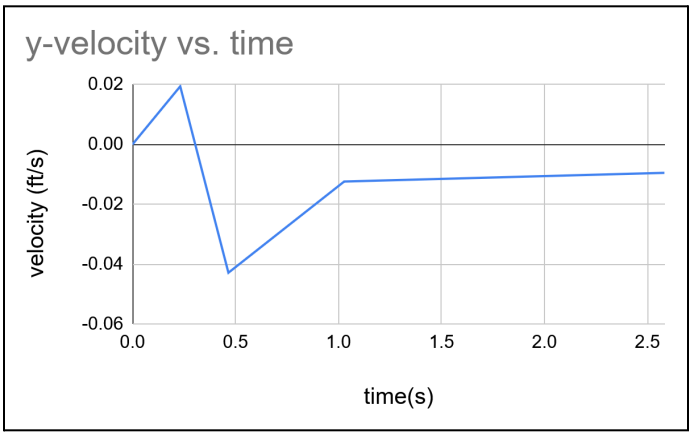

Figure 11

2.2.2. Comparison of Robot's Actual Performance:

The comparison of the 10-impulse case for both the physical robot and simulation are shown below. Data for the physical robot was collected using the impulse response code and tracking position

and velocity at every switch tick. Data for the simulation was collected by running the simulation using the impulse response data for the physical robot and retrieving values stored in the program.

The results are comparable, demonstrating very similar trends in both the position and velocity vs. time plots.

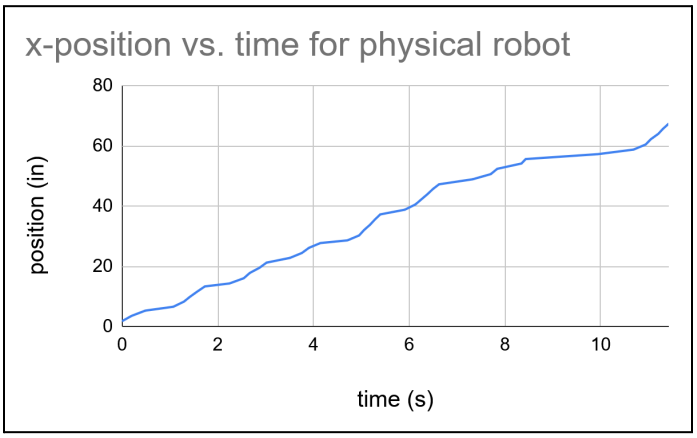

Figure 12

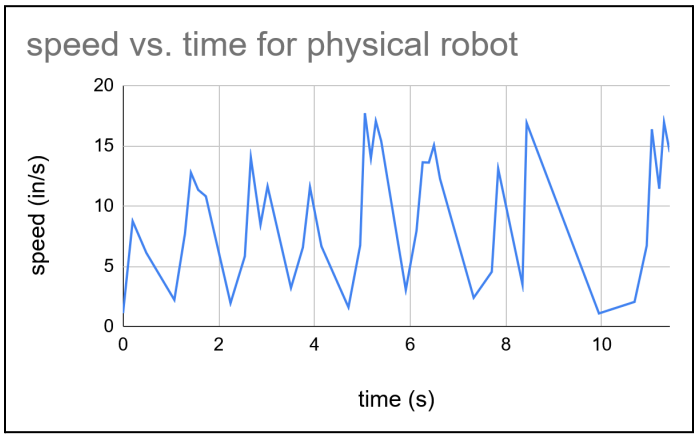

Figure 13

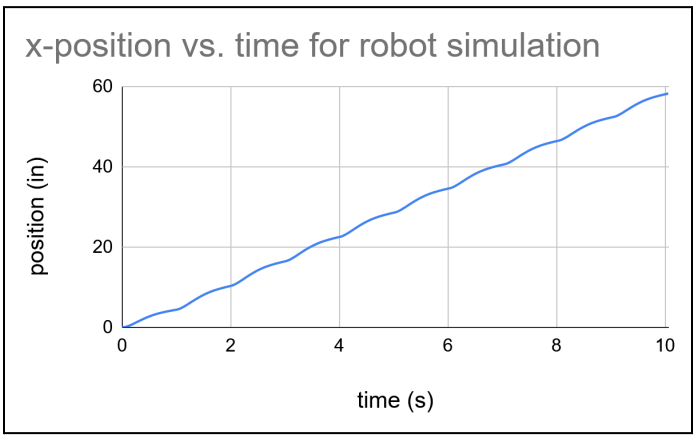

Figure 14

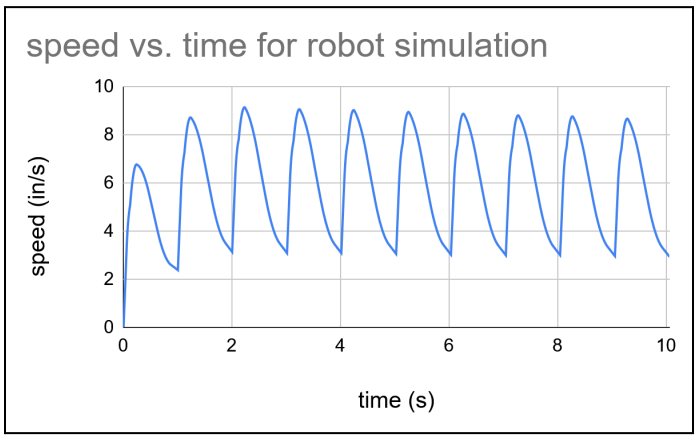

Figure 15

2.2.3. Causes of Discrepancies:

The discrepancy between the speed vs. time plots for the physical robot and the simulation is likely due to the setting in which the experiment was performed. The floor was not even, leading to large

discrepancies in behavior from one impulse to the next.

Other physical disturbances also had a considerable effect on the data. After an impulse, the robot would often tilt due to the impact. Subsequent piston fires would not be as effective in advancing the robot. Heading correction by the servo also had adverse effects. Large corrections would lead to displacement in the wrong direction, and turns sometimes served to further destabilize the robot.

There may also be other factors such as differences in the effects of the pressure decay, limitations of the limit switch resolution, or something as simple as friction.

3. Summary of Contributions

3.1. Team Contribution

Mechanical: Henry Nguyen	• Worked on the initial design and CAD model of the robot • Communicated with other roles to ensure proper integration of electrical system and software • Led the manufacturing process of the robot • Helped with experimental testing
Electrical: Erin Lee	• Created electrical diagram, wiring, and pneumatic system • Performed testing and debugging of electrical components • Assisted with the manufacturing process and initial testing of the robot • Ensured all electrical wiring and components were working prior to verification
Controls: Gavin Fujimoto	• Developed programs for Verification #2, impulse response, hybrid, and waypoint control • Performed parameter calibration with experimental role • Assisted with manufacturing, including the design of the steering system and integration of the limit switch • Performed verification of the integrity of all subcomponents prior to Verification #2
Experimental: Matthew Scott	• Performed calibration of the magnetometer, steering, and piston timing • Contributed heavily to manufacturing, especially in handling unexpected issues • Performed multiple parameter calibrations to enhance speed and accuracy • Helped with the design of the piston-holding structure

3.2. Strategies for Productive Engineering Design Team:

One strategy that was helpful with making sure our design team works well together is to have defined roles and responsibilities. This helped avoid situations where members avoided taking action because they thought a task fell under another role.

Another strategy to continue to have our design team work to be productive is to have clear communication during our work sessions. In tandem with the prior strategy, this eliminated any confusion about what tasks needed to be completed and who was assigned to complete them. Lab time was spent very effectively and problems were addressed promptly.

Appendix A - Impulse Response Program

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

#include <LIS3MDL.h>
#include <LSM6.h>

#define PI 3.14159265358979323846264338327795

LIS3MDL mag;
LSM6 imu;

LIS3MDL::vector<int16_t> m_min = { +2551, +148, -1568 };
LIS3MDL::vector<int16_t> m_max = { +4859, +3089, -952 };

Servo myservo;

int servoPin = 3;           // Pin that the servomotor is connected to
int solenoidPin = 2;        // Pin that the mosfet is connected to
int switchPin = 4;          // Pin that the switch is connected to
bool switchState;           // variable that stores the Reed switch state
bool solenoidState = LOW;   // variable that stores if solenoid is on or
off
int desired_servo = 0;      //desired servo angle
int num_impulses = 0;       //tracks current number of impulses

const unsigned long on_interval = 200;     // interval at which to turn
solenoid on (milliseconds)
const unsigned long off_interval = 2000;   // interval at which to turn
solenoid off (milliseconds)
const float dist = 0.5;                          //acceptable distance to
waypoint

unsigned long switch_ticks = 0;             //times switch is clicked
unsigned long last_switch_state = 0;        //stores last switch state
unsigned long input_servo = 0;              //corrected servo angle
unsigned long previousMillis = 0;                 // will store last time
solenoid was updated
```

```
unsigned long switch_previous_millis = 0;  // will store last time switch
was pressed
unsigned long previous_servo = 90;         //previous angle of the servo


const float distance_per_tick = 8.858 / 12 / 5;  // distance per limit
switch tick (ft) given by the radius of the wheel
const float heading_gamma = 0.6;                    //strength of filter
applied to heading to minimize effect of disturbances
const float servo_gain = 0.4;                        //proportional gain
applied to servo to reduce overshoot

float distance_traveled = 0;  //total distance traveled
float vel_x = 0;              //stores the x-component of velocity
float vel_y = 0;              //stores the y-component of velocity
float  temp_x  =  0;                  //used to store previous value of
x-position to compute x-velocity
float  temp_y  =  0;                  //used to store previous value of
y-position to compute y-velocity

unsigned long prev_heading;  //stores the value of the previous heading
to use for filtering

//struct is used as a remnant of waypoint control system, stores the
current x and y position of the robot
struct Coords {
  float x[3];
  float y[3];
};

Coords bot_coords = { 0, 0 };  //initializes struct for the robot

void setup() {
  delay(1000);                          //initial delay to account for time
to upload code before powering robot
  myservo.attach(servoPin);             // attaches the servo on pin 9 to
the servo object
  pinMode(solenoidPin, OUTPUT);      //Sets the pin as an output
  pinMode(switchPin, INPUT_PULLUP);  //Sets the pin as an input_pullup
```

```
  Serial.begin(9600);                    // starts serial communication @
9600 bps
  Wire.begin();

  if (!mag.init()) {
          Serial.println("Failed   to   detect   and   initialize   LIS3MDL
magnetometer!");
    while (1)
      ;
  }
  mag.enableDefault();

  if (!imu.init()) {
    Serial.println("Failed to detect and initialize LSM6 IMU!");
    while (1)
      ;
  }
  imu.enableDefault();
}



void loop() {
  mag.read();
  imu.read();
    float  init_heading  =  computeHeading();    //initializes  an  initial
heading to serve as the desired direction for the robot to travel
   prev_heading  =  init_heading;                //initializes the previous
heading  as  the  initial  heading  value  to  prevent  a  sudden  directional
change upon start

  while (num_impulses < 10) {  //loops until desired number of impulses
    mag.read();
    imu.read();

    float heading = computeHeading();  //updates heading every loop

    unsigned long currentMillis = millis();  //updates timer every loop
```

```
    //the following code block adjusts for the fact that the heading will
overflow from 360 to 0 and vice versa
     //the value of heading is adjusted to be locally continuous for a
range of values based on the initial heading
    //i.e. the heading won't suddenly change except for the case in which
the robot is 180 degrees from initial heading
    if (init_heading >= 180 && heading <= init_heading - 180) {
      heading = heading + 360;
    } else if (init_heading < 180 && heading >= init_heading + 180) {
      heading = heading - 360;
    }

    //digital filter applied to heading
    //based on value of gamma, adjusts current heading by some proportion
of the previous heading and updates previous heading
        heading = (1 - heading_gamma) * heading + heading_gamma *
prev_heading;
    prev_heading = heading;

     //defines desired servo angle as the difference between the initial
and current heading
    desired_servo = init_heading - heading;

    //correction for max servo angle, angles beyond 45 degrees in either
direction are capped
    if (desired_servo <= -45) {
      desired_servo = -45;
    } else if (desired_servo >= 45) {
      desired_servo = 45;
    }

    //uses the desired_servo input and adjusts it by the gain
     //the constant is added to convert the actual servo angle (from -90
to 90 deg) to the servo input (0 to 180 deg)
    //and is offset by an experimental amount so that the robot naturally
centers itself
    input_servo = -desired_servo * servo_gain + 100;

    //writes the resulting value to the servo
    myservo.write(input_servo);
```

```
    //code block to control solenoid
     //if solenoid is on and the on interval is exceeded, the solenoid is
turned off
      //every time the solenoid is turned off the number of impulses is
incremented
    if (solenoidState && currentMillis - previousMillis >= on_interval) {
      previousMillis = currentMillis;            //update last timer
      solenoidState = !solenoidState;            //invert state
      digitalWrite(solenoidPin, solenoidState);  //Switch Solenoid OFF
      num_impulses++;
    }

      //if the solenoid is off and the off interval is exceeded, the
solenoid is turned on
    if (!solenoidState && currentMillis - previousMillis >= off_interval)
{
      previousMillis = currentMillis;
      solenoidState = !solenoidState;
      digitalWrite(solenoidPin, solenoidState);  //Switch Solenoid ON;
    }

    //reads the value of the limit switch
    switchState = digitalRead(switchPin);

    //increments the number of ticks, the distance travelled, and updates
the position and velocity of the robot
     //calculates velocity by updating a timer which keeps track of the
time since the last limit switch click
     //only activates if the switch is pressed and was not pressed the
last cycle of the loop
    if (switchState && last_switch_state == 0) {
      distance_traveled = distance_traveled + distance_per_tick;
      switch_ticks++;
      //stores previous x and y position to calculate velocity
      temp_x = bot_coords.x[0];
      temp_y = bot_coords.y[0];
       //uses trigonometry to update the position of the robot based on
the current heading relative to the initial heading
```

```
            bot_coords.x[0]  =  bot_coords.x[0]  +  distance_per_tick  *
cos((heading - init_heading) * PI / 180.0);
            bot_coords.y[0]  =  bot_coords.y[0]  +  distance_per_tick  *
sin((heading - init_heading) * PI / 180.0);

      //calculates the difference in time since the last time the switch
was pressed and convert to seconds
         float  time_diff  =  (currentMillis  -  switch_previous_millis)  /
1000.0;

      //uses velocity equation v = dx/dt to calculate velocity
      vel_x = (bot_coords.x[0] - temp_x) / time_diff;  //ft/s
      vel_y = (bot_coords.y[0] - temp_y) / time_diff;
      switch_previous_millis = currentMillis;  //updates time

      //prints to serial monitor every time the switch is pressed
      Serial.print(" current time: ");
      Serial.print(currentMillis);
      Serial.print(" Current x: ");
      Serial.print(bot_coords.x[0], 6);
      Serial.print(" Current y: ");
      Serial.print(bot_coords.y[0], 6);
      Serial.print(" vel x: ");
      Serial.print(vel_x, 6);
      Serial.print(" vel y: ");
      Serial.println(vel_y, 6);
    }
     last_switch_state = switchState;  //used to determine if switch was
released before being pressed
  }
}

//heading function for magnetometer

template<typename T> float computeHeading(LIS3MDL::vector<T> from) {
  LIS3MDL::vector<int32_t> temp_m = { mag.m.x, mag.m.y, mag.m.z };

  // copy acceleration readings from LSM6::vector into an LIS3MDL::vector
  LIS3MDL::vector<int16_t> a = { imu.a.x, imu.a.y, imu.a.z };
```

```
  // subtract offset (average of min and max) from magnetometer readings
  temp_m.x -= ((int32_t)m_min.x + m_max.x) / 2;
  temp_m.y -= ((int32_t)m_min.y + m_max.y) / 2;
  temp_m.z -= ((int32_t)m_min.z + m_max.z) / 2;

  // compute E and N
  LIS3MDL::vector<float> E;
  LIS3MDL::vector<float> N;
  LIS3MDL::vector_cross(&temp_m, &a, &E);
  LIS3MDL::vector_normalize(&E);
  LIS3MDL::vector_cross(&a, &E, &N);
  LIS3MDL::vector_normalize(&N);

  // compute heading
        float    heading    =    atan2(LIS3MDL::vector_dot(&E,    &from),
LIS3MDL::vector_dot(&N, &from)) * 180 / PI;
  if (heading < 0) heading += 360;
  return heading;
}

/*
Returns the angular difference in the horizontal plane between a
default vector (the +X axis) and north, in degrees.
*/
float computeHeading() {
  return computeHeading((LIS3MDL::vector<int>){ 1, 0, 0 });
}
```