Electric Vehicle

Reflections from 24-25 Season by Min: EMAIL ms2602@iolani.org OR rre2601@iolani.org

Vehicle Design

- Structure of the vehicle:

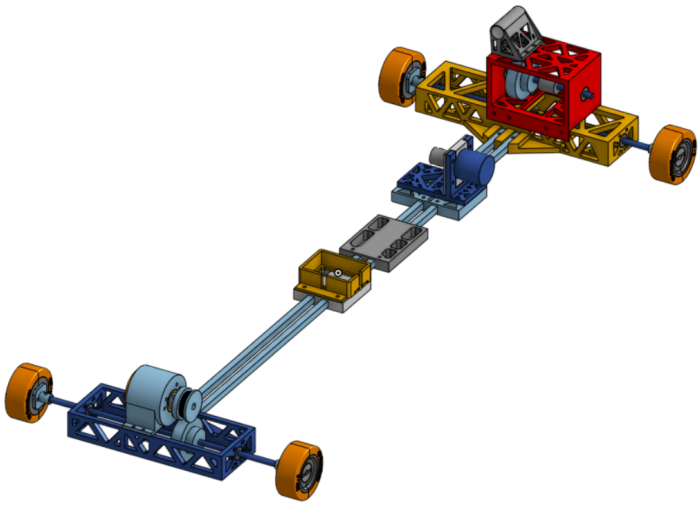


Figure 1: OnShape assembly of the electric vehicle

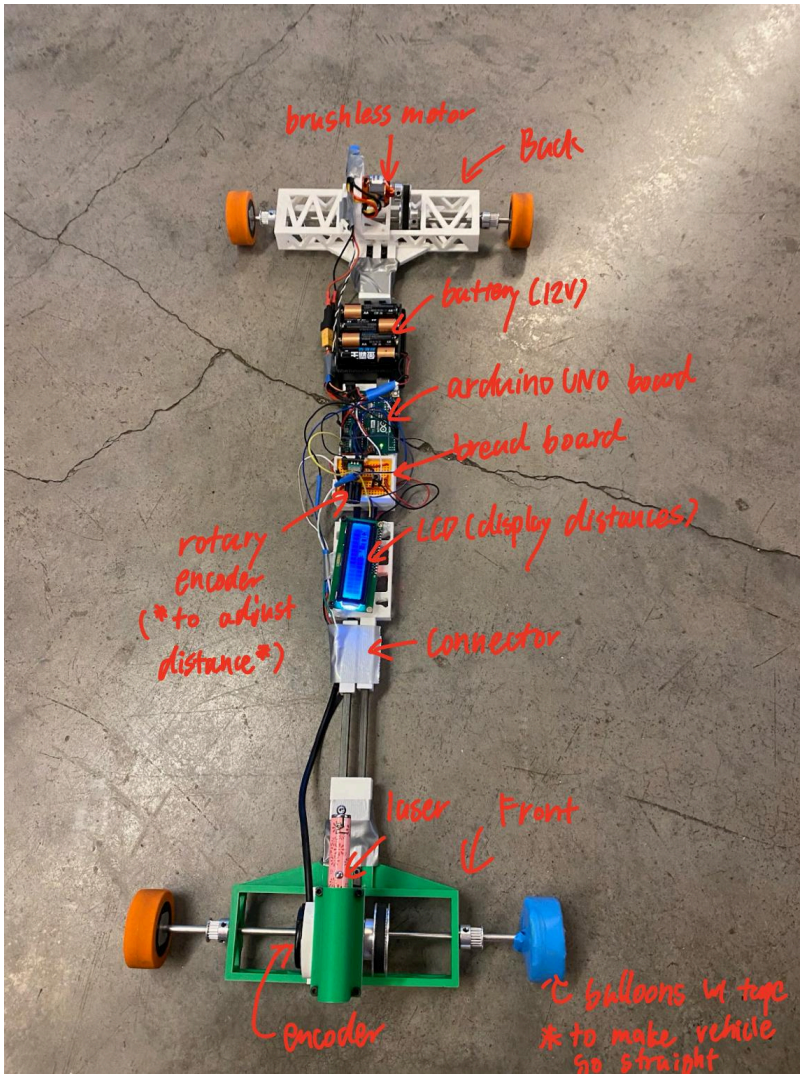
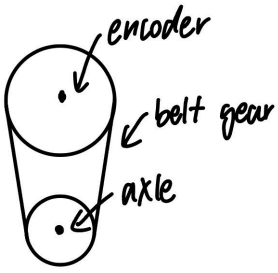
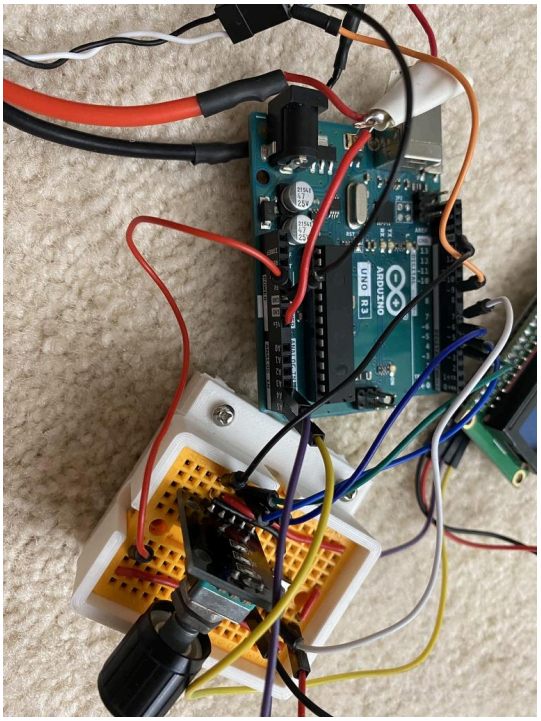


Figure 2: Fully Constructed Electric Vehicle II before 2025 Nationals

- carbon fibers rods X4 connected with connector at the middle (so technically two carbon fiber rods support the vehicle)
  - **carbon fibers are necessary to reduce the weight of the vehicle significantly**
- front chassis has encoder connected with belt gears (axle spins => belt gear => encoder spins) **refer to the simple diagram below**



    - Balloons added to one of the front wheels (obviously, it prevents the vehicle from veering off at the end)
    - Lasers attached at the front and used for the alignment of the vehicle
- Hardware of the robot:
    - arduino UNO board
    - breadboard to build circuit
    - **IMPORTANT**: how to set distance the vehicle travels (refer the figure below)
        - LCD display displays the distance the vehicle will travel, and you can adjust the distance using the rotary encoder
        - Press the rotary encoder to set the distance
        - Press the button on the breadboard to activate the vehicle

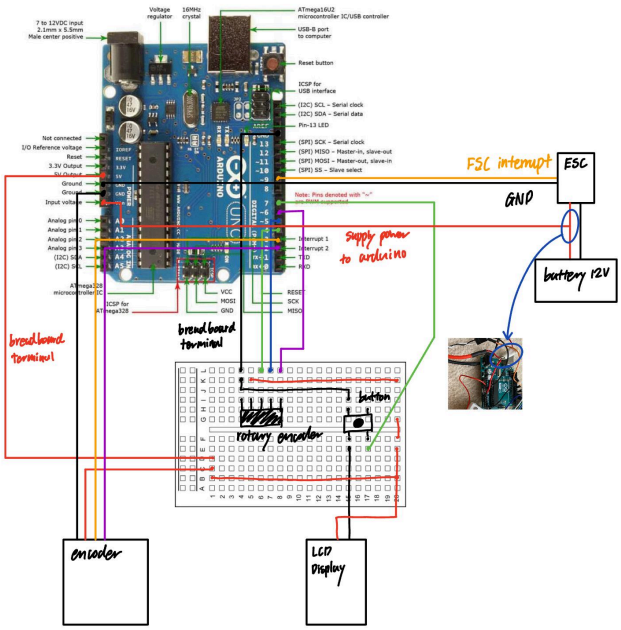


Brushless Motor and ESC Experimentation

- Motor Selection:
    - Experimented with motors at different kV ratings to optimize the balance between RPM (speed) and torque
    - Lower kV motors provide higher torque but lower top speed; higher kV motors provide higher RPM but lower torque
- Key Challenge: Startup Torque vs. Current Draw
    - Certain motors would stall or fail to initiate movement if the static friction and initial load required too much torque
    - High-torque motors (lower kV) could overcome static friction but drew excessive current during startup

    - If current draw exceeded the ESC's rating, it would either cut power to protect itself or cause the motor to overheat and potentially burn out
- ESC (Electronic Speed Controller) Considerations:
    - ESCs have specific continuous current and burst current ratings (measured in amps)
    - Had to match ESC current capacity to the motor's startup current requirements
    - Underpowered ESCs would shut down during acceleration; overpowered ESCs added unnecessary weight
- Systematic Testing Process:
    - Tested multiple motor kV ratings (e.g., 1000 kV, 1250 kV, 1500 kV) with different ESC current ratings (e.g., 40A, 80A, and more)
    - Measured actual current draw during startup using multimeter or ESC telemetry
    - Monitored motor temperature after test runs to identify thermal stress
- We ultimately found optimal motor-ESC pairing that balanced
    - **Flash Hobby Brushless Motor (1250 kV rating) with 40A ESC**

Circuit Diagram



Code

```
#include <Servo.h>
#include <LiquidCrystal_I2C.h>
#include <Encoder.h>
#include <Arduino.h>

const int N = 50;
double table1Y1[N+1], table1Y2[N+1];
double table2Y1[N+1], table2Y2[N+1];

// parameters for table1 (original f1/f2) //for 830 cm
double a1 = 1.0, b1 = 1.0, c1 = 1.5;
// parameters for table2 (kx - ax^3/6 etc.)
double a2 = 1.0, b2 = 1.0, c2 = 1.0;

// Precompute both tables once in setup()
void precomputeTables() {
  // range [0, 2*b1 + c1]
  double s1 = 0, e1 = 2*b1 + c1, d1 = (e1 - s1) / N;
  for (int i = 0; i <= N; ++i) {
    double x = s1 + i * d1;
    double y1 = 0, y2 = 0;
    // table1 functions
    if (x > 0 && x < b1) {
      y1 = (a1 * pow(x, 3)) / 6.0;
      y2 = (a1 * pow(x, 2)) / 2.0;
    } else if (x > b1 && x < b1 + c1) {
      y1 = (a1 * pow(x - b1,2) * b1) / 2.0
         + (a1 * pow(b1,2) / 2.0) * (x - b1)
         + (a1 * pow(b1,3)) / 6.0;
      y2 = (a1 * b1 * x) - (a1 * pow(b1,2)) / 2.0;
    } else if (x > b1 + c1 && x < 2*b1 + c1) {
      y1 = (a1 * pow(b1,3)) / 6.0
         + (a1 * b1 * pow(c1,2)) / 2.0
         + (a1 * pow(b1,2) * c1) / 2.0
         + (a1 * b1 * pow(x - b1 - c1,2)) / 2.0
         - (a1 * pow(x - b1 - c1,3)) / 6.0
         + (a1 * pow(b1,2) / 2.0) * (x - b1 - c1)
         + (a1 * b1 * c1) * (x - b1 - c1);
      y2 = ((a1 * b1) - (a1 * (x - b1 - c1) / 2.0)) * (x - b1 - c1)
         + (a1 * pow(b1,2)) / 2.0
```

```
        + (a1 * b1 * c1);
    }
    table1Y1[i] = y1;
    table1Y2[i] = y2;
  }
  // range [0, 2*b2 + c2]
  double s2 = 0, e2 = 2*b2 + c2, d2 = (e2 - s2) / N;
  for (int i = 0; i <= N; ++i) {
    double x = s2 + i * d2;
    double y1 = 0, y2 = 0;
    // table2 functions
    if (x > 0 && x < b2) {
      y1 = (a2*b2+a2*b2*b2*c2) * x - (a2 * pow(x,3)) / 6.0;
      y2 = (a2 / 2.0) * (b2 * (b2 + 2 * c2) - pow(x,2));
    } else if (x > b2 && x < b2 + c2) {
      y1 = (5.0/6.0) * a2 * pow(b2,3)
         + a2 * pow(b2,2) * c2
         + a2 * b2 * (c2 + b2/2.0) * (x - b2)
         - (a2 * b2 / 2.0) * pow(x - b2,2);
      y2 = (a2 * b2 / 2.0) * (3 * b2 + 2 * c2 - 2 * x);
    } else if (x > b2 + c2 && x < 2*b2 + c2) {
      y1 = (a2 * b2 * (5*pow(b2,2) + 9 * b2 * c2 + 3*pow(c2,2))) / 6.0
         + (a2 * pow(b2,2) / 2.0) * (x - b2 - c2)
         - (a2 / 2.0) * pow(x - b2 - c2,2)
         + (a2 / 6.0) * pow(x - b2 - c2,3);
      y2 = (a2 / 2.0) * (pow(b2,2)
         + 2 * b2 * (b2 + c2 - x)
         + pow(b2 + c2 - x,2));
    }
    table2Y1[i] = y1;
    table2Y2[i] = y2;
  }
}

// find nearest index in an array of f1-values
int findNearest(double input, const double arr[]) {
  int idx = 0;
  double minDiff = 1e18;
  for (int i = 0; i <= N; ++i) {
    double d = abs(arr[i] - input);
    if (d < minDiff) {
      minDiff = d;
```

```
      idx = i;
    }
  }
  return idx;
}


// ---------- PIN ASSIGNMENTS ----------
// User rotary encoder (for setting distance)
#define ENC1_PIN_A 4
#define ENC1_PIN_B 5
#define ENC1_BUTTON_PIN 6

// Start button (pull-down resistor)
#define BUTTON_PIN 7

// ESC signal pin (controls brushless DC motor)
#define ESC_PIN 9

// ---------- GLOBAL OBJECTS & VARIABLES ----------
LiquidCrystal_I2C lcd(0x27, 16, 2);
Encoder setEnc(ENC1_PIN_A, ENC1_PIN_B);
Servo esc;  // Create ESC object


long oldPosition = 0;        // Tracks the last known encoder "raw"
position
int  distanceValue = 700;    // The user-selected integer (700–1000)
bool selectionStarted = false;   // True after first click
bool distanceConfirmed = false;  // True after second click

/* Encoder control */
volatile long counter_LEFT = 0;   // 32-bit signed
volatile bool leftEncoderChanged = false;
const int TARGET_COUNT = 10000;  // Stop motor at this count
bool motorRunning = false;       // Track motor state
void ai0() {
  if (digitalRead(3) == LOW) {
    counter_LEFT--;
  } else {
    counter_LEFT++;
  }
```

```
  leftEncoderChanged = true;
}
void ai1() {
  if (digitalRead(2) == LOW) {
    counter_LEFT++;
  } else {
    counter_LEFT--;
  }
  leftEncoderChanged = true;
}



// Setup states
enum State { SET_DISTANCE, WAIT_START, RUNNING, STOPPED };
State state = SET_DISTANCE;
// Distance variables (in cm)
long targetDistance = 100;   // default 100 cm
long lastSetPos = 0;
bool buttonPressed = false;


void setup() {

  precomputeTables();

  lcd.init();
  lcd.begin(16,2);
  lcd.backlight();
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("KILL ME");
  Serial.begin(9600);
  Serial.println("hello cruel world");
  pinMode(2, INPUT_PULLUP);
  pinMode(3, INPUT_PULLUP);
  pinMode(BUTTON_PIN, INPUT_PULLUP);  // Enable internal pull-up for
button

  // Setting up interrupts for encoders
  attachInterrupt(digitalPinToInterrupt(2), ai0, RISING);
  attachInterrupt(digitalPinToInterrupt(3), ai1, RISING);
```

```
  pinMode(ENC1_BUTTON_PIN, INPUT);

  // Setting up ESC
  esc.attach(9);  // Attach ESC to pin 9
  esc.writeMicroseconds(1000); // minimum throttle (motor off)
  Serial.println("Calibrating ESC...");
  esc.writeMicroseconds(2000);  // Max throttle signal
  delay(2000);
  esc.writeMicroseconds(1000);  // Min throttle signal
  delay(2000);
  Serial.println("ESC Ready!");

}

void loop() {

 switch (state) {
    case SET_DISTANCE:
      handleSetDistance();
      break;
    case WAIT_START:
      handleWaitStart();
      break;
    case RUNNING:
      handleRunning();
      break;
    case STOPPED:
      // Do nothing or show stopped
      lcd.setCursor(0,0);
      lcd.print("-- Stopped --    ");
      break;
  }
}


// ---------- STATE HANDLERS ----------

void handleSetDistance() {
  readRotary(); // updates distanceValue as user turns the encoder

    // If the user clicks the encoder button, confirm the distance
```

```
    if (encoderButtonPressed()) {
      lcd.clear();
      lcd.setCursor(0, 0);
      lcd.print("Distance set to:");
      lcd.setCursor(0, 1);
      lcd.print(distanceValue);
      delay(2000);

      lcd.clear();
      lcd.setCursor(0, 0);
      lcd.print("Press btn to");
      lcd.setCursor(0, 1);
      lcd.print("start motor");
      state= WAIT_START;
    }
}


void handleWaitStart() {
   if (digitalRead(BUTTON_PIN) == LOW) {
    counter_LEFT = 0;              // start distance from zero
    leftEncoderChanged = false;
    // Begin motion
    esc.writeMicroseconds(1100); // adjust throttle as needed
    state = RUNNING;
    lcd.clear();
  }
}

void handleRunning() {

  double stopdist = table2Y1[0];`
  for (int i = 1; i < N; i++) if (table2Y1[i] > stopdist) stopdist =
table2Y1[i];
  lcd.setCursor(0, 1);
  lcd.print(distanceValue,1);
  // Read current tick count from the axle encoder
  if (leftEncoderChanged) {

    leftEncoderChanged = false;
    float factor = 0.97857143;
```

```
    // Convert ticks to distance
    float distCm =
0.887*2.21518987342*1.02142857143*1.94174757282*1.07285714286*abs(counter_
LEFT)/7.5191311697/8.23529411765/0.86206896551*factor*factor*1.00714286;
// note 100.0 forces floating-point division


    // Update LCD
    lcd.setCursor(0, 0);
    lcd.print("Trav: ");
    lcd.print(distCm, 1);
    lcd.print(" cm   ");
    if (distCm >= distanceValue) {
    esc.writeMicroseconds(1000); // Stop the motor
    state = STOPPED;
    }
    else if (distCm >= distanceValue-stopdist*100-100) {
      if (distCm >= distanceValue-stopdist-50) {
        esc.writeMicroseconds(1020);
      }
      double brakeCm = distCm-(distanceValue-stopdist*100-100);
      int i1 = findNearest(brakeCm/100, table1Y1);
        esc.writeMicroseconds((int)(1050 + (constrain(table2Y2[i1], 0.0,
2.0) * 150)));
    } else {
      int i1 = findNearest(distCm/100, table1Y1);
      esc.writeMicroseconds((int)(1050 + (constrain(table1Y2[i1], 0.0,
2.0) * 150)));
    }
  }

}


//
------------------------------------------------------------------------
-
// readRotary: uses the Encoder library to handle rotation logic
//
------------------------------------------------------------------------
-
void readRotary() {
```

```
  long newPosition = setEnc.read();

  // Only act if the position changed AND we've reached a "detent"
boundary
  // (i.e. newPosition is a multiple of 4).
  if (newPosition != oldPosition && (newPosition % 4 == 0)) {
    oldPosition = newPosition;

    // Map from raw encoder steps to [700..1000] range:
    // 1 "detent" = newPosition changes by +/-4, so newPosition/4
increments by 1.
    // We'll treat "0" steps as distance=700. So newPosition/4 = 0 =>
distance=700.
    // if newPosition/4 = 300 => distance=1000 (that's a difference of 300
from 700).
    int candidate = 700 + (newPosition / 4);

    // Constrain to 700..1000
    if (candidate < 700) {
      candidate = 700;
      setEnc.write(0); // keep the encoder in sync
      oldPosition = 0;
    }
    else if (candidate > 1000) {
      candidate = 1000;
      long maxSteps = (1000 - 700) * 4; // 300*4=1200
      setEnc.write(maxSteps);
      oldPosition = maxSteps;
    }

    // Update the global distance and the LCD
    distanceValue = candidate;
    showDistanceOnLCD();
  }
}

//
--------------------------------------------------------------------------
-
// encoderButtonPressed: Debounced check for the rotary encoder's push
button
```

```
//
------------------------------------------------------------------------
-
bool encoderButtonPressed() {
  static bool lastState = HIGH;
  bool currentState = digitalRead(ENC1_BUTTON_PIN);

  // Check for a falling edge (HIGH->LOW) with a short debounce
  if (lastState == HIGH && currentState == LOW) {
    delay(50);
    if (digitalRead(ENC1_BUTTON_PIN) == LOW) {
      lastState = LOW;
      return true;
    }
  }
  else if (currentState == HIGH) {
    lastState = HIGH;
  }
  return false;
}

//
------------------------------------------------------------------------
-
// showDistanceOnLCD: Helper to display the current distanceValue on the
LCD
//
------------------------------------------------------------------------
-
void showDistanceOnLCD() {
  lcd.setCursor(0, 1);
  lcd.print("     ");         // Clear old content
  lcd.setCursor(0, 1);
  lcd.print(distanceValue);
}
```

Notes for future reference!

Encoder Calibration and Distance Accuracy

- Core Problem: Encoder Counts vs. Actual Distance
    - Encoder counts are theoretically accurate, but actual distance traveled deviates from calculated distance
    - Main cause: Motor doesn't brake precisely at the target point because momentum causes overshoot or undershoot
    - Solutions were either to activate the manual motor break system, which may wear down the motor over time or apply a calibration factor to adjust target encoder counts

Calibration Factor Calculation

- Factor formula: Target Distance ÷ Actual Distance Traveled
- Example:
    - Set distance: 750 cm
    - Actual distance: 760 cm (overshot by 10 cm)
    - Calibration factor: 750 ÷ 760 = 0.987
    - Implementation: Multiply this factor by target encoder counts in Arduino code to correct for overshoot

Battery Management for Accurate Calibration

- Critical: Use fresh batteries, but NOT brand new ones
- Recommended calibration window: Run 3-4 test runs after switching batteries, then calibrate between runs 4-10
- Rationale:
    - Fully new batteries provide excessive motor power, causing inconsistent behavior
    - After ~10 runs, batteries begin draining significantly, reducing motor performance
    - Runs 4-10 represent the "stable operating range" where battery voltage is consistent
    - Minimizes confounding variables from battery charge fluctuations

End-of-Season Reflection

- Achievement: Successfully solved straight-line navigation! The vehicle travels straight reliably!
- Limitation: Target encoder counts in code don't precisely reflect actual distance traveled, though calibration factors compensate adequately
- Room for improvement: could have refined encoder-to-distance mapping for more intuitive code (e.g., 1000 counts = exactly 100 cm)

Challenge for Next Season: Obstacle Navigation

- New requirement: Vehicle must curve around obstacles and reach target point
- Reference: Similar to Scrambler C event from 2024 season
- Curving mechanics not yet explored…

2025-26 Tentative Season Plans (** please write all the plans below**)

By Min Shin (written during 2025 summer)

Path Calculator:

https://www.desmos.com/calculator/gnwcfcgapf

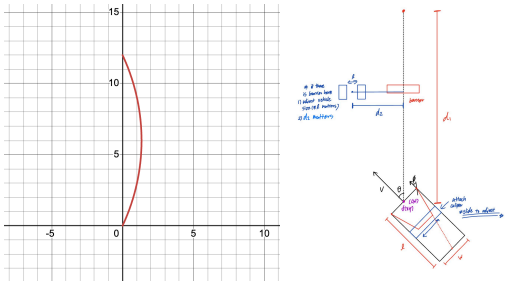
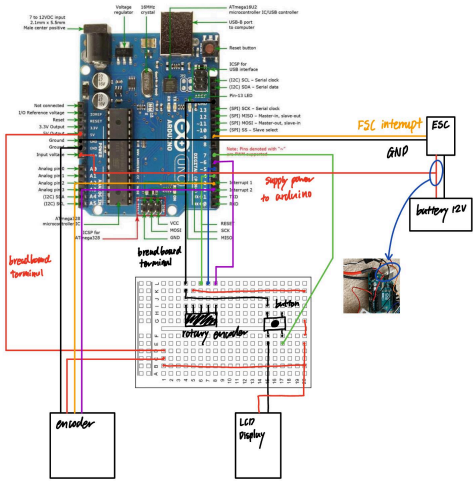


Figure 1: path sketched by Desmos and set-up diagram of vehicle with variables labeled
very theoretical; calibrations have to be done regardless

Circuit: identical to 2024-25 season circuit (works very well)



Use multiple arduino boards (one of the options for now)

Design:

****EMAIL ME to get access to CAD files****

Wheels:

- Use thicker and larger wheels for the rear axle and thinner and smaller wheels for front axle
    - Thickness changes the susceptibility of wheels to steering (thinner wheels are better for steering, while larger wheels are better for controlling/preventing the drift)
- Please figure out which dimension of wheels we are going to use (search for some in the cabinet or on the shelf? buy some if necessary)
    - Probably T61 and T81 banebot wheels (T61 should have slightly smaller diameter and thickness) (minimize the difference in diameter though; probably at most 1 inch?) (we already have T81 wheels from 2025 nationals)
    - 3-⅞" x 0.8" for rear wheels? 2-⅞" x 0.6" for front wheels? (find appropriate shaft size) (check size of the metal rods as well)

Anti-Sway Bar:

- For stability when turning
- It **reduces roll** (side-to-side tilting) when the vehicle turns
- When the vehicle leans (like in a turn), the suspension compresses more on one side — the sway bar resists this difference
- It **increases lateral stability**, keeps more tire contact with the ground, and improves steering response

Encoder:

- Use the identical encoder but attach it to rear axle rather than front axle (errors are more likely for the front axle)

Front Axle:

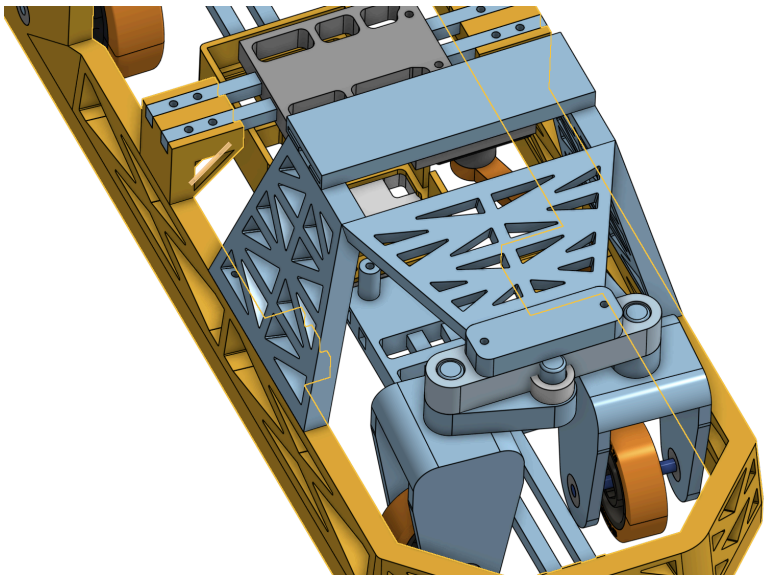
- Use anckermann geometry (please refer to the CAD files)

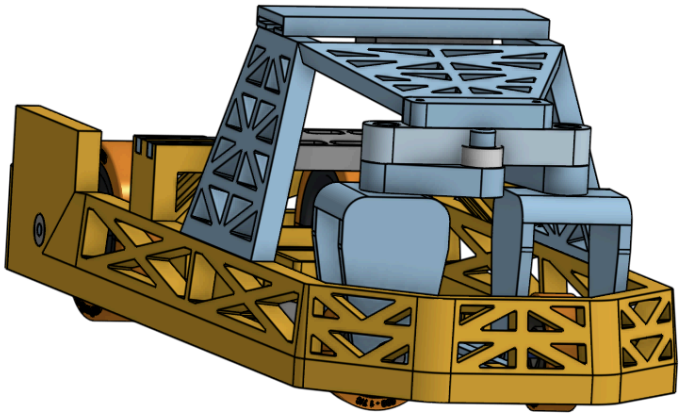
Calibrations due to External Factors (e.g., weight, friction)

- Try to concentrate weight on the rear axle (basically put all the components clustered together near the rear axle)
- Refer to the table below to adjust oversteering and understeering

Adjustments	To Increase Understeer	To Increase Oversteer
Front Tire Section	Smaller	Larger
Rear Tire Section	Larger	Smaller
Weight Distribution	More Forward	More Rearward

CAD Design:





The design uses **Ackermann steering**; although the assembly is slightly mismatched, sliding the caliper sideways allows the steering angle to vary. **Carbon fiber bars** are attached vertically and horizontally to reinforce the vehicle's structural integrity. Additionally, the 3D-printed parts incorporate **"lighten" features** to reduce weight without compromising strength.