Umbrella Corp. Solar Energy Harvester

Jackson Hilton, Hugo Maida, Liam McHugh, Andrew Moncada

Opportunity

Solar energy production is a critical part of the developing renewable energy sector. However, this technology is limited by the intensity of sunlight that is able to reach Earth and penetrate the atmosphere. Even a 100% efficient panel could only generate around 1 kW of power per square meter on a perfectly clear day with the sun directly overhead. We aim to create a system to concentrate sunlight on a solar collector to improve effectiveness.

Strategy

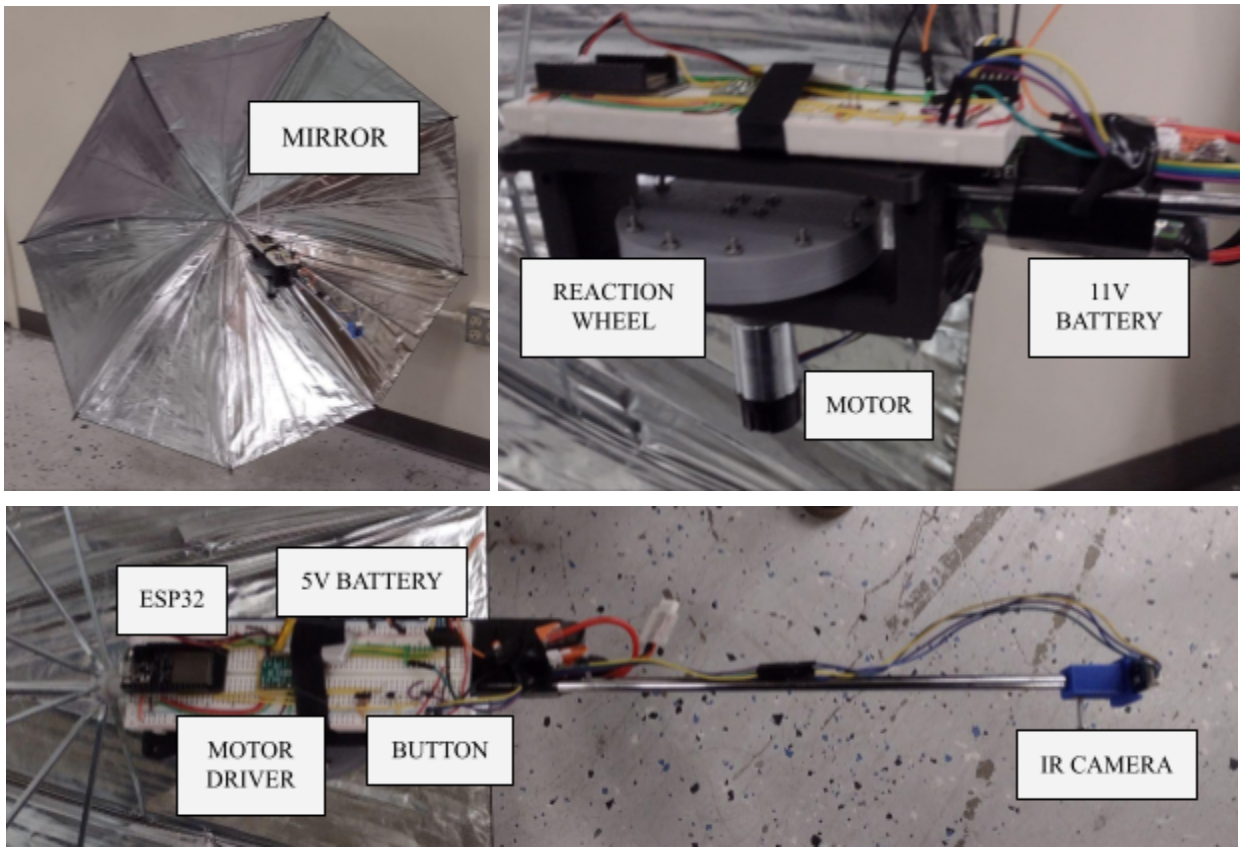
We plan to design a satellite system for solar energy collection. Harvesting this energy in space provides several advantages. Being closer to the sun and outside of the atmosphere both increase the intensity of available solar radiation. In addition, microgravity conditions allow the use of a very large parabolic mirror to further concentrate light.

Due to time and budget constraints, we will be focusing on creating a 1 degree of freedom proof-of-concept model of this system. We will use an infrared camera to detect the position of a heat source relative to our system. We then drive a reaction wheel to exchange angular inertia between the wheel and mirror, actively directing the mirror towards the heat source.

Our goal in the project was to achieve at least 1 degree of angular accuracy and 15 degrees per second squared of angular acceleration. On the acceleration front, we did much better than we set out to, easily hitting almost 30 degrees per second squared.

On the other hand, our angular accuracy is worse than we hoped. Low readout frequency (below 10 Hz) from our economical IR camera and a low level of control authority from our encoder prevents the system from making high-frequency control adjustments. Although partially solved with efficient smoothing and predictive tracking capabilities, the system still struggles with highly dynamic events (heat source rapidly changing position). During stable operation with a static source, accuracy of around 2-5 degrees is achieved. With a shifting point source or wind disturbance, accuracy decreases roughly linearly with the angular acceleration of the point source movement.

System Overview



Functional Decisions

We determined our minimum output torque using our target angular acceleration of 15 deg/s^2. The umbrella is 40g and is 44" long. Since the mass is concentrated in the mylar sheet of the umbrella, we assume that all 400g is 22" (0.5588m) away from the hub to simplify our calculations.

$I = mR^2 = 0.125\ kgm^2$

$\tau = \alpha I = 32.7\ Nmm$

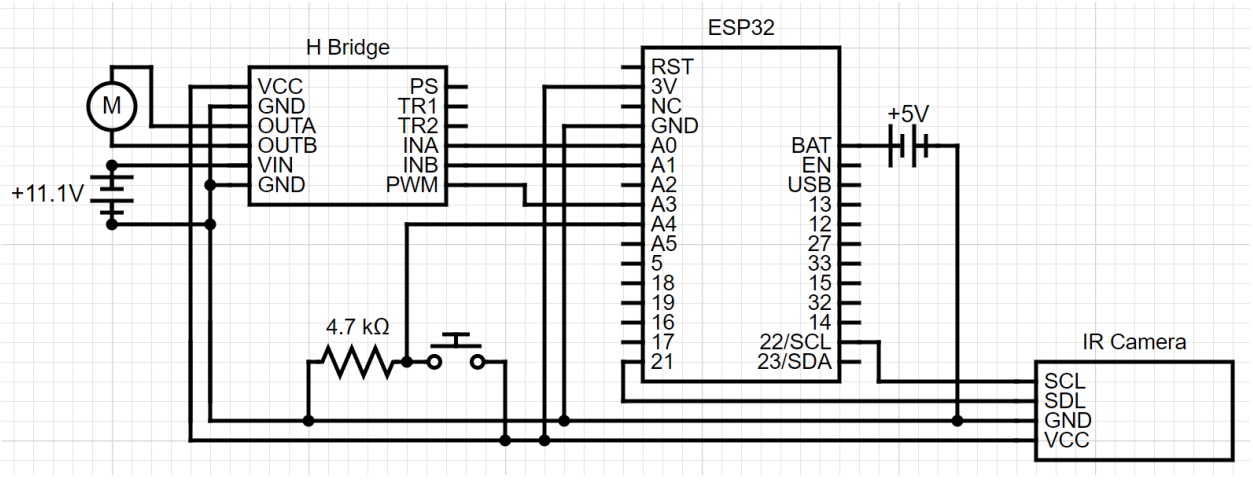
This constrains our motor choice, and we are able to find many options that can directly achieve this torque, so we do not require a speed-reducing transmission.

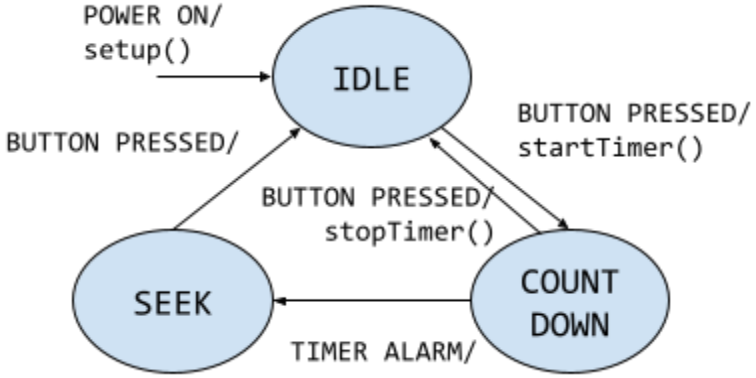
For the reaction wheel design we want a 100:1 angular acceleration ratio between the reaction wheel and the device, and an 8cm wheel.

$0.125kgm^2 = m \times 0.08^2$

$m = 20g$

Circuit & State Diagrams





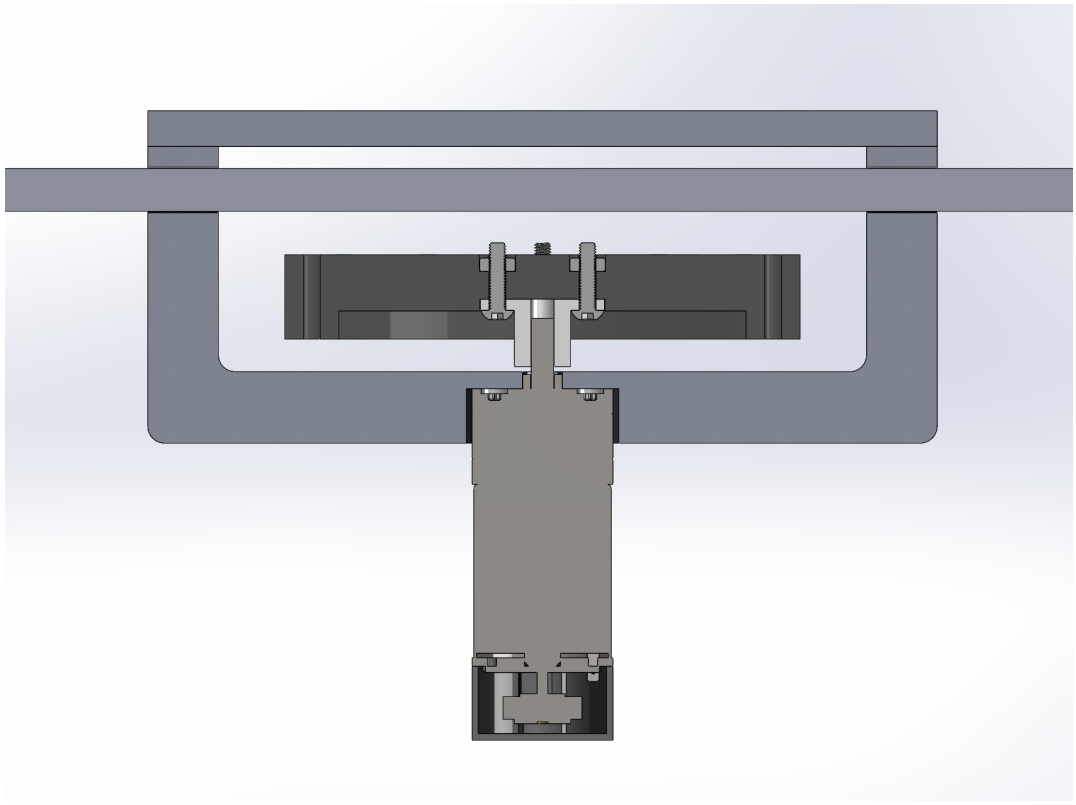
Reflection

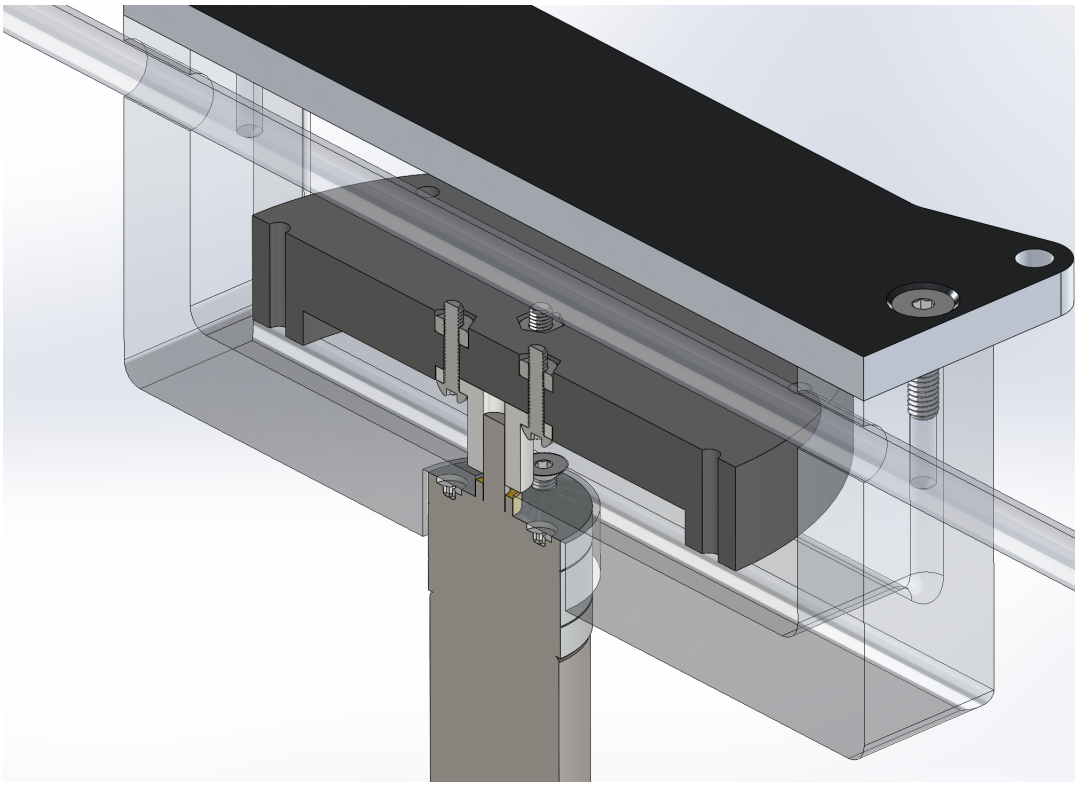
The project ended up being difficult for reasons we did not expect. At the outset of the project we were very enthusiastic to get it working and had a very strong push at the beginning. This was both good and bad. It was helpful because we managed to get a lot of work on the conceptual design done early. The difficulty arose because in our enthusiasm we spec'd out the project without a basic proof of concept. This led to many integration problems because we couldn't nail down many parts of our system until much later after the basic construction was complete. I think in the future that pushing hard to work on a very basic prototype early as opposed to being focused on making "final" decisions would have given us more time to discover issues with our design.

Appendix A: Bill of Materials

Component	*Unit Cost*	*Total QTY*	*Qty Ordered*	*Order Date*	*Total Cost*	Supplier
TOTAL COST (tax adjusted)					$219.35	
Hardware		Category Total	$116.07			
Motor	$60.00	1	1	9/28/2023	$60.00	Pololu
Motor Housing	$17.28	1	0		$17.28	3D Print
Electronics Mount	$8.37	1	0		$8.37	3D Print
M4 Heat-Set Insert	$0.00	2	0		$0.00	Jacobs
M4 Set Screws	$0.00	2	0		$0.00	Jacobs
Parabolic Mirror	$30.42	1	1	10/9/2023	$30.42	Amazon
Reaction Wheel	$0.00	1	0		$0.00	3D Print
M2.5 Nuts & Bolts	$0.00	12	0		$0.00	Jacobs
Electronics		Category Total	$85.81			
ESP32	$0.00	1	1		$0.00	Lab Kit
Motor Driver	$11.95	1	1	10/4/2023	$11.95	Pololu
IR Camera	$73.86	1	1	9/15/2023	$73.86	Amazon
11V Battery	$0.00	1	0		$0.00	Liam
Jumper Wires (Rainbow Pack)	$0.00	1	1		$0.00	Lab Kit

Appendix B: CAD Screenshots





Appendix C: Code

```
//MAIN CODE
//::::::Libraries:::::::// 
#include <Arduino.h>
//#include <esp_now.h>
//#include <WiFi.h>
//#include <Wire.h>
//#include <SPI.h>
//#include "Adafruit_MAX31855.h"
#include <ESP32Encoder.h>
#include <Adafruit_MLX90640.h>

// DEFINE ESP PINS
#define BIN_1 26
#define BIN_2 25
#define PWM 39
#define LED_PIN 13
#define BTN 36


// DEFINE SENSOR READRATE
#define READ_DELAY 70
#define sendDelay 140

//////////////////////IR SETUP//////////////////////////
#define WIDTH  32
#define HEIGHT 24
#define RADIUS 6 // vertical distance to look at
Adafruit_MLX90640 mlx;
int iMax = 0;
int error = 0;
float frame[WIDTH*HEIGHT]; // buffer for full frame of temperatures


////////////////////////// Communication Setup
/////////////////////////////////
int WIFIDEBUG = 1; // IF TOGGLED TO 1: Don't send/receive data.
int DEBUG = 1 ;
short int queueLength = 0;
// Create a struct_message called Packet to be sent.
//struct struct_message Packet;
```

```
//// Create a queue for Packet in case Packets are dropped.
//struct struct_message PacketQueue[120];

//::::::Broadcast Variables:::::::://
//esp_now_peer_info_t peerInfo;
// REPLACE WITH THE MAC Address of your receiver
//uint8_t broadcastAddress[] = {0xB0, 0xA7, 0x32, 0xDE, 0xC1, 0xFC};
// Callback when data is sent
// void OnDataSent(const uint8_t *mac_addr, esp_now_send_status_t status) {
//  sendTime = millis();
// }

// Callback when data is received
//void OnDataRecv(const uint8_t * mac, const uint8_t *incomingData, int len)
{
//  memcpy(&Commands, incomingData, sizeof(Commands));
//  SERIALState = Commands.COMState;
//}


//::::::STATE VARIABLES:::::::://
enum STATES {IDLE, CTDWN, SEEKMODE}; //add initialize state for countdown
String state_names[] = {"IDLE", "CTDWN", "SEEK"};
const float pi = 3.1415;
int DAQState;
int CommandState;
int ctdwn_time = 2000; //ms
float angle = 180; // ANGLE FROM DESIRED ORIENTATION. Initiated as 180deg(pi)
float lastangle = 180; //used for speed deriv.
double targetAngle = 0; // Angle target point
float loopOnce=0; // Used to reset loop routine

int event_timer;
int del = 150;
//Setup variables ----------------------------------
volatile bool buttonIsPressed = false;

//Initialization -----------------------------------
void IRAM_ATTR isr() {  // the function to be called when interrupt is
triggered
  if (millis()-event_timer > del) {
  buttonIsPressed = true; }
}
```

```
// setting PWM properties ----------------------------
const int freq = 5000;
const int ledChannel_1 = 1;
const int ledChannel_2 = 2;
const int ledChannel_PWM = 3;
const int resolution = 8;
int motor_PWM = 0;
int MAX_PWM = 255;
int lastrpm;
int i = 0;
float timed = 0;
int lasttime;
int sendTime;

//////////////////// Control Parameters //////////////////////////
int read_freq = 10; //SET CONSERVATIVELY - IR CAMERA MAX IS 16
int lastread = 0;
int motor_RPM = 0;
int MAX_RPM = 1200;
int lastcalc = 0;
float deltat = 0.0; //angle measurement interval

//////////////////// Begin of PID parameters
////////////////////////////////
// Good YouTube video resource for PID's
https://www.youtube.com/watch?v=0vqWyramGy8
double kp = -4.5;
double ki = 0;
double kd = 0.005;
volatile bool rd = false;
//////////////////// End of PID parameters
////////////////////////////////

//////////////////////////////// Begin of Wheel Speed Ctrl Vars
//////////////////////
double kp_speed = 1.5;
double ki_speed = 0.01;
double kd_speed = 0; // NOT USED
float pwmOut=0;
float pwmOut2=0;
int omegaSpeed = 0;
int omegaDes = 30;
```

```
int omegaMax = 2200;   // CHANGE THIS VALUE TO YOUR MEASURED MAXIMUM SPEED
int D = 0;
byte state = 0;
int potReading = 0;
float SumErr = 0;
////////////////////////////////// End of PID speed loop Vars
//////////////////////


////////////////////////////////// Begin of PI_pwm Vars
//////////////////////////
float positions=0;
int cc;
float speeds_filter;
////////////////////////////////// End of PI_pwm Vars
////////////////////////////


//######################### Begin of Kalman Filter
#####################################################
// Good YouTube video resource for Kalman Filter
https://www.youtube.com/watch?v=mwn8xhgNpFY
float Q_angle = 0.001;    // Covariance of angle noise
float Q_gyro = 0.003;    // Covariance of angle drift noise
float R_angle = 0.5;    // Covariance of accelerometer
char C_0 = 1;
float dt = 0.005; // The value of dt is the filter sampling time
float K1 = 0.05; // a function containing the Kalman gain is used to
calculate the deviation of the optimal estimate
float K_0,K_1,t_0,t_1;
float angle_err;
float q_bias;    // Instrument Drift
float angleY_one;
float angle_speed = 0.0;
float Pdot[4] = { 0, 0, 0, 0};
float P[2][2] = {{ 1, 0 }, { 0, 1 }};
float  PCt_0, PCt_1, E;
//############################ End of Kalman Filter Vars
#####################################################
```

```
////////////////////////// ENCODER SETUP
/////////////////////////////////////////
ESP32Encoder encoder;

//Setup interrupt variables ----------------------------
volatile float count = 0; // encoder count
int wheelspd = 0;
float cpr = 211.2; //encoder multiplied by gear ratio
volatile bool interruptCounter = false;    // check timer interrupt 1
volatile bool deltaT = false;     // check timer interrupt 2
int totalInterrupts = 0;   // counts the number of triggering of the alarm
hw_timer_t * timer0 = NULL;
hw_timer_t * timer1 = NULL;
portMUX_TYPE timerMux0 = portMUX_INITIALIZER_UNLOCKED;
portMUX_TYPE timerMux1 = portMUX_INITIALIZER_UNLOCKED;

//Interrupt Initialization ----------------------------------
void IRAM_ATTR onTime0() {
  portENTER_CRITICAL_ISR(&timerMux0);
  interruptCounter = true; // the function to be called when timer interrupt
is triggered
  portEXIT_CRITICAL_ISR(&timerMux0);
}

void IRAM_ATTR onTime1() {
  portENTER_CRITICAL_ISR(&timerMux1);
  count = encoder.getCount( );
  encoder.clearCount ( );
  deltaT = true; // the function to be called when timer interrupt is
triggered
  portEXIT_CRITICAL_ISR(&timerMux1);
}
////////////////////// ENCODER SETUP END
/////////////////////////////////////////



void setup() {
  Serial.begin(115200);
  while (!Serial) delay(1); // wait for Serial
  Serial.print("Serial Start");
  pinMode(LED_PIN, OUTPUT); // configures the specified pin to behave either
as an input or an output
```

```
  digitalWrite(LED_PIN, LOW); // sets the initial state of LED as turned-off
  ESP32Encoder::useInternalWeakPullResistors = UP; // Enable the weak pull up
resistors
  encoder.attachHalfQuad(33, 27); // 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_PWM, freq, resolution);

  // attach the channel to the GPIO to be controlled
  ledcAttachPin(BIN_1, ledChannel_1);
  ledcAttachPin(BIN_2, ledChannel_2);
  ledcAttachPin(PWM, ledChannel_PWM);

  // initialize timer
  timer0 = timerBegin(0, 80, true);  // timer 0, MWDT clock period = 12.5 ns
* TIMGn_Tx_WDT_CLK_PRESCALE -> 12.5 ns * 80 -> 1000 ns = 1 us, countUp
  timerAttachInterrupt(timer0, &onTime0, true); // edge (not level) triggered
  timerAlarmWrite(timer0, 2000000, true); // 2000000 * 1 us = 2 s, autoreload
true

  timer1 = timerBegin(1, 80, true);  // timer 1, MWDT clock period = 12.5 ns
* TIMGn_Tx_WDT_CLK_PRESCALE -> 12.5 ns * 80 -> 1000 ns = 1 us, countUp
  timerAttachInterrupt(timer1, &onTime1, true); // edge (not level) triggered
  timerAlarmWrite(timer1, 10000, true); // 10000 * 1 us = 10 ms, deltaT
period

  // at least enable the timer alarms
  timerAlarmEnable(timer0); // enable
  timerAlarmEnable(timer1); // enable
  ledcWrite(ledChannel_PWM, LOW);


  ////////////////////////////////Event//////////////////////////
  pinMode(BTN, INPUT);
  attachInterrupt(BTN, isr, FALLING);  // set the "BTN" pin as the interrupt
pin;
  event_timer = millis();
  lastcalc = millis();
////////////////// Communication ////////////////////////////////
```

```
  // Broadcast setup.
  // Set device as a Wi-Fi Station
//  WiFi.mode(WIFI_STA);
//  // Print MAC Accress on startup for easier connections
//  Serial.println(WiFi.macAddress());

  // Init ESP-NOW
//  if (esp_now_init() != ESP_OK) {
//    Serial.println("Error initializing ESP-NOW");
//    return;
//  }

  // Once ESPNow is successfully Init, we will register for Send CB to
  // get the status of Trasnmitted packet
  // esp_now_register_send_cb(OnDataSent);

  // Register peer
//  memcpy(peerInfo.peer_addr, broadcastAddress, 6);
//  peerInfo.channel = 0;
//  peerInfo.encrypt = false;

  // Add peer
//  if (esp_now_add_peer(&peerInfo) != ESP_OK){
//    Serial.println("Failed to add peer");
//    return;
//  }
  // Register for a callback function that will be called when data is
received
//  if (!WIFIDEBUG) {
//    esp_now_register_recv_cb(OnDataRecv);
//  }

  sendTime = millis();
  lasttime = millis();
  DAQState = IDLE;

  Serial.print("MLX SETUP Begin");
  if (! mlx.begin(MLX90640_I2CADDR_DEFAULT, &Wire)) {
    Serial.println("MLX90640 not found!");
  } else {
    Serial.println("Found Adafruit MLX90640");

    Serial.print("Serial number: ");
```

```
    Serial.print(mlx.serialNumber[0], HEX);
    Serial.print(mlx.serialNumber[1], HEX);
    Serial.println(mlx.serialNumber[2], HEX);
  }
  Serial.print("Complete");
  mlx.setMode(MLX90640_CHESS);
  mlx.setResolution(MLX90640_ADC_16BIT);
  mlx.setRefreshRate(MLX90640_16_HZ);
  Wire.setClock(1000000); // max 1 MHz

  Serial.print("Setup Complete");
}




void loop() { // main code here, to run repeatedly:
  ReactionWheelRPM();
  syncDAQState();
  logData(); //data report - controlled by debug modes
  if ((millis()-lastread)>(1000/read_freq)) {
    Get_Readings();
    Calculate_Angle();
    lastread = millis();
    rd = true;}
  if (deltaT) { //Speed Interrupt Action
    wheelspd = 6000*count/cpr; //speed in rpm
    deltaT = false;  }
  switch (DAQState) {
    case (IDLE):
      if (CommandState == CTDWN) { DAQState = CommandState; }
      digitalWrite(LED_PIN, LOW);
      ledcWrite(ledChannel_2, LOW);
      ledcWrite(ledChannel_1, LOW);
      motor_RPM = 200;
      if (buttonIsPressed == true && millis()-event_timer > del) {
        CommandState = CTDWN;
        event_timer = millis();
        buttonIsPressed = false;
      }
      break;
    case (CTDWN):
      if (CommandState == SEEKMODE) { DAQState = CommandState; }
```

```
      digitalWrite(LED_PIN, HIGH);
//      if (buttonIsPressed == true && millis()-event_timer > del) {
//        CommandState = IDLE;
//        event_timer = millis();
//        buttonIsPressed = false;
//      }
      if ((millis() - event_timer) > ctdwn_time){
        CommandState = SEEKMODE;
      }
      break;
    case (SEEKMODE):
      if (CommandState == IDLE) { DAQState = CommandState; }
      if (rd == true) {PID(); rd = false;}
      lasttime = millis();
      if (buttonIsPressed == true && millis()-event_timer > del) {
        CommandState = IDLE;
        motor_RPM = 0;
        event_timer = millis();
        buttonIsPressed = false;
      }
      break;
  }
}


void syncDAQState() {
    if (Serial.available() > 0) {
    // Serial.read reads a single character as ASCII. Number 1 is 49 in
ASCII.
    // Serial sends character and new line character "\n", which is 10 in
ASCII.
    int CommandState = Serial.read() - 48;}
}

void Get_Readings() {
  // take IR camera readingsn if newtime is over GenDelay-Lasttime
    mlx.getFrame(frame);
    // Compress to 1D array by averaging
    int i;
    int n;
    float line[WIDTH];
    for(int col = 0; col < WIDTH; col++){
      n = 0;
```

```
      for(int row = HEIGHT/2 - RADIUS; row <= HEIGHT/2 + RADIUS; row++){
        i = WIDTH * row + col;
        if(!isnan(frame[i])){
          line[col] += frame[i];
          n++;
        }
      }
      line[col] = line[col] / n;
    }
    // Blur to smooth data
    float line_filt[WIDTH];
    line_filt[0] = 0.25 * line[0] + 0.5 * line[0] + 0.25 * line[1];
    for(int i = 1; i < WIDTH-1; i++){
      line_filt[i] = 0.25 * line[i] + 0.5 * line[i+1] + 0.25 * line[i+2];
    }
    line_filt[WIDTH-1] = 0.25 * line[WIDTH-2] + 0.5 * line[WIDTH-1] + 0.25 *
line[WIDTH-1];

    // Find max temp location
    //int iMax = 0;
    float max = 0;
    for(int i = 0; i < WIDTH; i++){
      if(line_filt[i] > max){
        max = line_filt[i];
        iMax = i;
      }
    }
}

void Calculate_Angle() {
  //calculate angle from ideal. Use for state-space control
  //use angle memory to calculate angular velocity
  error = iMax - WIDTH/2;
  angle = (error*110)/32;
  angle = angle*0.7 + lastangle*0.3;
  float deltat = (millis()-lastcalc); //seconds
//  Serial.print(deltat);
  angle_speed = (lastangle-angle) / deltat;
  lastcalc = millis();
  lastangle = angle;
//  Kalman_Filter(angle, angle_speed)
}
```

```
void PID() {
  positions += (angle + targetAngle);
  positions = constrain(positions, -355,355);
//  Serial.println(positions);
//  Serial.println(angle_speed);
  motor_RPM += (kp * (angle + targetAngle) + ki * (positions) - kd *
angle_speed);
  motor_RPM = constrain(motor_RPM, -MAX_RPM, MAX_RPM);
  digitalWrite(LED_PIN, !digitalRead(LED_PIN));
  }

void ReactionWheelRPM(){
  float RPMerr = motor_RPM - wheelspd;
  Serial.print(RPMerr);
  SumErr += RPMerr;
  motor_PWM = kp_speed*RPMerr + ki_speed*SumErr;
  motor_PWM = constrain(motor_PWM, -MAX_PWM, MAX_PWM);
  if (motor_PWM > 0) {
    ledcWrite(ledChannel_2, motor_PWM); // clockwise
    ledcWrite(ledChannel_1, LOW); // power control while cw
  } else { // PWM is commanded negative (ccw)
    motor_PWM = abs(motor_PWM);
    ledcWrite(ledChannel_1, motor_PWM); // ccw
    ledcWrite(ledChannel_2, LOW); // power control while ccw
  }
}

////////////////////////////// Kalman Filter Calculations
//////////////////////
void Kalman_Filter(double angle_m, double vel_m)
{
  angle += (vel_m - q_bias) * dt;          //prior estimate
  angle_err = angle_m - angle;

  Pdot[0] = Q_angle - P[0][1] - P[1][0];    //The differential of the
covariance of the prior estimate error
  Pdot[1] = - P[1][1];
  Pdot[2] = - P[1][1];
  Pdot[3] = Q_gyro;

  P[0][0] += Pdot[0] * dt;    //The integral of the covariance differential
of the prior estimate error
```

```
  P[0][1] += Pdot[1] * dt;
  P[1][0] += Pdot[2] * dt;
  P[1][1] += Pdot[3] * dt;

  //Intermediate variables in matrix multiplication
  PCt_0 = C_0 * P[0][0];
  PCt_1 = C_0 * P[1][0];
  //denominator
  E = R_angle + C_0 * PCt_0;
  //gain value
  K_0 = PCt_0 / E;
  K_1 = PCt_1 / E;

  t_0 = PCt_0;  // Intermediate variables in matrix multiplication
  t_1 = C_0 * P[0][1];

  P[0][0] -= K_0 * t_0;    // Posterior estimation error covariance
  P[0][1] -= K_0 * t_1;
  P[1][0] -= K_1 * t_0;
  P[1][1] -= K_1 * t_1;

  q_bias += K_1 * angle_err;    // Posterior estimate
  angle_speed = vel_m - q_bias;   // The differential of the output value
gives the optimal angular velocity
  angle += K_0 * angle_err; // Posterior estimation; get the optimal angle
}



/////////////// DATALOGGING ////////////////////////////////////
void logData() {
  if (DEBUG) {printReadings();}
  if (millis()-sendTime > sendDelay) {
    sendTime = millis();
    addPacketToQueue();
    sendQueue();
  }
}

void printReadings() {
  Serial.print("Imax: ");
  Serial.print(iMax);
  Serial.print("   Ang: ");
```

```
  Serial.print(angle);
  Serial.print("   DAQ: ");
  Serial.print(DAQState);
  Serial.print("   CMD: ");
  Serial.print(CommandState);
  Serial.print("   CommRPM: ");
  Serial.print(motor_RPM);
  Serial.print("   RealRPM: ");
  Serial.println(wheelspd);
}

void addPacketToQueue() {
//  if (queueLength < 40) {
//    queueLength += 1;
//    PacketQueue[queueLength].messageTime = millis();
//    PacketQueue[queueLength].angle = angle;
//    PacketQueue[queueLength].queueLength = queueLength;
//    PacketQueue[queueLength].DAQState    = DAQState;
//  }
}

void sendQueue() { //need to add supporting structs
//  if (queueLength < 0) {
//    return;
//  }
  // Set values to send
//  Packet = PacketQueue[queueLength];
//  if (!WIFIDEBUG) {
//    // Send message via ESP-NOW
//    esp_err_t result = esp_now_send(broadcastAddress, (uint8_t *) &Packet,
sizeof(Packet));
//    if (result == ESP_OK) {
//      // Serial.println("Sent with success Data Send");
//      queueLength -= 1;
//    } else {
//      Serial.println("Error sending the data");
//    }
//  }
}
```