Scratch Sumo-Bot: Tracking a Moving Object

Thai Nghiem, Russell Binaco
Section 3

May 7, 2018

1 Abstract

This project implemented a PID controller for a sumo robot object-follower. The sumo robot was designed to meet the Rowan Prof Bots sumo robot competition, and was implemented to have minimal overshoot, minimal steady-state error and a low settling time. The sumo bot was designed and built from scratch at a cost lower than that of the purchasable Zumo kit bots, and was implemented using an MSP430F5529 microprocessor. Ultimately, the sumo bot was able to follow a moving object at a fixed distance and account for turning, always stopping directly facing the object. Two identical PID controllers were used, one for each motor. The PID controllers used constants Kp = 3, Ki = 1, and Kd = 1.

2 Introduction

The purpose of this project was to implement an object-following sumo robot. The inspiration for this project came from the Annual Rowan Prof Bots competition, a sumo robot competition held by the Rowan Chapter of IEEE. The Zumo kit bots that can be purchased for this competition are driven by an Arduino, which was not permitted for use in this project, so the sumo bot had to be built from scratch.

The object-following behavior of the robot was to be implemented using a PID controller. Given a desired distance from an object, the sumo bot should detect its current distance from the object, and move to the required distance, directly facing the object. The turning capabilities of the bot are made possible by using two PID controllers rather than one; one controller was used for each motor.

There are four main sections to this report. The Standards and Constraints section addresses the standards that were followed when constructing the sumo bot, and the constraints that affected the implementation of the object-follower system. The Control Design section describes the characterization of the system and the design considerations used in the Systems and Control aspect of the object follower, and provides a block diagram of the PID system that was built. The Design Discussion section is a complete description of the implementation of the sumo bot, including the hardware and software components. The results and conclusions section describes the actual performance of the object follower system, and includes the individual contributions of the team members. An appendix is included for code listings. Also, videos of the sumo bot and the complete code files will be attached with this report submission.

3 Standards And Constraints

Standards and constraints are regulations and limitations surrounding a project. Taking standards into consideration before designing the sumo bot plays an important role in making the second-order design decisions and determining the success of the system.

3.1 Standards

Since the secondary purpose of the project is to compete in the Annual Rowan Prof Bots competition, the sumo bot strictly follows the rules of competition. According to the rules, the design constraints of the bot are:

- The robot must have maximum dimensions of 10cm wide and 10cm long.
- The robot must have a mass no greater than 500g
- Robots must be self-impelled and self-controlled

All design choices must consider and fall within these parameters. The full competition's rules can be found by clicking this website

3.2 Constraints

Many minor constraints have to be taken into considerations when building this sumo bot object follower, such as the stability, price, motor speed, distance, and available hardware. First of all, the system must not have a high percent overshoot, since it might cause a collision with the target object when there is a sudden change in position. Second of all, the total cost of the object follower must be relatively cheap (under 100 US Dollars), as there was very little money that was capable of being spent. Next, since the motor has a limited no-load speed of 400 RPM, the effect of the PID controller must not be too high, as it could make the bot go too slow. Lastly, it was a constraint to not use an Arduino processor in this project because it has a large library that can make the task seem trivial to implement.

4 Control Design

The task of following a moving object is inherently a Systems and Control task that requires feedback. The input to the system is a desired distance, so that distance must be transformed into a form that the microprocessor can use to compute motor speeds. In addition to system characterization, the PID controller should be tuned to a desired overshoot and steady-state error, as well as responsiveness. The ability of the system to reach a stable steady state should also be considered. This section of the report discusses the desired behavior of the system from a systems and control perspective, and section 6.1 discusses how the system behaved relative to these design considerations.

4.1 Characterizing the System

The system is characterized by converting the distance between the robot and the target object into a digital value using the analog-to-digital converter (ADC). First, the output of the IR sensor (PRP220) was measured at varying distances from the sumo bot. The results are seen in the table in Figure 1 below.

Distance (cm)	Left Voltage	Right Voltage	ADC3	ADC4
4	0.93	0.92	290	265
5	1.15	1.14	210	200
6	1.31	1.3	166	165
7	1.43	1.43	126	125
8	1.51	1.51	105	104
9	1.59	1.59	89	88
10	1.64	1.64	70	75
12	1.68	1.68	58	59
14	1.71	1.71	51	52
16	1.73	1.73	44	44
18	1.75	1.75	38	38
20	1.76	1.76	38	38
24	1.78	1.78	31	31

Figure 1: Characterization table.

The results show the successful detection of an object within 30cm, a distance long enough to trail behind an object. The table is converted into a graph as seen in Figure 2.

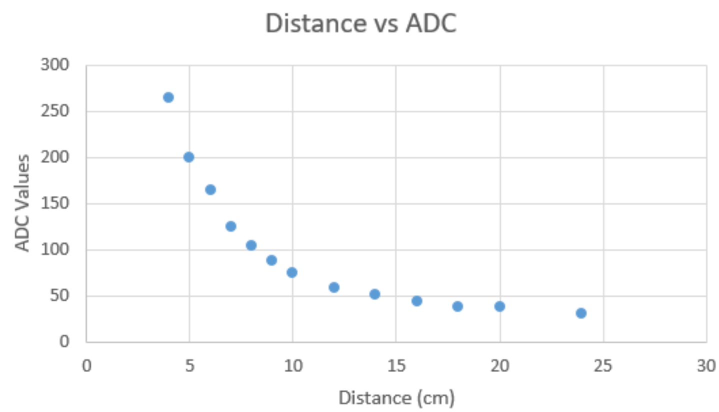

Figure 2: Distance vs. ADC graph.

This line is then divided into 3 pieces to create a piece-wise function using equations (1) and (2) shown below. This is because an exponential decay graph is more computationally intensive to characterize than a straight line, and a piecewise function still provides a sufficient approximation of the characterized system. The implementation of the characterization code will be discussed later in Section 5.

$$m = \frac{y_2 - y_1}{x_2 - x_1} \tag{1}$$

$$b = y_1 - x_1 \cdot m \tag{2}$$

4.2 Design Goals

For a PID controller, design goals relate to the steady-state error, responsiveness (i.e. settling time) and overshoot of the system. When following an object, little to no overshoot is desired since overshoot could cause the sumo bot to collide with the object, which may not be desired. Settling time and steady-state error should be minimized. Ultimately, the goal is to reach the desired distance from the object as quickly as possible, without overshooting that distance, and to minimize the oscillation and error at that distance.

4.3 Stability

By characterizing the system via the ADC readings from the sumo bot itself, the system is inherently stable. The only possible concern that could cause instability is a bad environment: for example, direct sunlight contains infrared light, which would cause the ADC readings to rise substantially and throwing off the characterization. Assuming the sumo bot is used indoors, only bad ADC readings from stray IR light or noise could cause a deviation in the system.

4.4 Control Loop Design

Figure 3 below shows the control loop for this system. Each portion of this loop was implemented as follows: First, the transformation from a desired distance to its equivalent ADC value was accomplished through the system characterization described above. The error signal comes from the difference between the average of the 10 most recent ADC readings and this characterized value. The implementation of the conversion from the error signal to a PWM duty cycle via the PID controller will be discussed in section 5.2. This process ultimately changes the position of the sumo bot, which is consistent with the initial set point also being a position.

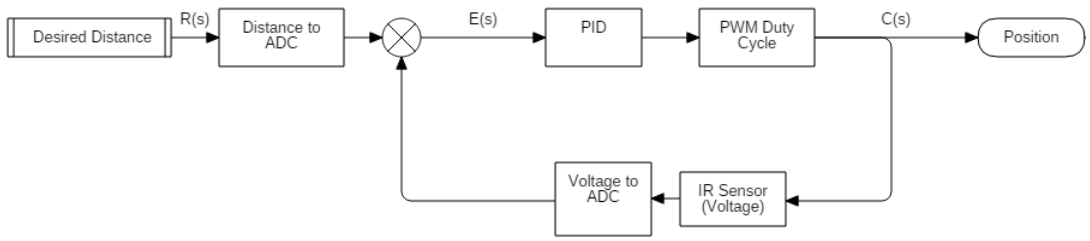

Figure 3: Block Diagram of PID System.

5 Design Discussion

5.1 Hardware

The sumo-bot was designed from scratch, with the advantage of being low cost, easy to adjust, and maintainable. The sumo-bot communicates with a MSP430F5529 using the built-in ADC to read values from sensors and control two motors using an H-bridge. Each of the mentioned hardware parts of the robot will be discussed in detail in the following subsections.

5.1.1 Micro-controller

Choosing the right micro-controller was the first design choice to made, as it determines the efficiency of the system. The MSP430F5529 was decided on for various reasons. First of all, the micro-controller fits within the 10cm x 10cm dimension constraint given by the Prof Bots competition. Also, the F5529 has 14 ADC channels (12 external at the device's pins and 2 internal), which is enough for full functionality and communication with the sensors. The F5529 is also faster than most of the other boards, including the MSP430G2553, when making decisions from sensor readings. Finally, familiarity with the 5529 from previous projects and reusability of relevant code also played a large part in the microprocessor decision.

5.1.2 PCB

Three versions of the PCB was designed for the sumo-robot, but only the second version was used in the final product. In the last version, which can be seen from Figure 4 of the PCB, there are 4 main sections.

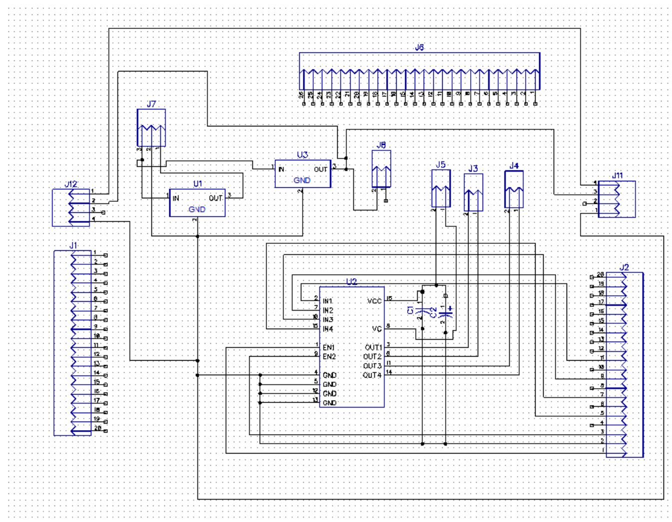

Figure 4: Schematic of breakout board used to power components.

The first section is the break-out board for the MSP430 to be mounted on. The second section is the H-bridge IC and the bulk and bypass capacitors that control the behavior of the 2 motors. The third section, which is on the top, has the two voltage regulators (5V and 3.3V). These are both used to supply power for the H-bridge and the MSP430. Finally, the last section, which is right next to the voltage regulator, is used for the IR sensors. In this section, the holes for the IR sensors are not connected to the pins of the MSP430, as to allow changes when actually building the robot.

The layout of the PCB can be seen in Figure 5

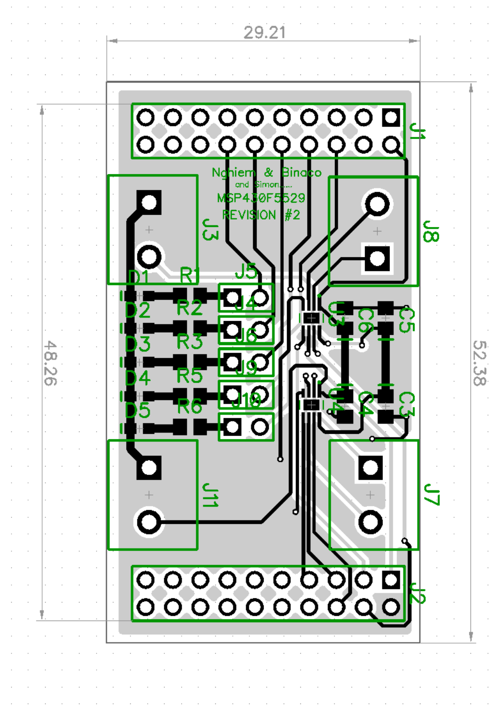

Figure 5: PCB Layout of the final product.

5.1.3 Housing

The next step was to design the chassis and board housing in Solidworks to be 3-D printed. The chassiss design is based off of the Zumo 32U Prime kit sumo bot. The adjustments made are to optimize the specific motors chosen and adding in additional through-holes for the plow and board attachment. The models are shown below in Figure 6.

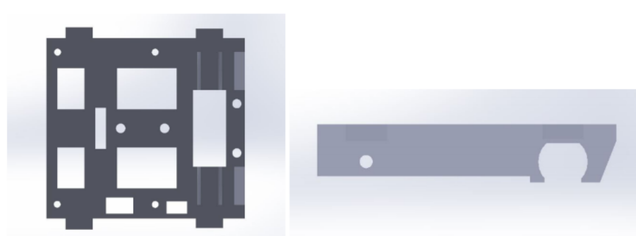

Figure 6: 3D Model of the Housing

The next step was to take the 3-D design and import it to the Ultimaker Cura Studio to be 3-D printed. The design took approximately 8 hours to print on print speed setting of 0.1 and infill of 50%.

5.1.4 Sensors

There were many challenges when finally implementing the sensor design of the scratch sumo bot. Initially, the infrared sensor (TSSP4056) needed a casing to be more accurate. However, it did not fit as well into the mechanical design of the bot, and the team was not satisfied with the performance of the TSSP4056. Upon further research of the old sumo bots used as references, the PRP220 IR sensor was discovered. The sensor was tested by the circuit as seen in Figure 7. This circuit was then implemented in a ProtoBoard. The sensor was able to detect changes between black and white as well as objects within 30cm very well.

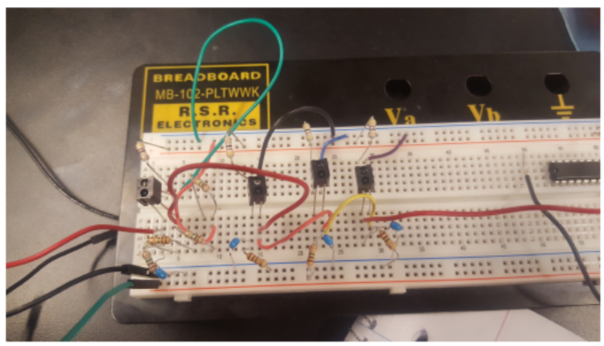

Figure 7: Picture of breadboard test setup with IR sensors.

5.1.5 Control Diagram

A general overview of the functioning of the system can be seen in Figure 8. A 12V energy source powers a 5V regulator, a 3.3V regulator, and the motors that are connected to the H-Bridge. The 3.3V regulator powers the infrared LEDs, which reflect off surfaces and into the nearby infrared sensors. The sensors send an analog voltage back into the MSP430, which converts the reading into an analog signal and compares it to thresholds that drive the logic of the H-bridge. Finally, the H-bridge controls the motors of the sumobot. The code is explained more in detail in the design approach.

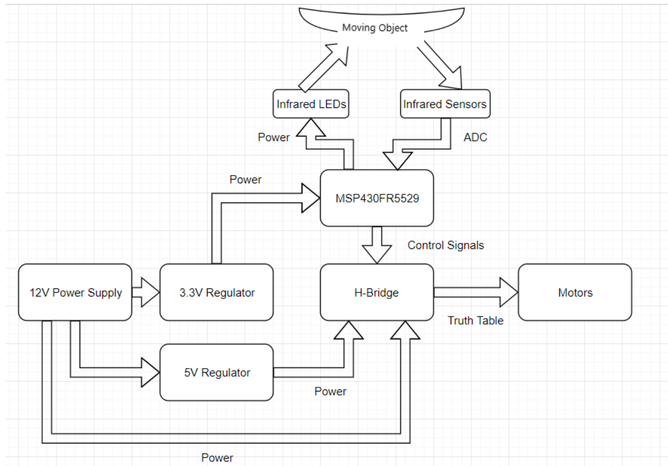

Figure 8: Diagram of the sumo-bot as a whole.

5.2 Software

5.2.1 ADC12

The analog to digital (ADC) conversion plays an important role in the design of the sumobot. The sensors output a voltage depending the distance of objects in front of them. The ADC interrupt was toggled in a while loop so the microprocessor is constantly checking the voltage readings from the sensors in order to make decisions based on the inputs.
The code for the ADC conversion that is used in sensing the environment can be seen below (with the ADC initialization omitted).

```
/*
 * Enable the ADC converter to start sensing the environment
 * This function is called in main
 */
void sensing(void)
{
    ADC12CTL0 |= ADC12ENC;          // ADC12 enabled
    ADC12CTL0 |= ADC12SC;           // Start sampling/conversion
    __bis_SR_register(GIE);        // LPM0, ADC12_ISR will force exit

}
```

5.2.2 Averager

Since the ADC values vary greatly between each sample and cause the system to be behave in an unexpected way, we implemented an averager in the ADC interrupt.

This averager takes the average of 10 ADC values and updates the PID controller according to this value. As a result, the system is much more stable and the robot behaves as expected. The code for the averager can be seen below.

```
// For every 10 ADC samples, we use 1 for stability of the system
        if(index < 10){     //adds new value to array for average of 10
            buf3[index] = ADC12MEM3; // buffer for right motor
            buf4[index] = ADC12MEM4;
            index++;
        }
        else{     //computes average of 10 and transmits value; resets arra
            long average3 = 0;
            long average4 = 0;
            int i = 0;
            for(i = 0; i < 10 ; i ++){
                average3 += buf3[i];
                average4 += buf4[i];
            }
            average3 /= 10;
            average4 /= 10;
            index=0;
            flag=1;
            // -120 for Full vs Weak
            adc3 = average3 ; //changes duty cycle
            adc4 = average4 ; //changes duty cycle

            // Calling the PID controller function
            updatePID();
```

5.2.3 PWM - Motor Speed

Pulse width modulation is used to control the speed of each motor on the sumo-bot. In order for this to be achieved, two PWM signals were output to Pin 1.4 and Pin 2.4. Pin 1.4 controlled the speed of the right motor and Pin 2.4 controlled the speed of the left motor. TA0CCR3 was used to control the right motor and TA2CCR1 was used to control the left motor. The PWM cycle is controlled using values 0-999. Setting TA0CC3/TA0CCR0 = 0 will produce a duty cycle of 0% and setting TA0CCR3/TA0CCR0 = 999 will produce a duty cycle of 100%.
The code to set the speed of motor using PWM can be seen below (with the PWM initialization being omitted).

```
/*
 * This function set the PWM value of the
 * two motors, hence control their speed
 */
void setMotor(int left, int right)
```

```
{
    if(right < 0) //Backwards, clock-wise
    {
        P2OUT |= BIT5; //Sets direction for H-bridge
        TA0CCR3 = (left * -1); //Speed of the motor CCR/1000
    }
    else if(right == 0) //stopping
    {
        TA0CCR3 = 0;
    }
    else //Forwards, counter clock-wise
    {
        P2OUT &= ~BIT5; //Sets direction for H-bridge
        TA0CCR3 = left; //Speed of the motor CCR/1000
    }
                .....
        [same code is done for the left]
}
```

5.2.4 Characterization

As previously discussed, the distance between the sumo-bot and the target object is characterized into ADC values as to control the PWM of the 2 motors. This is done by turning the graph in Figure 2 into 3 straight line graphs, as shown in Figure 9. The first graph represents the ADC value versus a distance ranging from 0 to 7 cm. The second graph represents the ADC value versus a distance ranging from 7 to 12 cm. The last graph represents the ADC value versus a distance ranging from 12 to 30 cm.

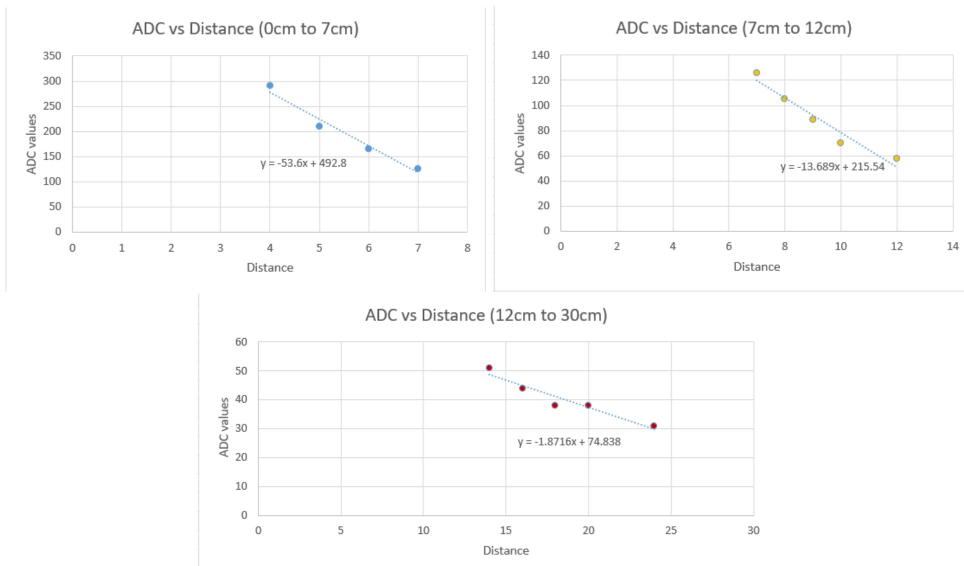

Figure 9: 3 straight line graphs derived from 1 curved line.

These graphs are then utilized to create 3 piecewise functions using Equations (1) and (2). The 3 functions are implemented in the code as follows to give us the equivalent ADC values.

```
/*
 * Characterization code for the system
 * Convert the desired distance to ADC value
 * Using piece-wise functions
 */
void distToADC(int distance){
     // 0cm to 7cm range
     if (distance >= 0 && distance <= 7){
          adc_out = distance*-53.6 + 492.8;
     // 7cm to 12cm range
     } else if (distance > 7 && distance <= 12){
          adc_out = distance*-13.689 + 215.54;
     // 12cm to 30cm range
     } else if (distance > 12 && distance  <= 30){
          adc_out = distance*-1.8716 + 74.838;
     }
     // Default value
     else{
          adc_out = 290;
     }
}
```

Hence, when the user inputs a desired distance of 5 cm away from the target object, the above function will assign it into the first if-statement, and produce an equivalent ADC value that will control the speed of the 2 motors.

5.2.5 PID Controller

The PID control was implemented using field variables (to maintain their value between function calls) and the updatePID function below. Within the function, the proportional, integral and derivative terms are calculated and summed. Also, there are two PID controllers, one for the left motor and one for the right motor. For the derivative term, the previous error and the current error are stored. The proportional term is simply the error multiplied by the proportional constant, Kp. The derivative term is the derivative constant, Kd, multiplied by the difference of the previous error and the current error. Lastly, the integral term is a running sum of the error. To prevent integral windup, this term is windowed between +/- 50. When summed, the derivative term is negative.

```
/*
 * Proportional - Integral - Derivative Controller of the robot
 * Is called in the Interrupt
 */
void updatePID(void){
    // Calculating the Error term
    error3Prev = error3; // Right motor
    error4Prev = error4; // Left motor
    error3 = desired - adc3;
    error4 = desired - adc4;

    // Calculating the Proportional term
    proportional3 = Kp * error3; // Right motor
    proportional4 = Kp * error4; // Left motor

    // Calculating the Derivative term
    derivative3 = Kd * (error3Prev-error3); // Right motor
    derivative4 = Kd * (error4Prev-error4); // Ledt motor

    // Calculating the accumulator for the Error term
    accumulator3 = accumulator3 += error3;
    accumulator4 = accumulator4 += error4;

    // Capping accumulator values
    if(accumulator3>50){ // Right motor
        accumulator3 = 50;
    }else if(accumulator3<-50){
        accumulator3 = -50;
    }
```

```
        if(accumulator4>50){   // Left motor
            accumulator4 = 50;
        }else if(accumulator4<-50){
            accumulator4 = -50;
        }

        // Calculating the Integral term
        integral3 = Ki * accumulator3; // Right motor
        integral4 = Ki * accumulator4; // Left motor

        // Adding the Proportional - Integral - Derivative temrs
        // up to create the PID controller
        pid3 = proportional3+integral3-derivative3;
        pid4 = proportional4+integral4-derivative4;

        //convert pid range to PWM range
        setScaledMotor(pid3,pid4);
}
```

6 Results and Conclusions

6.1 System Behavior

This section describes the performance of the object-following system, first with only a proportional term (Kd and Ki = 0) and then with a tuned PID controller, as well as the performance at steady state. Figure 12 below shows the sumo bot with its target object.

6.1.1 Proportional Control

For proportional control, the constant Kp was tuned to be 3. With proportional control only, the sumo bot was able to successfully follow a moving object including turning, but the performance of the system did not meet the design goals. While the bot moved quickly towards the object from a long distance away as desired, there was significant overshoot when the bot passed the desired distance from the object. Also, the bot continued to oscillate in forwards and backwards motion significantly around the desired distance. This was expected to occur, and the derivative and integral terms were then added.

6.1.2 PID Control

With a proportional, integral and derivative term, the system characteristics improved significantly. The sumo bot's approach towards the object was initially fast, and slowed

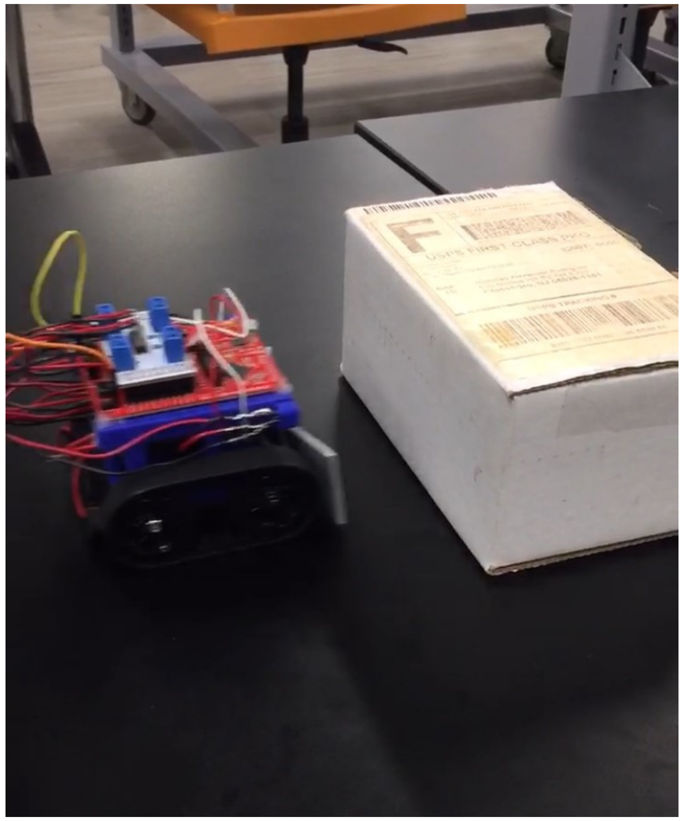

Figure 10: Sumo-bot and its object

as it got closer to the desired distance, but did not noticeably overshoot. As such, the bot reached steady state almost as soon as it reached the desired distance, stopping in front of the object. Overall, the system response (i.e. settling time) was slightly slower, but the reduced overshoot and steady state error more than make up for this in terms of the design goals.

6.1.3 Steady State and Errata

Occasionally, once the bot reached a stationary steady state, it would turn slightly and then oscillate rotationally from facing directly forwards. This was attributed to a bad sensor reading due to noise or incoming IR light. Noise reduction had already been attempted via the averaging of the ADC values, but a better solution may be a median filter rather than an averaging filter. This could be addressed as future work. However, after a bad reading, the bot was able to correct itself as the rotational oscillation decreased after the erroneous movement.

Also, an issue was encountered where the characterization of the system itself was dependent on the strength left in the batteries that powered the sumo bot. Figure 11 shows the relationship between the system characterization using fresh batteries and batteries that had been used in testing.

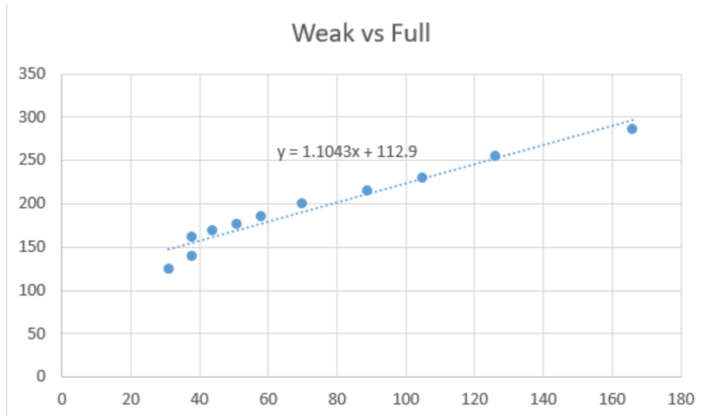

Figure 11: ADC value between low battery and full battery.

This relationship is nearly linear, which is useful since the system characterization will only be off by a fixed offset rather than being completely erroneous. What this means for the system is that the desired set point may not be accurately converted, and the steady state distance will not be correct. Since there is no way for the code itself to determine how much power is left in the batteries, this problem cannot be resolved. The issue was speculated to be due to the H-bridge design and how power is provided to the system. A solution would require a redesign of the hardware and a rebuild of the sumo bot.

6.2 Conclusions

Ultimately, the sumo bot object following system was built successfully. The bot was constructed from scratch, including 3D-printed parts and PCB designs, and met the specifications and standards for the Rowan IEEE Prof Bots competition. The distance of the sumo bot to an object was characterized into ADC values. Code for controlling the motor speed of the bots was written, as well as a software implementation of a PID controller. The system response met the design goals, and the cost of building the sumo bot is less than the price of an Arduino-driven, pre-built bot. Figure 12 shows the completed bot.

6.3 Contributions

The team always worked together throughout the process of making the sumo-bot object follower. Almost all of the work on the project was done while meeting together,

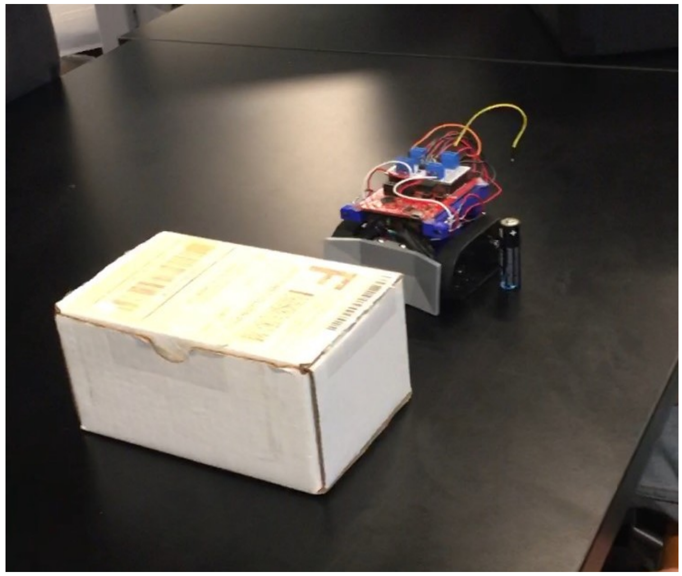

Figure 12: Sumo-bot and its object.

and each member contributed equally. However, each member has his own specialty and can be broken down into different categories seen in the following table.

Thai Nghiem	Russell Binaco
Project Proposal	PID Controller Research
Lead Hardware Designer	Lead Software Designer
System Characterization	Battery Sponsor

Acknowledgement: The authors would like to thank Dr. Al-Quzwini, Russell Trafford and Simonas Bublis for giving us advice and support throughout the project.

7 Appendix A

```
#include <msp430.h>
//for averaging ADC values
int adc3 = 0;
int adc4 = 0;
int buf3[10];
int buf4[10];
unsigned int index;

int Kp=3;
int Kd=0;
int Ki=0;
int error3 = 0;
int error4 = 0;
int error3Prev = 0;
int error4Prev = 0;
int proportional3 = 0;
int proportional4 = 0;
int derivative3 = 0;
int derivative4 = 0;
int accumulator3 = 0;
int accumulator4 = 0;
int integral3 = 0;
int integral4 = 0;
int pid3 = 0;
int pid4 = 0;
int tempLeft = 0;
int tempRight = 0;
int adc_out=0;
int scale_value = 8;

int control_type = 0;
int flag = 0;
int desired = 0; //ADC value
void PWMInit(void);
void sensing(void);
void starting(void);
void setMotor();
void ADCInit(void);
void updatePID(void);
void distToADC(int distance);
void setScaledMotor(int left, int right);
void pinInit(void);


/**
 * main.c
 */
int main(void)
{
    // Stop watchdog timer
    WDTCTL = WDTPW | WDTHOLD;

    // Initialization
    pinInit();
```

```
    PWMInit();
    ADCInit();

    // Reset the speed of both motors
    setMotor(0,0);

    /*
    *   Compute desired value in ADC
    *   Max: 200 (closest)
    *   Min: 100 (furthest
    *
    *   desired = 200;
    */

    // Desired trailing distance
    int distance = 5; // in centimeters
    distToADC(distance); // Distance to ADC value

    // Sumo-bot constantly sense the environment
    while(1) //inifite loop
    {
        sensing();
    }
}


void starting(void)
{
    sensing();
}

/*
 * Enable the ADC converter to start sensing the environment
 * This function is called in main
 */
void sensing(void)
{
    ADC12CTL0 |= ADC12ENC;        // ADC12 enabled
    ADC12CTL0 |= ADC12SC;         // Start sampling/conversion
    __bis_SR_register(GIE);      // LPM0, ADC12_ISR will force exit

}

/*
 * Proportional - Integral - Derivative Controller of the robot
 * Is called in the Interrupt
 */
void updatePID(void){
    // Calculating the Error term
    error3Prev = error3; // Right motor
    error4Prev = error4; // Left motor
    error3 = desired - adc3;
    error4 = desired - adc4;

    // Calculating the Proportional term
    proportional3 = Kp * error3; // Right motor
    proportional4 = Kp * error4; // Left motor
```

```
    // Calculating the Derivative term
    derivative3 = Kd * (error3Prev-error3); // Right motor
    derivative4 = Kd * (error4Prev-error4); // Ledt motor

    // Calculating the accumulator for the Error term
    accumulator3 = accumulator3 += error3;
    accumulator4 = accumulator4 += error4;

    // Capping accumulator values
    if(accumulator3>50){ // Right motor
        accumulator3 = 50;
    }else if(accumulator3<-50){
        accumulator3 = -50;
    }

    if(accumulator4>50){  // Left motor
        accumulator4 = 50;
    }else if(accumulator4<-50){
        accumulator4 = -50;
    }

    // Calculating the Integral term
    integral3 = Ki * accumulator3; // Right motor
    integral4 = Ki * accumulator4; // Left motor

    // Adding the Proportional - Integral - Derivative temrs
    // up to create the PID controller
    pid3 = proportional3+integral3-derivative3;
    pid4 = proportional4+integral4-derivative4;

    //convert pid range to PWM range
    setScaledMotor(pid3,pid4);
}

/*
 * This function scale the PWM range, since the left motor is
 * slightly stronger than the right motor, making the robot
 * steer right.
 */
void setScaledMotor(int left, int right){
    //left limit -999 to 999
    //right limit -870 to 870
    //expected range -200 to 200
    tempLeft = (left*5/2)*scale_value;
    tempRight = ((right*9)/4)*scale_value;

    // 30 is minimum value (furthest)
    if (tempLeft < 30 && tempLeft > 0){
        tempLeft = 30;
    }
    if(tempRight < 30 && tempRight > 0){
        tempRight = 30;
    }

    if (tempLeft > -30 && tempLeft < 0){
        tempLeft = -30;
```

```
    }
    if(tempRight > -30 && tempRight < 0){
        tempRight = -30;
    }
    // 999 is maximum value (closest)
    if(tempLeft< -999){
        tempLeft = -999;
    }
    if(tempRight< -999){
        tempRight = -999;
    }

    if (tempLeft>999){
        tempLeft = 999;
    }
    if (tempRight>870){
        tempRight = 870;
    }
    //        Right        Left
    setMotor(tempLeft, tempRight);

}

/*
 * This function set the PWM value of the
 * two motors, hence control their speed
 */
void setMotor(int left, int right)
{
    if(right < 0) //Backwards, clock-wise
    {
        P2OUT |= BIT5; //Sets direction for H-bridge
        TA0CCR3 = (left * -1); //Speed of the motor CCR/1000
    }
    else if(right == 0) //stopping
    {
        TA0CCR3 = 0;
    }
    else //Forwards, counter clock-wise
    {
        P2OUT &= ~BIT5; //Sets direction for H-bridge
        TA0CCR3 = left; //Speed of the motor CCR/1000
    }

    if(left < 0)
    {
        P1OUT |= BIT5;
        TA2CCR1 = (right * -1);
    }
    else if(left == 0)
    {
        TA2CCR1 = 0;
    }
    else // forwards, clock-wise
    {
        P1OUT &= ~BIT5;
        TA2CCR1 = right;
```

```c
    }
}

/*
 * Characterization code for the system
 * Convert the desired distance to ADC value
 * Using piece-wise functions
 */
void distToADC(int distance){
    // 0cm to 7cm range
    if (distance >= 0 && distance <= 7){
        adc_out = distance*-53.6 + 492.8;
    // 7cm to 12cm range
    } else if (distance > 7 && distance <= 12){
        adc_out = distance*-13.689 + 215.54;
    // 12cm to 30cm range
    } else if (distance > 12 && distance  <= 30){
        adc_out = distance*-1.8716 + 74.838;
    }
    // Default value
    else{
        adc_out = 290;
    }
}
void pinInit(void){
    P2DIR |= BIT0;  // Pin 2.0 initialization
    P2OUT |= BIT0;

    P2DIR |= BIT2;  // Pin 2.2 initialization
    P2OUT |= BIT2;

    P2DIR |= BIT5;
    P1DIR |= BIT5;

    P1SEL =0; //Select GPIO option
    P1DIR |=BIT0; //set Port 1.0 output --LED
    P1OUT &= ~BIT0;  // LED OFF

    P4SEL =0; //Select GPIO option
    P4DIR |=BIT7; //set Port 4.7 output --LED
    P4OUT &= ~BIT7;  // LED OFF

    P1DIR &=~(BIT1); //set Port 1.1 input -- pushbutton
    P1REN|=BIT1;//enable pull-up/pull-down resistor on
    P1OUT|=BIT1; //choose the pull-up resistor

    P1IE |=BIT1;//enable the interrupt on Port 1.1
    P1IES |=BIT1;//set as falling edge
    P1IFG &=~(BIT1);//clear interrupt flag
}
void PWMInit(void)
{
    P1DIR |= BIT4;      // Initialize PWM to output on P1.4
    P1SEL |= BIT4;

    P2DIR |= BIT4;      // Initialize PWM to output on P2.4
    P2SEL |= BIT4;
```

```

    TA0CCR0 =1000−1;
    TA2CCR0 =1000−1;
    TA0CCTL3 =OUTMOD_7;
    TA0CCR3 =999;
    TA2CCTL1 =OUTMOD_7;
    TA2CCR1 = 999;
    TA0CTL = TASSEL_2 + MC_1 + TACLR;
    TA2CTL = TASSEL_2 + MC_1 + TACLR;
}

void ADCInit(void)
{
    ADC12CTL0 = ADC12ON+ADC12MSC+ADC12SHT02; // Turn on ADC12, set sampling
     time
    ADC12CTL1 = ADC12SHP+ADC12CONSEQ_1;       // Use sampling timer, single
     sequence
    ADC12MCTL0 = ADC12INCH_0;                 // ref+=AVcc, channel = A0
    ADC12MCTL1 = ADC12INCH_1;                 // ref+=AVcc, channel = A1
    ADC12MCTL2 = ADC12INCH_2;                 // ref+=AVcc, channel = A2
    ADC12MCTL3 = ADC12INCH_3;
    ADC12MCTL4 = ADC12INCH_4 + ADC12EOS;        // ref+=AVcc, channel = A3,
     end seq.
    ADC12IE = 0x10;                           // Enable ADC12IFG.3
    ADC12CTL0 |= ADC12ENC;                    // Enable conversions
    /*Sets ADC Pin to NOT GPIO*/
    P6SEL |= BIT1 + BIT2 + BIT3 + BIT4;                        // P6.0 ADC
      option select
    P6DIR &= ~(BIT1 + BIT2 + BIT3 + BIT4);
    P6REN |= BIT1 + BIT2 + BIT3 + BIT4;
    P6OUT &= ~(BIT1 + BIT2 + BIT3 + BIT4);
    ADC12CTL0 |= ADC12ENC;
    ADC12CTL0 |= ADC12SC;
}

#if defined(__TI_COMPILER_VERSION__) || defined(__IAR_SYSTEMS_ICC__)
#pragma vector = ADC12_VECTOR
__interrupt void ADC12_ISR(void)
#elif defined(__GNUC__)
void __attribute__ ((interrupt(ADC12_VECTOR))) ADC12_ISR (void)
#else
#error Compiler not supported!
#endif
{

    switch(__even_in_range(ADC12IV,34))
    {
    case  0: break;                           // Vector  0:  No interrupt
    case  2: break;                           // Vector  2:  ADC overflow
    case  4: break;                           // Vector  4:  ADC timing
     overflow
    case  6: break;                                  // Vector  6:  ADC12IFG0
    case  8:break;                           // Vector  8:  ADC12IFG1
    case 10: break;                           // Vector 10:  ADC12IFG2
    case 12:break;                           // Vector 12:  ADC12IFG3
    case 14:
        // For every 10 ADC samples, we use 1 for stability of the system
```

```
        if(index < 10){      //adds new value to array for average of 10
            buf3[index] = ADC12MEM3; // buffer for right motor
            buf4[index] = ADC12MEM4;
            index++;
        }
        else{     //computes average of 10 and transmits value; resets array
    index
            long average3 = 0;
            long average4 = 0;
            int i = 0;
            for(i = 0; i < 10 ; i ++){
                average3 += buf3[i];
                average4 += buf4[i];
            }
            average3 /= 10;
            average4 /= 10;
            index=0;
            flag=1;
            // -120 for Full vs Weak
            adc3 = average3 ; //changes duty cycle
            adc4 = average4 ; //changes duty cycle

            // Calling the PID controller function
            updatePID();
        }

        __bic_SR_register_on_exit(LPM0_bits);
        break;                            // Vector 14:  ADC12IFG4
    case 16: break;                           // Vector 16:  ADC12IFG5
    case 18: break;                           // Vector 18:  ADC12IFG6
    case 20: break;                           // Vector 20:  ADC12IFG7
    case 22: break;                           // Vector 22:  ADC12IFG8
    case 24: break;                           // Vector 24:  ADC12IFG9
    case 26: break;                           // Vector 26:  ADC12IFG10
    case 28: break;                           // Vector 28:  ADC12IFG11
    case 30: break;                           // Vector 30:  ADC12IFG12
    case 32: break;                           // Vector 32:  ADC12IFG13
    case 34: break;                           // Vector 34:  ADC12IFG14
    default: break;
    }
}

/*
 * Button Interrupt that control the Controller Type of the robot
 * There are 4 modes in total: only Proportional, Proportional-Integral,
 * Proportional-Derivative, and all Proportional-Integral-Derivative
 */
#pragma vector=PORT1_VECTOR
__interrupt void PORT_1(void)
{
    P1IE &= ~BIT1; // Disable interrupt

    //Debounce 1
    __delay_cycles(1);

    //Debounce 2
    TA1CTL = TASSEL_1 + MC_1 + ID_1; //Set up Timer A, Count up, divider 2
```

```
    TA1CCTL0 = 0x10; // Set up compare mode for CCTL
    TA1CCR0 = 2000; // Duration at which the interrupt is disable
                    // Duration 2000/16kHz = 1/8 sec.
    P1IFG &=~(BIT1); // Clear flag

    // Loop back to control type 0
    if (control_type > 3){
        control_type = 0;
    }

    switch(control_type){
    case 0: //Only PI
        P4OUT |= BIT7; // Turn 4.7 on for 01
        P1OUT &= ~BIT0; // Turn 1.0 on
        Ki =1;
        scale_value = 5;
        break;
    case 1: // PD
        P1OUT |= BIT0; // Turn 1.0 on for 10
        P4OUT &= ~BIT7; // Turn 4.7 off
        Ki = 0;
        Kd = 1;
        scale_value = 8;
        break;
    case 2: //PID
        P4OUT |= BIT7; // Turn 4.7 on for 11
        P1OUT |= BIT0; // Turn 1.0 on for
        Ki = 1;
        Kd = 1;
        scale_value = 1;
        break;
    case 3: //Only P
        P4OUT &= ~BIT7; // Turn 4.7 on for 11
        P1OUT &= ~BIT0; // Turn 1.0 on for
        Ki = 0;
        Kd = 0;
        scale_value = 8;
        break;
    default:
        break;
    }
    control_type += 1;


}
#pragma vector=TIMER1_A0_VECTOR
__interrupt void Timer_A0(void)
{
    P1IE |= BIT1; //Enable interrupt again.
}
```