ME102B -- Fall 2020
Final Project

Makeshift Rock Tumbler with Ultrasonic Feedback Control
By: Asena Yildiz

Section 1: Project Description:

I recently bought a pound of petrified wood and thought that building a rock-tumbling device would be a useful application of my learnings from our mechatronics class. Specifically, I wanted to engineer a rock tumbler with materials lying around. I also wanted to get a strong foundation of motor control. I looked online to research different methods on building your own rock tumblers, and I decided I would utilize a single motor to run the system, placing the rock tumbling housing into a box, with the motor turning a wheel that touches the cylinder, turning it with friction. For my cylinder, I repurposed a ground coffee jar. The cylinder contains grit and stones for polishing. I also wanted to implement feedback control of the motor as it is spinning, to reach a target angular velocity set by the user.

After doing some dynamics calculations, I figured out the appropriate torque and max RPMs necessary for the motor I needed to buy. Specifically for rock tumblers, the angular velocity of the housing does not need to be very fast and I saw online that most rock tumblers do not achieve angular velocities higher than 35 RPMs. I decided that for demonstration purposes, the user would not set the angular velocity of the canister directly, and I focused my efforts to determine the wheel's angular velocity. Future iterations of this product might involve using gear relationships to extrapolate the necessary RPM of the motor to achieve a given RPM for a canister.

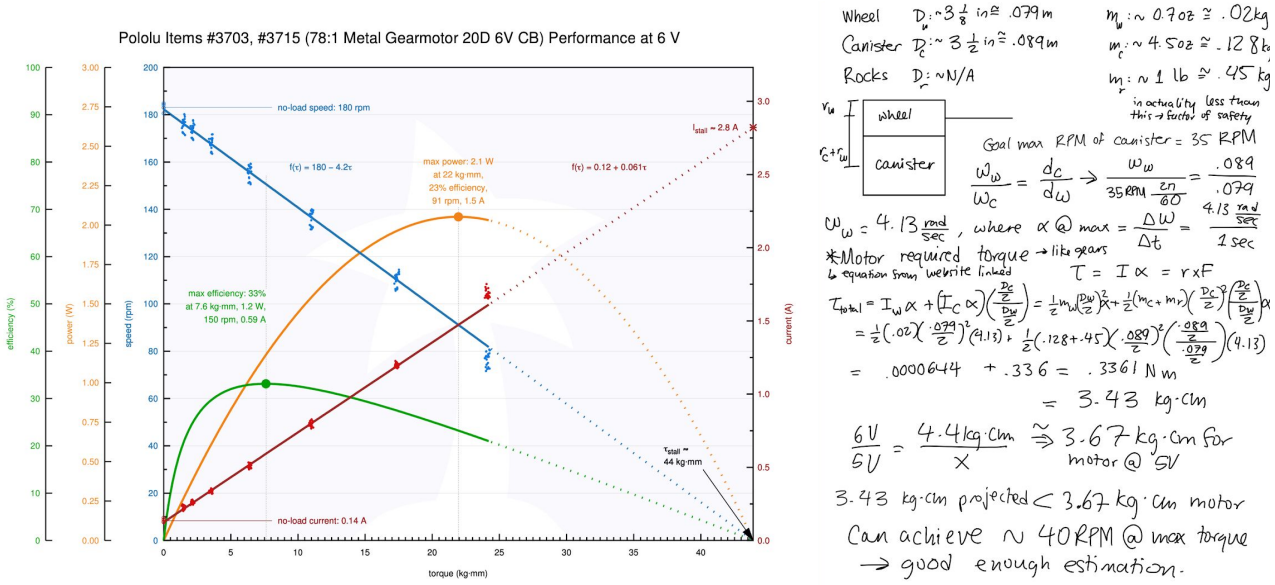

Figure 1: Pololu Motor datasheet that suited the motor requirements estimated. Calculations for estimation of motor requirements (i.e. angular velocity and maximum torque) based off of link: http://www.mechengdesign.co.uk/PlannedWeb/mech226/gearsys/gearaccel.htm

Even though I had good intentions buying a motor encoder for the Pololu motor, when my parts arrived, I realized I bought the wrong version of my motor, one without the dual shaft, so I was unable to use the encoder. Initially I was frustrated with this, but then I decided I would engineer a new solution with my lab kit's sensors, to act like a makeshift encoder. As a result, the scope of my project shifted to the following: (1) exploring the use of an ultrasonic sensor for feedback control of a motor, specifically to allow the user to set an angular velocity for rock tumbling, (2) utilize calibration techniques, and (3) have a single motor turn a canister. I will walk through the design considerations and work done to achieve this result.

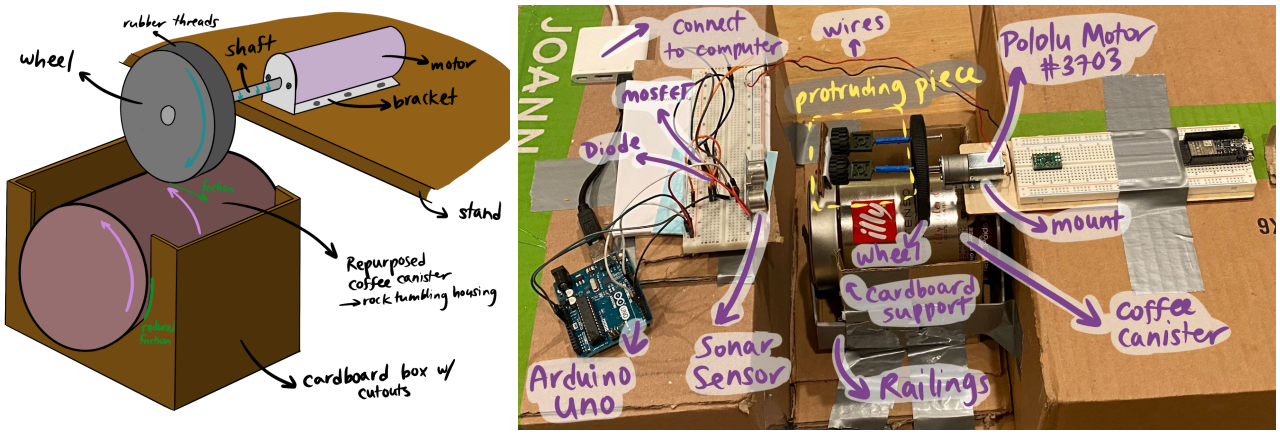

Figure 2: Setup Images -- on the left is my initial concept and on the right is the most updated prototype.

For the device's operation and testing, this is a link to the example video: https://youtu.be/1xXfg8kDjbs.

Section 2: Electromechanical details:

Interacting with the rock tumbling housing (canister): The wheel's rubber track touches off on the metallic canister, utilizing friction to rotate the container as the motor applies a torque to the system. For supporting the housing, a cardboard box with reduced friction (rubbed down) cutouts holds the canister in place as it turns.

Feedback system: A PI controller adjusts the RPM to match the target entered in the Arduino serial monitor. By having a piece protrude perpendicular to the wheel, and setting up a sonar sensor facing the wheel, I was able to approximate the angular velocity of the wheel by marking 1 revolution whenever the sonar sensor saw the piece. For tuning the PI controller, since a rock tumbler runs for several hours, having a fast approach to the target speed was not very important, so I was conservative with my K_P and K_I values. Additionally, there is occasional imprecision from the sonar sensor technique, so the PI controller does not use the most recently measured RPM as its input, but rather the average of the last 5 RPM measurements.

Calibration & Running: Due to the fact that the setup can change slightly when inserting and removing the rock tumbler, determining when a revolution occurs is difficult to do with hardcoded threshold values for sonar output. To fix this, when the rock tumbler is first started up, before doing anything, it enters a calibration stage. There is also a periodic recalibration that occurs every 10 minutes to account for any environmental changes that may have occurred. During the calibration stage the program looks for successive strings of large values or small values. It uses these values to approximate a high and low threshold. During the running phase, values that are close to these calibrated values (within some hard coded delta) are used to determine when a revolution occurs. Specifically, if the program has previously seen a string of values close to the calibrated low value and now sees a string of values close to the calibrated high value, it will mark that as one revolution. It can then use the time differential from the previous revolution to approximate RPM.

In order to account for momentarily blips of incorrect data, the calibration phase and the run-phase both require a certain number of data points in a row in order to confirm it is indeed a new revolution. By playing around with this method I was able to get a very good approximation for RPM. In a test, I hand counted the number of revolutions, getting 62, when the RPM calculated by the Arduino was 62.7.

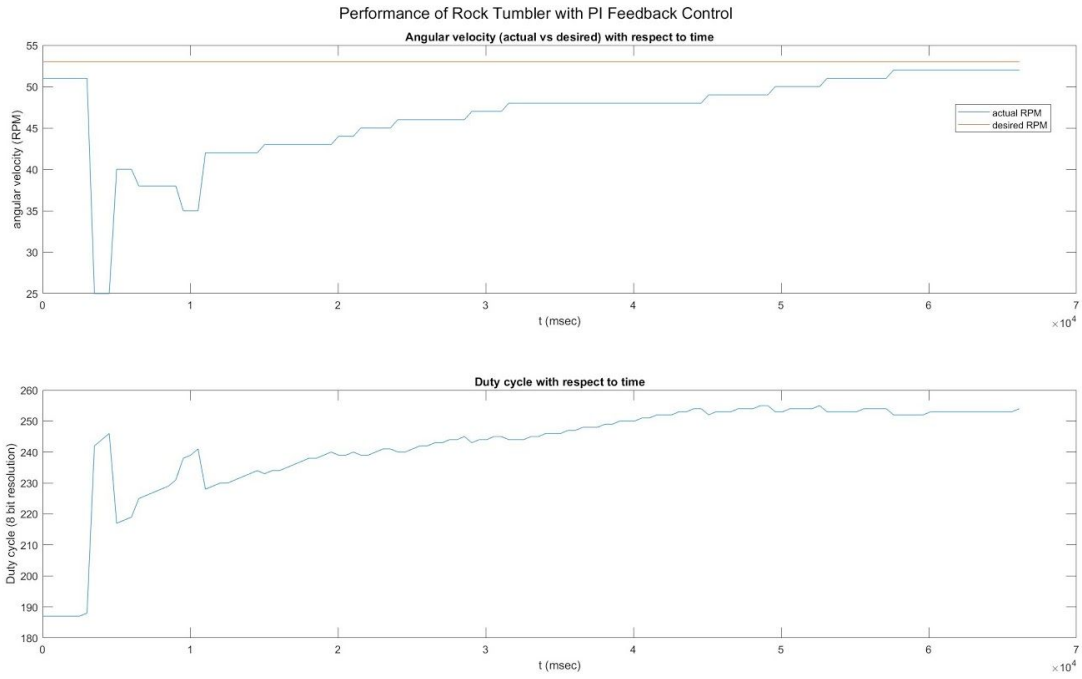

Figure 3: Graphing actual angular velocity vs desired angular velocity (53 RPM), as well as the corresponding duty cycle. If I run the rock tumbler for a very long time, it should eventually reach the desired angular velocity.

Section 3: Circuit Details:

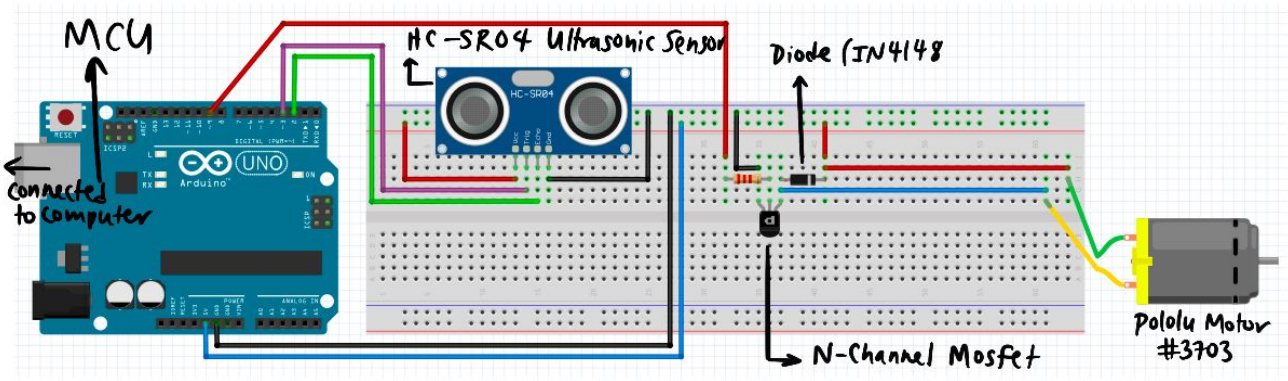

Figure 4: Circuit Diagram made with Fritzing

Because of my ordering mistake, I needed to engineer a unique solution to get a sense of the angular velocity of the spinning wheel and provide feedback control with a PI controller. For the sake of the project, my focus was to learn how to use an ultrasonic sensor, implement feedback control, and turn a wheel. For the success of the design, these were the main electronic components: (1) a Brushed DC Motor (Pololu #3703), (2) an ultrasonic sensor (HC-SR04), (3) Arduino Uno, and (4) Transistor and Diode.

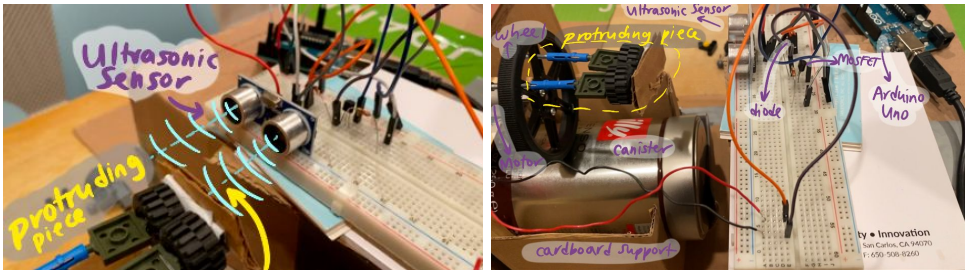

Figure 5: Ultrasonic Sensor setup and the connected circuit

(1) Brushed DC Motor (Pololu #3703) had the torque and angular velocity capabilities necessary for my application. One limitation is that I decided to use the Arduino's 5V power supply to run the motor which impacted the motor from running at top performance. As a result, I had to significantly reduce the weight in the canister during testing. Another limitation was that by having the protruding piece coming off of the wheel, the motor mounting bracket started bending like a cantilever beam, so I had to place a book on the motor to keep the wheel flat against the canister when turning. Going forward, I will need to modify the motor housing slightly to prevent this from happening for better engineering practice.
(2) Arduino Uno is the microcontroller I decided to use, due to the ability for it to run the motor with 5V power supply while also doing feedback control. I already owned one previously, which made it a good fit.
(3) Ultrasonic Sensor (HC-SR04) is the sensor I used to implement feedback control. It measured the distance away from it, as pictured in Figure 6, and could detect when the protruding piece made a revolution, as I mentioned earlier in Section 2.
(4) The Transistor (N-Channel MOSFET) and Diode (IN4148) were useful to get the system to run, where the diode allowed for the back-emf to dissipate safely and the transistor controlled the motor. In addition, I use a 2.2 kΩ resistor to run the transistor.

Section 5: Finite State Machine:

The behavior of the rock tumbler is illustrated below. As described in section 2, initially a calibration period is entered, after which the program continually takes measurements of RPM with the sonar. While running our loop, it checks what the user has entered. If the user sets the speed to 0 (which is also the initial default) it turns the motor off. Otherwise if the user has set a non-zero desired speed, but a measurement of RPM has not been made yet it sets the motor to a max speed. If the user has set a non-zero desired speed and the program has made at least one RPM measurement, then it can use its PI controller to manage the speed of the motor. Assuming the motor is not in the off state then every 10 minutes it will recalibrate.

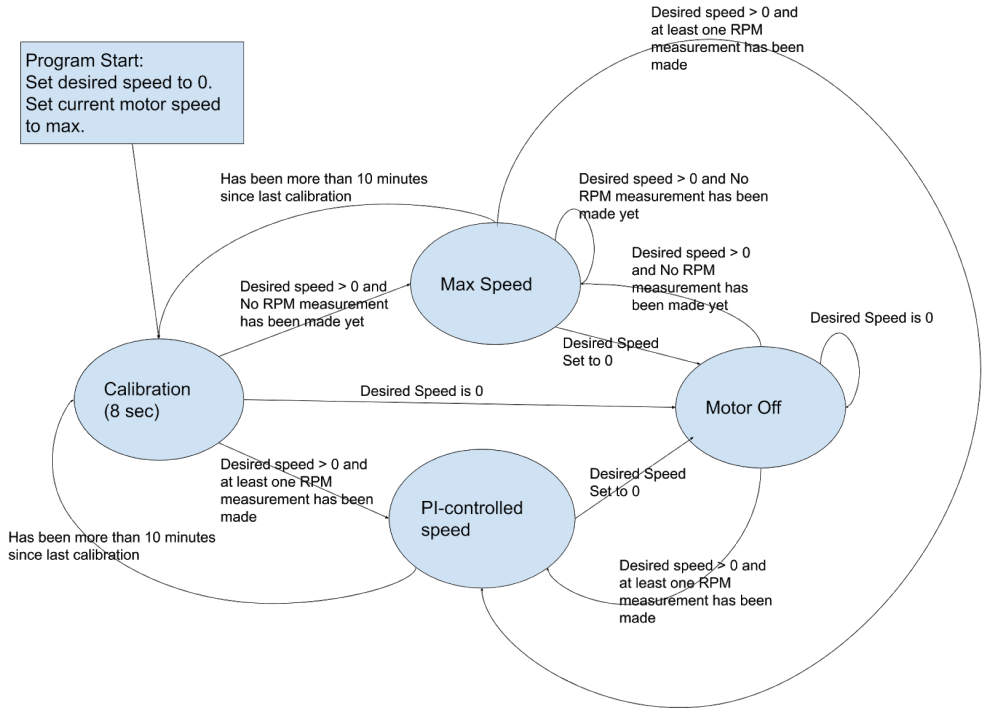

Figure 6: Finite State Diagram of the Rock Tumbler Performance

See Appendix A for Bill of Materials, Appendix B for complete Arduino IDE code, Appendix C for MATLAB.

Appendix A: Bill of Materials, containing purchased parts and reused parts

Name	Description/Use	Quantity
#3703 78:1 Metal Gearmotor 20Dx43L mm 6V CB	Motor tasked with turning wheel which turns canister	1
#3499 Magnetic Encoder Pair Kit for 20D mm Metal Gearmotors, 20 CPR, 2.7-18V	Would have been used to determine the actual RPM of the motor -- bought the wrong motor to use this part	1
#3690 Pololu Multi-Hub Wheel w/Inserts for 3mm and 4mm Shafts - 80×10mm, Black, 2-Pack	Wheel with rubber treads	1
#1077 Machine Screw: M3, 25mm Length, Phillips (25-pack)	Longer screws; primarily used to fasten down motor mounting bracket and to attach jutting out piece from wheel	6 used
#1069 Machine Hex Nut: M3 (25-pack)	Securing fasteners for motor mounting bracket	4 used
#1138 Pololu 20D mm Metal Gearmotor Bracket Pair	Mounting bracket for motor	1
Coffee Container	Cannister to house rocks and tumbling medium	1
HC-SR04 Ultrasonic Sonar Distance Sensor	Senses distance, used to determine when jutting-out piece from the wheel makes a complete revolution - can be used to emulate feedback control	1
IN4148 Diode	To provide a safe path for back-emf	1
N-Channel MOSFET Transistor	Transistor to drive the motor	1
2.2 kOhm Resistor	To put to the base of the transistor	1
Male-Male Wires	Connecting circuit together	Several
Lego Pieces	Making up the protruding piece on the wheel	Several
Cardboard	Support for the parts and connecting the protruding piece with a flat surface for the ultrasonic sensor	Several
Arduino Uno	Already had an Arduino Uno - used to run the program and use 5V power supply	1
Rock Tumbling Materials	Rocks and rock tumbling medium to simulate situation	Several

Appendix B: Arduino Code

```
#define echoPin 2
#define trigPin 3
#define motorPin 9

// calibration /revolution info
// “we” means me, the user, program, etc.
//we calibrate to determine "lowValue" and "highValue" which represent the approximate distance measurements for when the protruding element of the wheel is not in front of the sonar sensor and when it is.
const int MIN_CALIB = 10; //the minimum allowed calibration value
const int MAX_CALIB = 1500; //the maximum allowed calibration value
const int MAX_CALIB_DELTA = 200; //the max delta allowed between successive distance measures such that they are considered the same type (high or low)
const int NUM_IN_A_ROW_NEEDED = 8; //the number of successive measurements before we are certain this indeed a period of high/low distance
const unsigned long TIME_TO_SPEND_CALIBRATING = 8L*1000L; //how long to spend during the calibration step
int lowValue = 56; //default value -- subject to change post-calibration
int highValue = 388; //default value -- subject to change post-calibration
unsigned long lastCalibrationTime = millis(); //when was the last time we recalibrated?
const unsigned long howOftenCalibrate = 10L*60L*1000L; //how frequently to recalibrate

// measuring RPM info
int currHighDistanceCount =0; //how many "high" values (meaning values within delta of highValue) have we measured in a row from the sonar
int currLowDistanceCount  =0; //how many "low" values (meaning values within delta of lowValue) have we measured in a row from the sonar
const unsigned long MAX_FEASIBLE_RPM = 80; //noise filter -- we know that the motor with a rock tumbler realistically won't spin above this speed so we can filter out bad values
const unsigned long MIN_NEW_REV_DELTA = 500; //noise filter --ignore new measurements of a revolution with milliseconds between less than this value. Somewhat redundant with above, but we have second one because we want things that make it past this one but the not the previous one to count as a revolution for purposes of lastNewRevTime when calculating RPM
unsigned long lastNewRevTime = 0; //the last time we saw a revolution

//an array of 5 RPMS so that we can calculate the average of the last 5 RPMS for purposes of our PI controller
const int NUM_RPMS = 5;
int RPMS[NUM_RPMS] = {0};
int RPMS_index = 0; //this value will wrap around so we fill array like 0, 1,..n, 0, 1,,..n, etc.

bool atLeastOnceThrough = false; //true when we have found NUM_RPMS within our current user target, until this point we populate the missing values in the  array with the most recent values. This is set to false when the user enters a new RPM so that when the PI controller kicks in we do not bundle in old RPM values to our average calculation
bool haveEverGeneratedARPM = false; //until we generate at least one RPM (in the entire history of our program), this is set to false so we can choose a default value for speed

// target RPM and PI info
bool inLowZone = true; //toggle so we know whether we have just considered ourselves to have the protruding element far (true) or close (false)
int targetUserRPM = 0; //what the user wants the RPM to nbe
const int MIN_ALLOWABLE_INPUT_RPM = 50; //don't allow the user to enter in speeds that are too slow
const float PWM_MAX = 255;
const float PWM_MIN = 180; //values below this cause motor stall
const float KP = 2;
const float KI = KP/50;
float iTerm = 0;
unsigned long lastSpeedUpdate = millis(); //when was the last time we used our PI controller to update the motor speed?
```

```
const unsigned long HOW_OFTEN_UPDATE_SPEED = 500; //only update our motor speed using the PI controller at regular intervals

void setup() {
  analogWrite(motorPin, 0);
  Serial.begin(115200);
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
  Serial.println("calibrating...");
  analogWrite(motorPin, 255);
  calibrate();
  analogWrite(motorPin, 0);
}

void loop() {
    checkUserInput(); //check if user has entered in a new target RPM
    if (targetUserRPM == 0) {
      analogWrite(motorPin, 0);
      return;
    }
    checkCalibrate(); //check if it has been enough time since our last calibration that we need to recalibrate
    calculateAngularSpeed();
    unsigned long currTime = millis();
    if (!haveEverGeneratedARPM) {
      analogWrite(motorPin, 255);
      lastSpeedUpdate = currTime;
    }
    else if (((currTime - lastSpeedUpdate) > HOW_OFTEN_UPDATE_SPEED)) {
      setMotorSpeed();
      lastSpeedUpdate = currTime;
    }
}

void setMotorSpeed() {
  int currentRPM =0;
  for (int i = 0; i < NUM_RPMS; i++) {
      currentRPM += RPMS[i];
  }
  currentRPM /= NUM_RPMS;

  int error = targetUserRPM - currentRPM;
  iTerm += KP*KI*((float)error);
  iTerm = constrain(iTerm, PWM_MIN, PWM_MAX);
  float pTerm = KP*((float)error);
  int duty_value = constrain(pTerm+iTerm, PWM_MIN, PWM_MAX);
  analogWrite(motorPin, duty_value);
  Serial.print(millis());
  Serial.print(" ");
  Serial.print("Desired: ");
  Serial.print(targetUserRPM);
  Serial.print(" Actual: ");
  Serial.print(currentRPM);
  Serial.print(" Duty: ");
  Serial.print(duty_value);
  Serial.print(" Pterm: ");
  Serial.print(pTerm);
  Serial.print(" Iterm: ");
```

```
  Serial.println(iTerm);

}

void checkUserInput() {
  if (Serial.available() > 0) {
   int motorSpeed = Serial.parseInt();
   if (motorSpeed < MIN_ALLOWABLE_INPUT_RPM && motorSpeed > 0) {
     Serial.println("Motor speed is too low, must be at least 50 RPM or 0 for stop");
     return;
   }
   Serial.print("Updating motor speed to ");
   Serial.print(motorSpeed);
   Serial.println("RPMS");
   if (motorSpeed != targetUserRPM) {
     atLeastOnceThrough = false; //if the user has entered in a new RPM from before, then clear this flag so that when the PI controller
kicks in we do not bundle in old RPM values to our average calculation
   }
   targetUserRPM = motorSpeed;
  }
}

void calculateAngularSpeed() {
  int distance = getDistance();
  if (inLowZone && abs(distance - highValue) < MAX_CALIB_DELTA) { //if we currenty low and within delta from the high val
    currHighDistanceCount++;
    if (currHighDistanceCount >= NUM_IN_A_ROW_NEEDED) {
       triggerNewRev();
       inLowZone = false;
       return;
    }
  } else {
    currHighDistanceCount = 0;
  }

  if (!inLowZone && abs(distance - lowValue) < MAX_CALIB_DELTA) { //if we are currently high and within delta frmo the low value
    currLowDistanceCount++;
    if (currLowDistanceCount >= NUM_IN_A_ROW_NEEDED) {
      inLowZone = true;
      return;
    }
  } else {
    currLowDistanceCount = 0;
  }
}
```

// CONTINUED ON NEXT PAGE...

```
//function that is called every time our sonar sensor sees the protruding element
void triggerNewRev() {
  unsigned long currMeasuredTime = millis();
  unsigned long delta = currMeasuredTime - lastNewRevTime;
  if ( delta > MIN_NEW_REV_DELTA) {
    int RPM = (int)((1.0/((float)delta))*60.0*1000.0);
    if (lastNewRevTime != 0 && RPM <= MAX_FEASIBLE_RPM) {
      haveEverGeneratedARPM = true;
      Serial.print("rpm: ");
      Serial.println(RPM);
      if (!atLeastOnceThrough) {
        //until we've filled up the entire array with new measurements
        //populate the missing values with the most recently calculated values
        // (this in effect gives more weight to more recent measurements, until
        // we achieve our target number of measurements)
        for (int i =RPMS_index; i < NUM_RPMS; i++) {
          RPMS[i] = RPM;
        }
      } else {
        RPMS[RPMS_index] = RPM;
      }
      RPMS_index++;
      if (RPMS_index == NUM_RPMS) {
        RPMS_index = 0;
        atLeastOnceThrough = true;
      }
    }
    lastNewRevTime = currMeasuredTime;
  }
}


void checkCalibrate() {
  if ( (millis() - lastCalibrationTime) > howOftenCalibrate) {
    Serial.println(" recalibrating");
    calibrate();
  }
}


 // CONTINUED ON NEXT PAGE...
```

```
void calibrate() {
  unsigned long startTime = millis();
  unsigned long currTime = startTime;
  int count = 0;
  int curr = 0;
  int average = 0;
  while ( (currTime - startTime) < TIME_TO_SPEND_CALIBRATING) { //find min/max over 8 second range
    int distance = getDistance();
    currTime = millis();
    if (distance < 0) { //throw out garbage values
      continue;
    }

    //require NUM_IN_A_ROW_NEEDED values each within some delta of each other for us to consider it be a valid indicuation of
high/low
    if (abs(distance - curr) > MAX_CALIB_DELTA) {
      //we failed the check so reset everything
      count = 0;
      curr = distance;
      average = 0;
    } else {
      average += distance;
      count++;
    }
    if (count == NUM_IN_A_ROW_NEEDED) {
       average /= NUM_IN_A_ROW_NEEDED;
      lowValue = constrain(min(lowValue, average), MIN_CALIB, MAX_CALIB);
      highValue = constrain(max(highValue, average), MIN_CALIB, MAX_CALIB);
      average = 0;
    }
  }
  lastCalibrationTime = millis();
  Serial.print("re-calibrated!: " );
  Serial.print(lowValue);
  Serial.print(" ");
  Serial.println(highValue);
}

int getDistance() {
  //https://create.arduino.cc/projecthub/abdularbi17/ultrasonic-sensor-hc-sr04-with-arduino-tutorial-327ff6
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  // Sets the trigPin HIGH (ACTIVE) for 10 microseconds
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);
  // Reads the echoPin, returns the sound wave travel time in microseconds
  int duration = pulseIn(echoPin, HIGH);

  return duration;
}
```

Appendix C: MATLAB Code to Plot Data of example PI controller performance

```
%ME102B Anaylsis for Final Project

data_array = table2array(Dataset);

time = data_array(:,1);
time = time - time(1); %in milliseconds
desired_rpm = data_array(:,2); %in RPM
actual_rpm = data_array(:,3); %in RPM
duty_cycle = data_array(:,4); % in percentage

figure(1)
sgtitle('Performance of Rock Tumbler with PI Feedback Control')
subplot(2, 1, 1)
plot(time, actual_rpm)
hold on
plot(time, desired_rpm)
title('Angular velocity (actual vs desired) with respect to time');
xlabel('t (msec)');
ylabel('angular velocity (RPM)');
legend('actual RPM', 'desired RPM');
subplot(2,1,2)
plot(time, duty_cycle)
title('Duty cycle with respect to time');
xlabel('t (msec)');
ylabel('Duty cycle (8 bit resolution)');
```

Appendix D: More Images

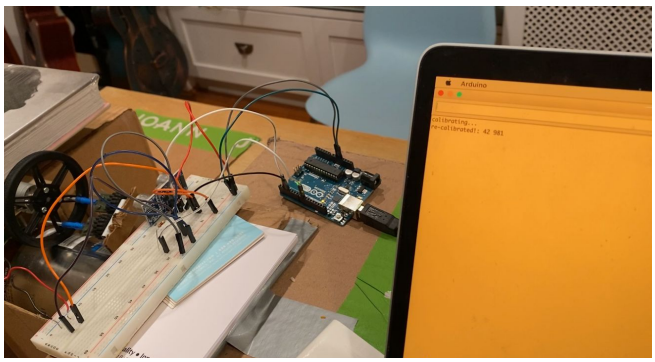

Figure 7: Calibration process image