Overhead Obstacle Detection Device for Visually Impaired Individuals

Group 6: Dylan Lee, Shanon Lee, Yarah Feteih, Emily Blum

Opportunity

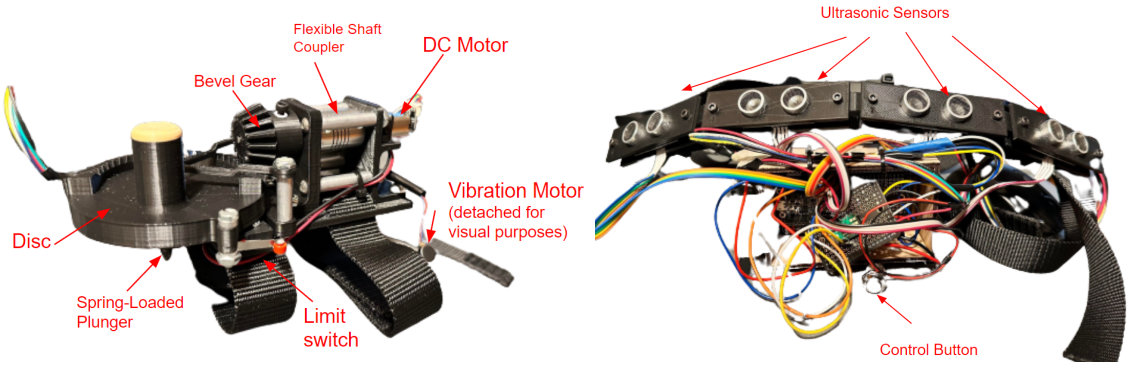
While white canes allow visually impaired individuals to navigate past ground-level obstacles in front of them, they do not account for overhead obstacles. We designed an assistive wearable device that detects overhead obstacles and relays information about their direction and proximity through haptic feedback.

High Level Strategy

Our obstacle detection system includes two separate devices – a sensor array that detects overheard objects and a haptic feedback device that conveys the location of obstacles to the user. The sensor array contains 4 ultrasonic sensors with a combined field of view of 90 degrees. The sensor data is processed on an ESP32 microcontroller to calculate the angle and distance of the obstacle relative to the user. This location information is conveyed to the user through a wrist-mounted haptic device which contains a probe to specify direction and a vibration motor to specify proximity.

Our initial plan was to directly modify the white cane to detect low obstacles missed during the sweeping motion. We decided to pivot towards a separate feedback device to avoid adding weight to the white cane that could make the swinging motion difficult.

Integrated Physical Design

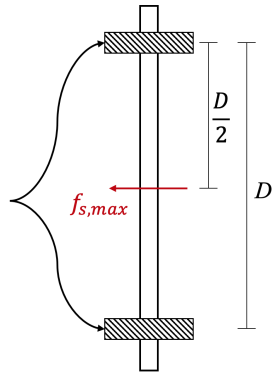


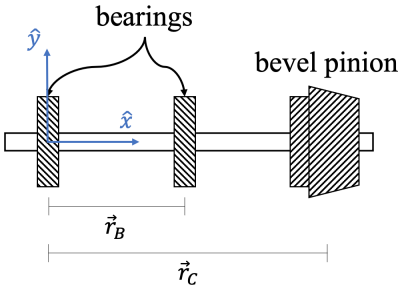
Critical Design Decisions

The two main function-critical decisions to be made were the selection of the DC motor and selection of bearings, both of which are part of the main bevel gear transmission system. The main load on the transmission system is friction between the plunger and the user's skin, whose magnitude is in the worst case the maximum static friction given by $f_{s,max} = \mu_s n$ where μ_s is the coefficient of static friction between PLA and skin, and n is the normal force between the plunger and the skin. The normal force is equal to the worst-case spring restoring force: $n = k\delta$, where k is the spring constant and δ is defined as the worst-case spring displacement. In the worst-case scenario, then $f_{s,max} = \mu_s k\delta$. The torque required to drive the disc is then $\tau_D = \vec{R} \times \vec{f}_{s,max} = \mu_s k\delta R$, where $\vec{R}$ is the radius of the path that the plunger follows. The torque τ_p required to drive the bevel pinion (i.e. the required torque of the motor) is then $\tau_p = \frac{\mu_s k\delta R}{R_G}$, where $\vec{R}_G$ is the gear ratio between the bevel pinion and the disc. Research showed that the coefficient of

static friction between skin and 3-D printed PLA with 20% infill is approximately $\mu_s = 0.37$. The selected spring has a spring constant of $k = 2$ lbf·in^{-1}, with a designed maximum deflection of $\delta = 0.25$ in. The path radius is designed to be $R = 0.75$ in, and the gear ratio is $R_G = 3$. The required motor torque is then calculated to be 0.0463 lbf·in. The 75:1 Micro Metal Gearmotor from Pololu provided in the UC Berkeley Microkits can provide up to 0.152 lbf·in of continuous torque, giving a satisfactory load factor of $\frac{0.152}{0.0463} = 3.29$.

There are two shafts on which the bearing loads must be determined. The first is the shaft on which the disc is mounted. The primary force on the shaft mounting the disc results again from the friction between the plunger and the user's skin. The disc is mounted at the midpoint of two identical bearings a distance D apart (diagrammed on the right). Performing a moment balance about one of the two bearings gives $\frac{D}{2}\hat{x} \times \vec{f}_{s,max} + D\hat{x} \times \vec{F}_{bearing} = 0$. When dotting with $\hat{x}$, bearings $\hat{y}$, and $\hat{z}$, the only nontrivial equation comes from dotting it with $\hat{z}$: $\frac{D}{2} f_{s,max} + D F_{bearing} = 0$, and so $F_{bearing} = \frac{f_{s,max}}{2} = \frac{\mu_s k \delta}{2}$ which, when plugging in the values listed above, gives a bearing force of 0.0925 lbf, which is only a fraction of the 155 lbf rating of the selected bushings.



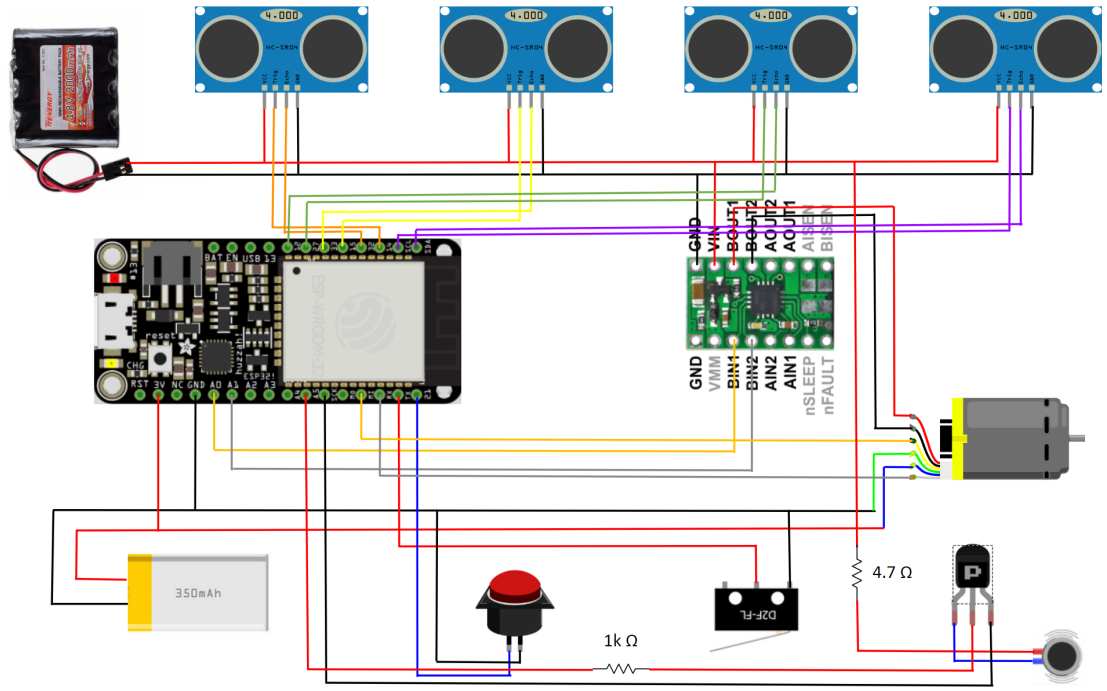


The second shaft is the pinion shaft (diagrammed to the left), which must withstand the force resulting from torque transmission, including that which arises due to the pressure angle between the gears. This force can be determined as $\vec{F}_p = \left(F_{p,x}, F_{p,y}, F_{p,z}\right) = \left(0, \frac{\tau_p}{r_p}\tan(\varphi), \frac{\tau_p}{r_p}\right)$ where τ_p is calculated above, φ is the pressure angle of the gear teeth, and $r_p = \frac{d_p}{2}$ is the pitch radius of the bevel pinion (half the pitch diameter). Performing a moment balance about point A, we have that $\vec{r}_c \times \vec{F}_p + \vec{r}_B \times \vec{F}_B = 0$. Dotting the equation with $\hat{y}$ yields $\frac{r_c \tau_p}{r_p} + r_B F_{B,z} = 0$, and dotting the equation with $\hat{z}$ yields $\frac{r_c \tau_p \tan(\varphi)}{r_p} + r_B F_{B,y} = 0$. Solving for $\vec{F}_B$ gives $\vec{F}_B = -\frac{r_c \tau_p}{r_p r_B}\left(\hat{y} - \tan(\varphi)\,\hat{z}\right)$.

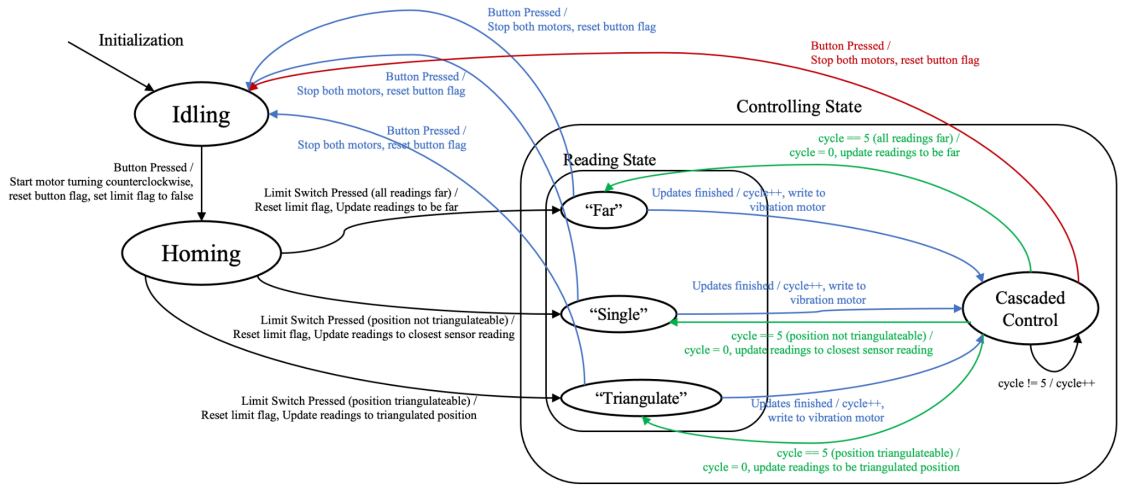
Then, performing an overall force balance on the system yields $\vec{F}_A = \frac{\tau_p\left(r_c - r_B\tan(\varphi)\right)}{r_p r_B}\hat{y} - \frac{\tau_p\left(r_c\tan(\varphi) - r_B\right)}{r_p r_B}\hat{z}$.

The pinion torque τ_p was found above to be 0.0463 lbf·in, the pitch radius r_p used is 0.5 in, the pressure angle is 20°, r_B was designed to be 0.375 in, and r_c was designed to be 0.448 in. Plugging these values into the equations above gives $F_B = \left|\vec{F}_B\right| = 0.117$ lbf and $F_A = \left|\vec{F}_A\right| = 0.093$ lbf, which are only a fraction of the 36 lbf rating of the selected bushings.

Circuit Diagram



State Transition Diagram



Final Thoughts

Our team collaborated and worked tirelessly to make our project a success. We went through multiple trials and errors with designing and manufacturing our initial prototypes, but the concrete milestones we set for ourselves kept us on track to meet our high-level deliverable timelines. After implementing our haptic feedback device, we would like to take it a step further by advancing our data collection process. We would increase the number of ultrasonic sensors in the longitudinal direction to have a more representative mapping of where objects are above the ground. If the project were to be redone, we would have liked to investigate an alternative location for delivering the tactile haptic feedback. Since sensitivity to touch varies along the arm, we would like to research where we can locate the feedback assembly to ensure the user can best feel the haptic feedback.

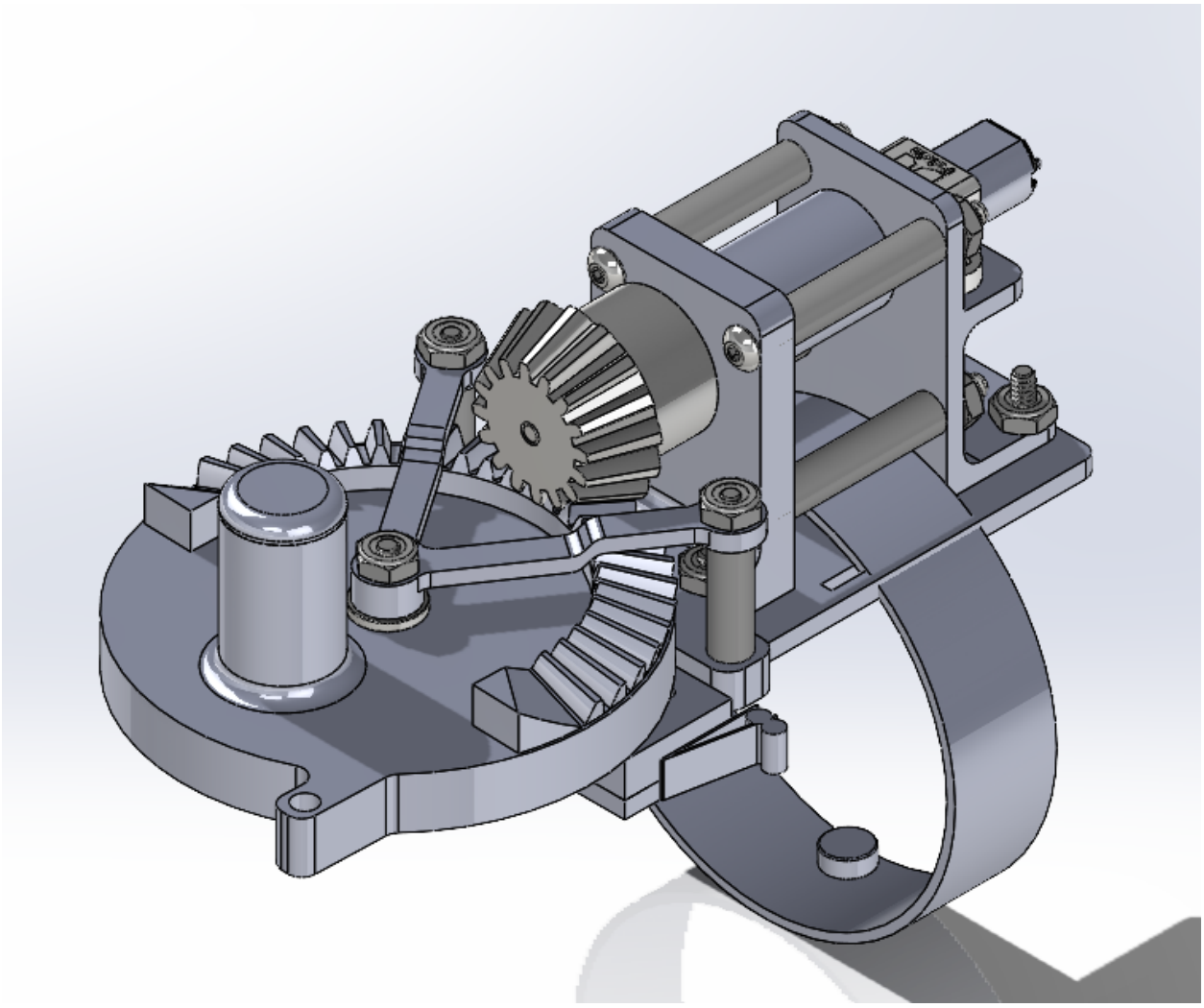
APPENDICES

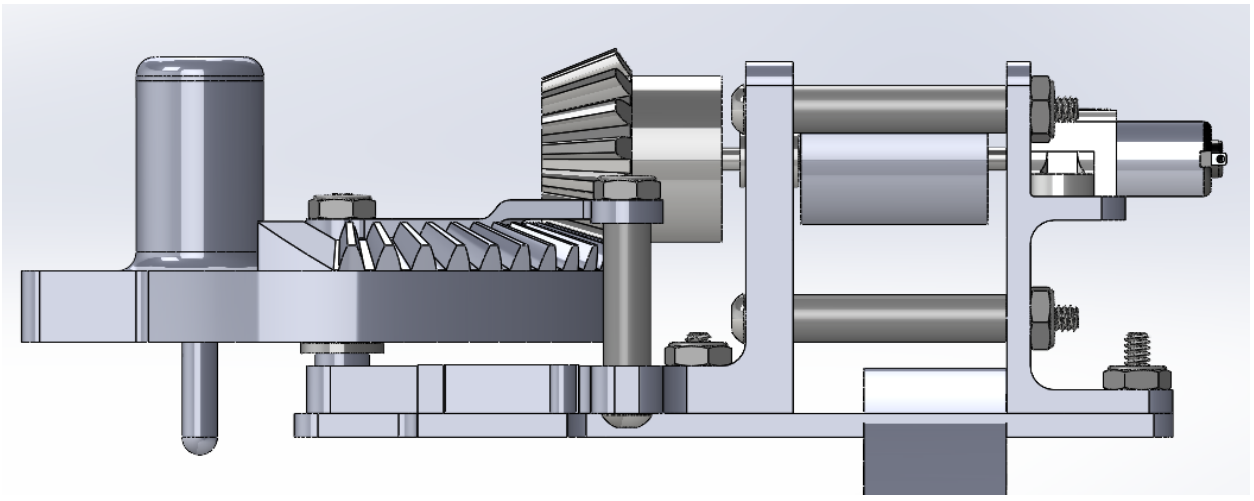
Appendix A: Bill of Materials

Vendor	Part No.	Description	Req. Qty.	Order Qty.	Unit of Measure	Price	Total
Amazon		Flexible Shaft Coupler	1	1	Pack of 5	$14.99	$14.99
McMaster Carr	92510A450	Aluminum Unthreaded Spacer 1/4" OD, 1-1/4" Long, for Number 6 Screw Size	4	4	Each	$0.79	$3.16
McMaster Carr	90633A007	Low-Strength Steel Thin Nylon-Insert Locknut Zinc-Plated, 6-32 Thread Size	11	1	Pack of 100	$10.20	$10.20
McMaster Carr	91255A136	Button Head Hex Drive Screw Black-Oxide Alloy Steel, 6-32 Thread, 1-3/4" Long	4	1	Pack of 10	$7.62	$7.62
McMaster Carr	1637	Shaftless Vibration Motor	1	2	Each	$4.95	$9.90
McMaster Carr	7815K805	Multipurpose Flanged Sleeve Bearing for 5/32" Shaft Diameter and 5/16" Housing ID, 1/4" Long	2	2	Each	$10.38	$20.76
McMaster Carr	92510A447	Aluminum Unthreaded Spacer 1/4" OD, 3/4" Long, for Number 6 Screw Size	2	2	Each	$0.52	$1.04
McMaster Carr	92949A160	18-8 Stainless Steel Button Head Hex Drive Screw 6-32 Thread Size, 1-1/8" Long	2	1	Pack of 100	$10.28	$10.28
McMaster Carr	90357A002	Ultra-Low-Profile Socket Head Screw Alloy Steel, 6-32 Thread Size, 1/2" Long	4	4	Each	$3.86	$15.44
McMaster Carr	2705T111	Light Duty Dry-Running Flanged Sleeve Bearing Thermoplastic-Blend, for 3 mm Shaft Diameter, 3 mm Long	2	2	Each	$1.65	$3.30
McMaster Carr	90337A413	18-8 Stainless Steel Low-Profile Precision Shoulder Screw 5/32" Diameter x 1" Long Shoulder, 9/32" Head Diameter	1	1	Each	$8.77	$8.77
McMaster Carr	1265K31	Rotary Shaft 316 Stainless Steel, 3 mm Diameter, 200 mm Long	1	1	Each	$17.86	$17.86
Initeq		M3-0.5 Long Brass Threaded Inserts	1	1	Pack of 20	$7.99	$7.99
Amazon		HiLetgo 10pcs Micro Limit Switch KW12-3 AC 250V 5A SPDT 1NO 1NC Micro Switch Normally Open Close Limit Switch with Roller Lever Arm Black	1	1	Pack of 10	$5.99	$5.99
Amazon		Nylon Backpack Straps	1	1	Each	$11.32	$11.32
Amazon		Velcro Straps	1	1	Each	$5.00	$5.00

Amazon		Solderable Breadboard	1	1	Each	$11.99	$11.99
Amazon		Jumper Cables	1	1	Each	$8.99	$8.99
Amazon		4.8V Batteries	1	1	Each	$18.73	$18.73
Pololu	2215	75:1 Micro Metal Gearmotor	1	1	Each	$12.71	$12.71
Pololu	989	Micro Metal Gearmotor Bracket Pair	1	1	Each	$2.30	$2.30
Adafruit	4007	Ultrasonic Distance Sensor	4	4	Each	$3.95	$3.95
Adafruit	3619	ESP32 Feather	1	1	Each	$17.56	$17.56
Adafruit	2750	LiPo Battery	1	1	Each	$6.26	$6.26
Jameco	256031	MOSFET BS170	1	1	Each	$0.39	$0.39
Amazon		Starelo 12mm Momentary Push Button Switch, Normally Open	1	5	Pack of 5	$11.99	$11.99
Pololu	2130	DRV8833 Dual Motor Driver Carrier	1	1	Each	$6.95	$6.95

Appendix B: CAD





Appendix C: Full Code

```
#include <ESP32Encoder.h>
#include <Arduino.h>
#include <math.h>
#include <NewPing.h>

ESP32Encoder encoder;


//----------------DEFINITIONS----------------//

// PINS
#define BIN_1 25    // PWM pin 1
#define BIN_2 26    // PWM pin 2
#define ENC1 19     // encoder pin 1
#define ENC2 16     // encoder pin 2
#define BTN 21      // button pin
#define LIM 17      // limit switch pin
#define TRIGGER1 22 // trigger pin for ultrasonic sensor 1
#define ECHO1    23 // echo pin for ultrasonic sensor 1
#define TRIGGER2 32 // trigger pin for ultrasonic sensor 2
#define ECHO2    14 // .
#define TRIGGER3 33 // .
#define ECHO3    15 // .
#define TRIGGER4 12 // .
#define ECHO4    27 // echo pin for ultrasonic sensor 4
#define MOTOR    4   // vibration motor pin

// DIMENSIONS
#define PAN_DIST 10.16  // cm
#define SMALL_ANG PI/18 // radians
#define BIG_ANG PI/9    // radians

// OTHER DEFINITIONS
#define DB_TIME 300      // button debounce time, ms
#define SONAR_NUM 4      // number of ultrasonic sensors
#define MAX_DISTANCE 200 // maximum distance to ping (cm)
#define LPF_LENGTH 10    // number of values to take running avg over


//----------------INITIALIZE VARIABLES----------------//

// X AND Y POSITIONS OF ULTRASONIC SENSORS
const float us_x[SONAR_NUM] = {-PAN_DIST*cos(SMALL_ANG) - PAN_DIST/2*cos(BIG_ANG),
                               -PAN_DIST/2*cos(SMALL_ANG),
                               PAN_DIST/2*cos(SMALL_ANG),
                               PAN_DIST*cos(SMALL_ANG) + PAN_DIST/2*cos(BIG_ANG)};
const float us_y[SONAR_NUM] = {1,
                               PAN_DIST/2*sin(BIG_ANG) + PAN_DIST/2*sin(SMALL_ANG + 1),
                               PAN_DIST/2*sin(BIG_ANG) + PAN_DIST/2*sin(SMALL_ANG + 1),
                               1};

// VARIABLES FOR SENSING ARRAY
float us_dist[SONAR_NUM];       // distance from origin to ultrasonic sensors (cm)
float us_theta[SONAR_NUM];      // angle from origin to ultrasonic sensors (rad)
float dist_btwn[SONAR_NUM - 1]; // distance between ultrasonic sensors (cm)
float readings[SONAR_NUM];      // stores sensor readings (cm)
float min_reading;              // helper variable indicating the minimum reading
float prev_dist[LPF_LENGTH];    // last LPF_LENGTH calculated distances (cm) at any time
float prev_ang[LPF_LENGTH];     // last LPF_LENGTH calculated angles (rad) at any time
float dist;                     // running average of distances (cm)
float ang;                      // running average of angles (rad)
byte min_index;                 // helper variable indicating the index of minimum reading
byte low_index;                 // helper variable indicating the "leftmost" relevant sensor
byte high_index;                // helper variable indicating the "rightmost" relevant sensor
float num;                      // helper variable storing numerator of an arccos argument
float den;                      // helper variable storing denominator of an arccos argument
float d1;                       // helper variable storing reading of "leftmost" relevant sensor
float d2;                       // helper variable storing reading of "rightmost" relevant sensor

// VARIABLES FOR MOTOR CONTROL
int omega = 0;                // measured speed
int omegaDes = 0;             // desired speed
int omegaError = 0;           // error in speed
int omegaErrorSum = 0;        // cumulative error in speed
int theta = 0;                // measured angular position
int thetaDesList[LPF_LENGTH]; // Using a running average of desired positions to eliminate noise
```

```
int thetaDes = 0;             // desired position
int thetaError = 0;           // error in position
int thetaMax = 375;           // max position
int D = 0;                    // duty cycle command for DC motor
int cycle = 0;                // helper variable to ensure sensors are not read too quickly

int Kp_theta = 1;        // outer loop proportional gain for position
int Kp_omega = 30;       // inner loop proportional gain for velocity
int Ki_omega = 1;        // inner loop integral gain for velocity
int omegaDesSat = 20;    // saturator for desired speed to prevent position windup
int omegaErrorSat = 160; // saturator for speed error to prevent speed windup

// STATE VARIABLES
byte state = 0;            // for outer state machine
byte inner_state = 0;      // for inner state machine
byte calculator_state = 0; // for calculator state machine

// TIMER VARIABLES
volatile int count = 0;                // encoder count
volatile bool interruptCounter = false;
volatile bool deltaT = false;
volatile bool debounceFlag = false;
int totalInterrupts = 0;
hw_timer_t * timer1 = NULL;
hw_timer_t * timer2 = NULL;
portMUX_TYPE timerMux1 = portMUX_INITIALIZER_UNLOCKED;
portMUX_TYPE timerMux2 = portMUX_INITIALIZER_UNLOCKED;


// PWM PROPERTIES FOR MOTORS
const int freq = 20000;          // DC motor PWM frequency
const int vibFreq = 5000;        // vibration motor PWM frequency
const int ledChannel_1 = 1;
const int ledChannel_2 = 2;
const int ledChannel_3 = 3;
const int resolution = 8;
const int MAX_PWM_VOLTAGE = 255; // Do not exceed this command

// VIBRATION MOTOR SPEED
int vibSpeed = 0; // Duty cycle command for vibration motor

// ULTRASONIC SENSOR OBJECT ARRAY
NewPing sonar[SONAR_NUM] = {
  NewPing(TRIGGER1, ECHO1, MAX_DISTANCE),
  NewPing(TRIGGER2, ECHO2, MAX_DISTANCE),
  NewPing(TRIGGER3, ECHO3, MAX_DISTANCE),
  NewPing(TRIGGER4, ECHO4, MAX_DISTANCE)
};


//-----------------INTERRUPT INITIAZLIATIONS-----------------//

// TIMER INTERRUPTS

// for position controlling
void IRAM_ATTR onTime1() {
  portENTER_CRITICAL_ISR(&timerMux1);
  count = encoder.getCount( );
  encoder.clearCount ( );
  deltaT = true;
  portEXIT_CRITICAL_ISR(&timerMux1);
}

// for debouncing
void IRAM_ATTR onTime2() {
  if (digitalRead(BTN) == HIGH) {
    portENTER_CRITICAL_ISR(&timerMux2);
    debounceFlag = true;
    portEXIT_CRITICAL_ISR(&timerMux2);
  }
}
```

```
// OTHER INTERRUPT INITIALIZATIONS
bool buttonFlag = false;
bool limitFlag = false;
float last_button_time = 0;
float last_limit_time = 0;

// for button press
void IRAM_ATTR button_isr() {
  buttonFlag = true;
  timerWrite(timer2, 0);
  timerStart(timer2);
}

// for limit switch press
void IRAM_ATTR limit_isr() {
  limitFlag = true;
}


//----------------SETUP----------------//
void setup() {

  // Finish initializing array variables that require calculation
  for (byte i = 0; i < SONAR_NUM; i++) {
    us_dist[i] = sqrt(pow(us_x[i], 2) + pow(us_y[i], 2));
    us_theta[i] = atan2(us_y[i],us_x[i]);
  }
  for (byte i = 0; i < SONAR_NUM - 1; i++) {
    dist_btwn[i] = sqrt(pow(us_x[i] - us_x[i + 1], 2) + pow(us_y[i] - us_y[i + 1], 2));
  }

  // Encoder setup
  ESP32Encoder::useInternalWeakPullResistors = UP; // Enable the weak pull up resistors
  encoder.attachHalfQuad(ENC1, ENC2); // Attache pins for use as encoder pins
  encoder.setCount(0);  // set starting count value after attaching


  // Configure LED PWM functionalitites
  ledcSetup(ledChannel_1, freq, resolution);
  ledcSetup(ledChannel_2, freq, resolution);
  ledcSetup(ledChannel_3, vibFreq, resolution);

  // Attach the channel to the GPIO to be controlled
  ledcAttachPin(BIN_1, ledChannel_1);
  ledcAttachPin(BIN_2, ledChannel_2);
  ledcAttachPin(MOTOR, ledChannel_3);

  // Attach timer interrupts
  timer1 = timerBegin(1, 80, true);
  timerAttachInterrupt(timer1, &onTime1, true);
  timerAlarmWrite(timer1, 10000, true); // 10 ms
  timerAlarmEnable(timer1);
  timer2 = timerBegin(1, 80, true);
  timerAttachInterrupt(timer2, &onTime2, true);
  timerAlarmWrite(timer2, 150000, true); // 150 ms
  timerAlarmEnable(timer2);
  timerStop(timer2);

  // Attach button and limit interrupts
  pinMode(BTN, INPUT_PULLUP);
  attachInterrupt(BTN, button_isr, RISING);
  pinMode(LIM, INPUT_PULLUP);
  attachInterrupt(LIM, limit_isr, RISING);

  // Initilialize vibration motor pin
  ledcWrite(ledChannel_3, LOW);

}
```

```
//----------------MAIN LOOP----------------//
void loop() {

  switch (state) {

    case 0: // OUTER STATE: IDLING
      if (checkButtonPress() == true) {
        startHomingSequence();
        state = 1;
      }
      break;

    case 1: // OUTER STATE: HOMING
      if (checkLimit() == true) {
        setHomePosition();
        state = 2;
        inner_state = 0;
      }
      break;

    case 2: // OUTER STATE: CONTROLLING (Contains inner state machine)
      switch (inner_state) {

        case 0: // INNER STATE: READING (Contains calculator state machine)
          readSensors();
          findMinReadings();
          if (allReadingsFar() == true) {
            calculator_state = 0;
          } else if (triangulateable() == false) {
            calculator_state = 1;
          } else {
            calculator_state = 2;
          }

          switch (calculator_state) {
            case 0: // CALCULATOR STATE: "Far"
              updateReadingsFar();
              break;
            case 1: // CALCULATOR STATE: "Single"
              updateReadingsSingle();
              break;
            case 2: // CALCULATOR STATE: "Triangulate"
              updateReadingsTriangulate();
              break;
            default:
              break;
          }

          writeVibrationMotor();
          cycle = 0;
          inner_state = 1;

          // Check if user switched to idling state
          if (checkButtonPress() == true) {
            toIdlingService();
            state = 0;
          }
          break;

        case 1: // INNER STATE: CASCADED CONTROL
          cascadedControl();
          cycle++;
          if (cycle == 5) {
            inner_state = 0;
          }
          if (checkButtonPress() == true) {
            toIdlingService();
            state = 0;
          }
          break;




        default:
          break;

      }
      break;

    default:
      break;

  }
}
```

```
//----------------FUNCTION LIBRARY----------------//

//- - - - - - - - -ACTION FUNCTIONS START HERE- - - - - - - - - -//
//**************************************************
// This function updates the readings of all sensors
//**************************************************
void readSensors() {
  for (byte i = 0; i < SONAR_NUM; i++)
  {
    readings[i] = sonar[i].ping_cm();
    if (sonar[i].ping_cm() < 1) {
      readings[i] = MAX_DISTANCE;
    }
  }
}


// ************************************************************************
// This function determines the "leftmost" sensor (low_index) and "rightmost"
// sensor (high_index) to be used for calculation. They are found based on
// the minimum reading (closest object) from any of the sensors, identified
// by min_index.
// ************************************************************************
void findMinReadings() {
  min_index = findMinIndex(readings);
  if (min_index == 0) {
    low_index = 0;
    high_index = 1;
  }
  else if (min_index == SONAR_NUM - 1) {
    low_index = SONAR_NUM - 2;
    high_index = SONAR_NUM - 1;
  } else {
    if (readings[min_index - 1] < readings[min_index + 1]) {
      low_index = min_index - 1;
      high_index = min_index;
    } else {
      low_index = min_index;
      high_index = min_index + 1;
    }
  }
}


// ******************************************************
// This function checks if all the readings are "far away",
// i.e. >0.9*MAX_DISTANCE
// ******************************************************
boolean allReadingsFar() {
  return readings[min_index] >= 0.9*MAX_DISTANCE;
}


// ********************************************************************
// This function simply updates the latest distance reading to be the max
// distance and the latest angle reading to be the same as the previous
// angle reading (this function is called only if all the readings are
// "far")
// ********************************************************************
void updateReadingsFar() {
  shiftDistances(MAX_DISTANCE);
  calculateAverageDistance();
  shiftAngles(prev_ang[LPF_LENGTH - 1]);
  calculateAverageAngle();
}
```

```
// ****************************************************************
// This function checks if the detected object has a triangulateable
// position, i.e. the argument for the arccos in triangulation is
// between -1 and 1
// ****************************************************************
boolean triangulateable() {
  d1 = readings[low_index];
  d2 = readings[high_index];
  num = pow(dist_btwn[low_index],2) + pow(d1,2) - pow(d2,2);
  den = 2*dist_btwn[low_index]*d1;
  return abs(num) <= abs(den);
}


// ************************************************************************
// This function simply updates the latest distance reading to be the
// minimum reading of the sensors (this function is called only if the
// position is not triangulateable)
// ************************************************************************
void updateReadingsSingle() {
  shiftDistances(readings[min_index]);
  calculateAverageDistance();
  shiftAngles(us_theta[min_index]);
  calculateAverageAngle();
}

// ************************************************************************
// This function triangulates the position of the nearest obstacle and
// updates the latest distance and angle readings to match. This function
// is called only if the position is triangulateable
// ************************************************************************
void updateReadingsTriangulate() {
  float zeta = acos(num/den);

  float beta = acos((pow(us_dist[low_index],2) + pow(dist_btwn[low_index],2) -
    pow(us_dist[high_index],2))/(2*us_dist[low_index]*dist_btwn[low_index]));

  shiftDistances(sqrt(pow(d1,2) + pow(us_dist[low_index],2) -
    2*d1*us_dist[low_index]*cos(beta + zeta)));

  calculateAverageDistance();

  float gamma = asin(d1/prev_dist[LPF_LENGTH] * sin(beta + zeta));

  shiftAngles(us_theta[low_index] - gamma);

  calculateAverageAngle();
}


//*********************************************************
// This function writes the desired PWM to the vibration motor
//*********************************************************
void writeVibrationMotor() {
  vibSpeed = -2.5*dist + 255;
  ledcWrite(ledChannel_3, vibSpeed);
}
```

```
//*****************************************************
// This function implements cascaded control of the motor
//*****************************************************
 void cascadedControl() {

  if (deltaT) {
      portENTER_CRITICAL(&timerMux1);
      deltaT = false;
      portEXIT_CRITICAL(&timerMux1);

      omega = count;
      theta += omega;
      shiftThetaDes(ang/PI*thetaMax);
      calculateAverageThetaDes();

      thetaError = thetaDes - theta;
      omegaDes = 0.2*Kp_theta*thetaError;
      if (omegaDes >= omegaDesSat) {
        omegaDes = omegaDesSat;
      }
      if (omegaDes <= -omegaDesSat) {
        omegaDes = -omegaDesSat;
      }

      omegaError = omegaDes - omega;
      omegaErrorSum += omegaError;
      if (omegaErrorSum >= omegaErrorSat) {
        omegaErrorSum = omegaErrorSat;
      }
      if (omegaErrorSum <= -omegaErrorSat) {
        omegaErrorSum = -omegaErrorSat;
      }
      D = Kp_omega*omegaError + 0.2*Ki_omega*omegaErrorSum;

      //Ensure that you don't go past the maximum possible command
      if (D > MAX_PWM_VOLTAGE) {
          D = MAX_PWM_VOLTAGE;
      }
      else if (D < -MAX_PWM_VOLTAGE) {
          D = -MAX_PWM_VOLTAGE;
      }

      //Map the D value to motor directionality
      //FLIP ENCODER PINS SO SPEED AND D HAVE SAME SIGN
      if (D > 0) {
          ledcWrite(ledChannel_1, LOW);
          ledcWrite(ledChannel_2, D);
      }
      else if (D < 0) {
          ledcWrite(ledChannel_1, -D);
          ledcWrite(ledChannel_2, LOW);
      }
      else {
          ledcWrite(ledChannel_1, LOW);
          ledcWrite(ledChannel_2, LOW);
      }
  }
}
//- - - - - - - - - -ACTION FUNCTIONS END HERE- - - - - - - - - - -//



//- - - - - - - - - -EVENT CHECKERS + SERVICES START HERE- - - - - - - - - -//
//***************************************
// This function checks for a button press
// ***************************************
bool checkButtonPress() {
  return buttonFlag && debounceFlag;
}


//***********************************************
// This function checks for the limit switch press
//***********************************************
bool checkLimit() {
  return limitFlag;
}
```

```
//***************************************
// This function starts the homing sequence
//***************************************
void startHomingSequence() {
  buttonFlag = false;
  debounceFlag = false;
  timerStop(timer2);
  ledcWrite(ledChannel_1, LOW);
  ledcWrite(ledChannel_2, 225);
}


//*************************************
// This function sets the home position
//*************************************
void setHomePosition() {
  limitFlag = false;
  theta = thetaMax;
  ledcWrite(ledChannel_1, LOW);
  ledcWrite(ledChannel_2, LOW);
}


//*************************************************
// This function is the service when switching to idling
// (stopping motors, resetting button press flags)
//*************************************************
void toIdlingService() {
  buttonFlag = false;
  debounceFlag = false;
  timerStop(timer2);
  ledcWrite(ledChannel_1, LOW);
  ledcWrite(ledChannel_2, LOW);
  ledcWrite(ledChannel_3, LOW);
}
//- - - - - - - - - -EVENT CHECKERS + SERVICES END HERE- - - - - - - - - -//


//- - - - - - - - - -HELPER FUNCTIONS START HERE- - - - - - - - - -//
//***********************************************************
// This function returns the index of the minimum value in arr
//***********************************************************
byte findMinIndex(float arr[]) {
  int min_val = arr[0];
  byte cur_index = 0;
  for (byte i = 1; i < SONAR_NUM; i++)
  {
    if (arr[i] < min_val)
    {
      min_val = arr[i];
      cur_index = i;
    }
  }
  return cur_index;
}


//*********************************************************
// This function shifts all elements of the prev_dist array
// left by 1, adding value to the end
//*********************************************************
void shiftDistances(float value) {
  for (byte i = 0; i < LPF_LENGTH - 1; i++) {
    prev_dist[i] = prev_dist[i + 1];
  }
  prev_dist[LPF_LENGTH - 1] = value;
}
```

```
//**********************************************************
// This function shifts all elements of the prev_ang array
// left by 1, adding value to the end
//**********************************************************

void shiftAngles(float value) {
  for (byte i = 0; i < LPF_LENGTH - 1; i++) {
    prev_ang[i] = prev_ang[i + 1];
  }
  prev_ang[LPF_LENGTH - 1] = value;
}


//**********************************************************
// This function shifts all elements of the thetaDesList array
// left by 1, adding value to the end
//**********************************************************
void shiftThetaDes(float value) {
  for (byte i = 0; i < LPF_LENGTH - 1; i++) {
    thetaDesList[i] = thetaDesList[i + 1];
  }
  thetaDesList[LPF_LENGTH - 1] = value;
}


//**********************************************************
// This function calculates the average of the prev_dist array
// and stores it in dist
//**********************************************************
void calculateAverageDistance() {
  dist = 0;
  for (byte i = 0; i < LPF_LENGTH; i++) {
    dist += prev_dist[i];
  }
  dist /= LPF_LENGTH;
}


//**********************************************************
// This function calculates the average of the prev_ang array
// and stores it in ang
//**********************************************************

void calculateAverageAngle() {
  ang = 0;
  for (byte i = 0; i < LPF_LENGTH; i++) {
    ang += prev_ang[i];
  }
  ang /= LPF_LENGTH;
}


//**************************************************************
// This function calculates the average of the thetaDesList array
// and stores it in thetaDes
//**************************************************************
void calculateAverageThetaDes() {
  thetaDes = 0;
  for (byte i = 0; i < LPF_LENGTH; i++) {
    thetaDes += thetaDesList[i];
  }
  thetaDes /= LPF_LENGTH;
}
```