ILR01: Sensor and Motor Control Lab

Maitreya Naik

Team B: Arcus

Teammates: Logan Wan, Clare Cui, Angad Sidhu

ILR01

Oct. 14, 2016

1. Individual Progress

For the Sensors and Motor Control Lab, I was responsible for setting up the DC Motor Velocity and Position Control by taking inputs from the Potentiometer. I was also responsible for including the push button and de-bouncing code in the circuit, and adding an LED to signal the button push registration. I was also responsible for most of the final electronic integration of the components. The final system picture is in Figure 1.

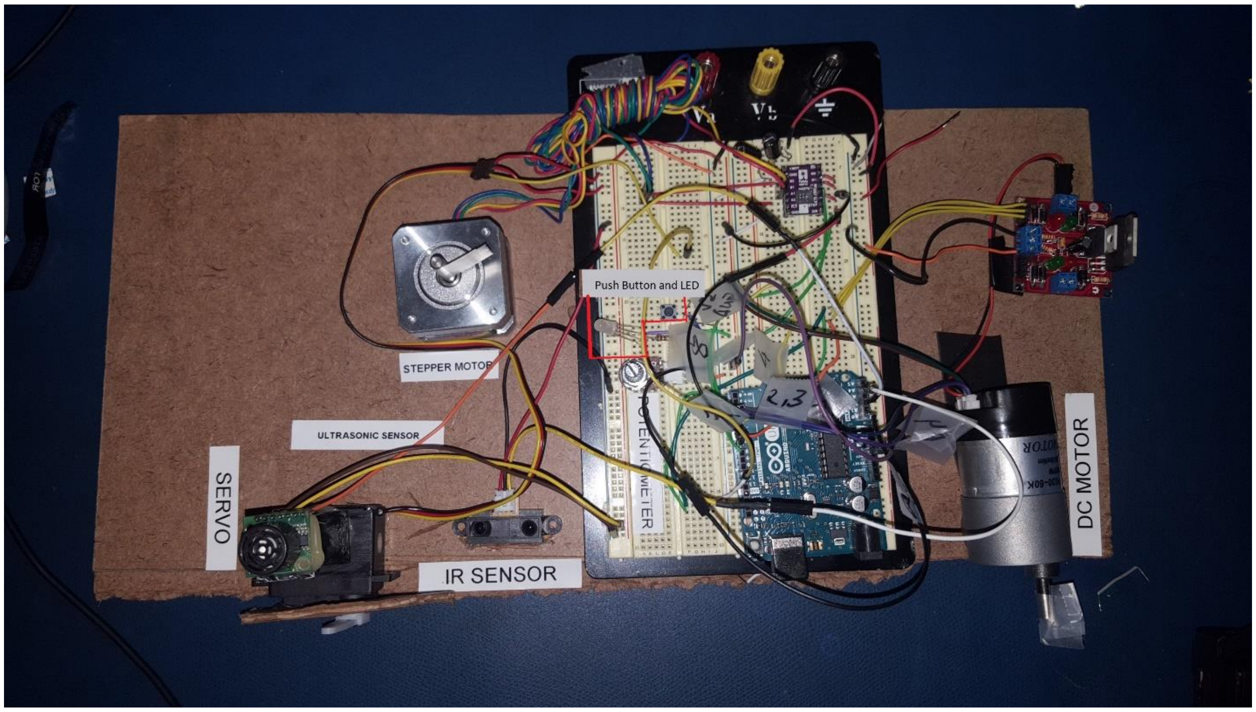

Figure 1: Final System *[Picture by Clare Cui]*

1.1. Potentiometer

The Potentiometer values were read from an Analog pin on the Arduino and an Average Filter was implemented on the incoming Potentiometer values to receive a reliable input.

For the DC Motor Velocity control, the filtered Potentiometer value was directly mapped to a velocity set point according to Table 1.

For the DC Motor Position control, the initial Potentiometer reading is recorded. For a change of Potentiometer value, a mapping to a set angle is done according to Table 2.

1.2. DC Motor Driver

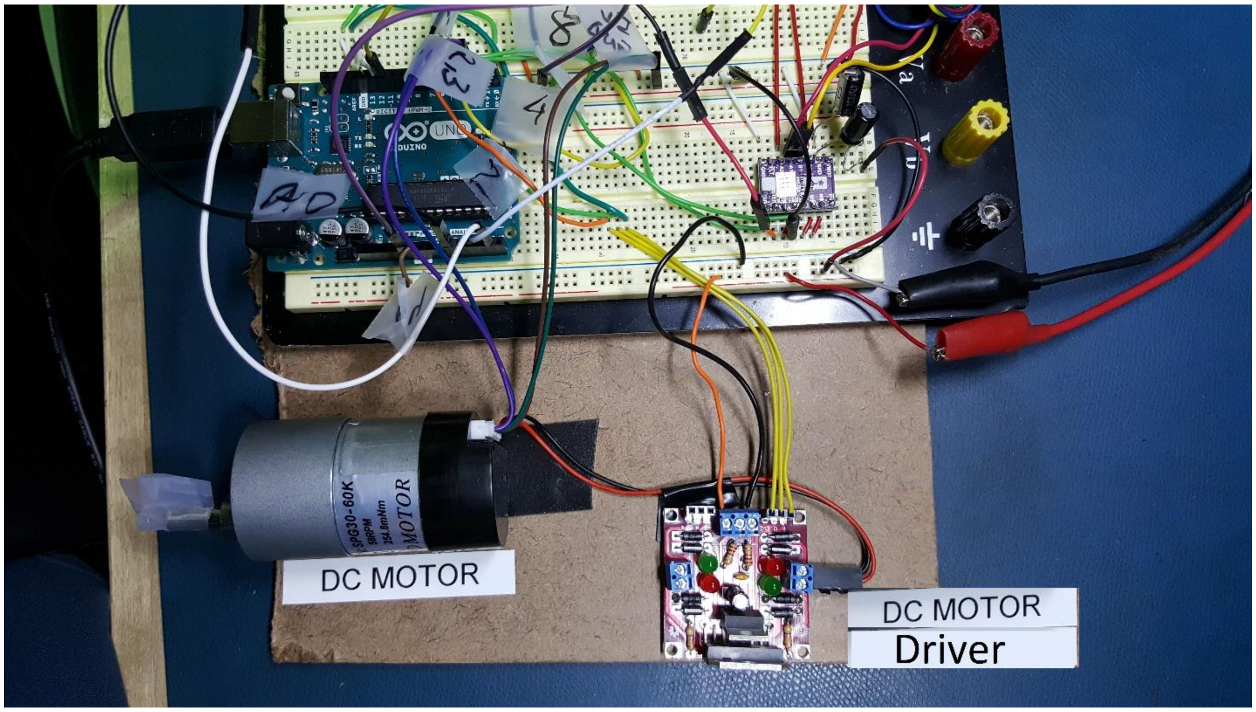

Figure 2: DC Motor and Driver with the Arduino

The DC Motor (Cytron SPG30-60K) is rated to operate at 12V, and draw 90mA while free running and up to 1800mA when stalled [1]. Due to this high current possibility, a Solarbotics DC Motor Driver [2] was used to implement Arduino Logic commands in directing the Motor movements and controlling its speed through PWM from the Arduino. Moreover, the DC Motor was run at a higher voltage (within maximum voltage of 15V) to ensure that the Motor starts up from rest for smaller PWM values. Consequently the motor ran at 75 RPM instead of the rated 58 RPM.

1.3. DC Motor

The DC motor (Cytron SPG30-60K) was driven in 2 different states for the lab. They were:

1. Velocity Control
2. Position Control

For the velocity control, the velocity in RPM was set by the Voltage value received using the potentiometer according to the following linear mapping using the Arduino "map" function.

Table 1: Velocity Control Map

Potentiometer		Set Velocity
AnalogRead	Voltage	
0	0V	-75 RPM
1023	5V	75 RPM

For the position control, the input was a relative angle movement from the last angle movement. Hence, the mapping was done from a difference between last potentiometer voltage and current potentiometer voltage.

Table 2: Position Control Map

Potentiometer Difference		Set Angle Movement
AnalogRead	Voltage	
-1023	-5V	-179˚
1023	5V	180˚

For both types of controls, a PID controller was implemented using software with input from the Motor encoders.

For the velocity control, the calibrated parameters were K_p =0.6, K_i =0.02, and K_d=0.1. Resulting in a rise time of about 2secs, and a maximum overshoot of 2%

For the position control, the calibrated parameters were K_p =1, K_i =0.01, and K_d=1.2. Though this resulted in multiple overshoots before stopping at the desired position. This issue is further explained under "Challenges".

The process of PID control was the following:

1. Acquire Set Point from Potentiometer readings.
2. Update Motor Commands:
    a. Measure current value (velocity or position) from Encoder Counts and Time.
    b. (*for Position Control only*) - If Motor is in allowable margin of SetPoint:
        i. Stop Motor.
    c. [PWM, Direction]= PID(SetPoint, Measured Value).
    d. Send PWM and direction command to motor.
3. Return to Step 1

1.4. Encoders

The encoders attached to the DC Motor [1], along with the Arduino "millis()" function were used to measure the DC Motor Velocity and Position. This was then provided as feedback to the PID Controller. The encoders occupied both the interrupt pins (2 & 3) on the Ardunio Uno.

Using the Encoder library by Paul Stoffregen [3], reliable Encoder counts could be retrieved from interrupt pins and reset to calculate the motor RPM as:

$$RPM = (\Delta n) * 60/(\Delta t)/(N_r)$$

Where, $\Delta n = Difference\ between\ successive\ encoder\ counts$

$\Delta t = Time\ difference\ between\ successive\ enoder\ counts$

$N_r = Counts\ per\ Main\ Shaft\ Rotation$

$= 3\ pulses\ per\ Rotation\ per\ Gear\ Ratio * 60\ Gear\ Ratio * 4\ Counts\ per\ pulse$

$= 720$

1.5. Push Button

Since both the interrupt pins were used by the Encoders, the push button and de-bouncing was implemented using Polling. An LED was attached in series to the switch to glow whenever a switch press was detected. The push button code designed by me allowed to switch between the different states of operation. Here, the states are:

a. Servo Motor Control
b. Stepper Motor Control
c. DC Motor Velocity Control
d. DC Motor Position Control

1.6. Electronic Integration

I had informed the Arduino pin connection requirements for the 3 sensors, servo motor, and the motor drivers, as well as not using Pins 0 and 1 to avoid Serial prints to affect our system. Hence, final integration was only a matter of repositioning the individual sensor-motor pairs to reduce wire crossings, shortening wire lengths, and being user-friendly. The driver and steppers were setup closer to each other, and effort was made to keep their respective sensors close on the final board.

2. Challenges

2.1. DC Motor PID Control

The most challenging part was the PID calibration for the Velocity and Position Controls.

As the PID control required tuning of the proportional, integral, and differential constants to ensure proper tracking of the set parameters with acceptable rise time, reduced oscillations and feasible settling time, it was quite time-consuming.

However, the appropriate method toward tuning a PID controller is to:

1. Set a proportional gain (K_p) to reach close to the set quantity in short time, but with least overshoot and oscillations.
2. Set an integral gain (K_i) to reach the set quantity faster instead of being just close to it. However, to big an integral gain will amplify accumulated errors too much and make the system unstable.
3. Set the differential gain (K_d) to predict future gains and stabilize the system. However, this increases the settling time.

Though this was straightforward for Velocity control, for the Position Control, the time difference between successive measurements was quite high (180ms). Coupled with the motor requiring a minimum PWM of 40% to start from rest (~30RPM), resulted in the motor overshooting the required position before being stopped.

It was later realized that this huge time difference was resulting because of Step 2a. in the PID process described in 1.3. DC Motors. The repeating calculation resulted in the constant delays.

Instead, the SetPoint could be changed into required number of Encoder Counts for the Position Control, and the PID function could be directly called, removing the delays from "float" type calculations, resulting in a favorably working Position Control.

The following are good references to learn PID Control implementation: [4], [5]

2.2. Collaboration

When a team member needed to test and add his/her components to the full system in each other's absence, it resulted in incorrect connections for other modules of the circuit. Though this only happened twice, one of the times was right before the demonstration to the TA. Though the issue was a silly one of the direction input wires to the motor driver being exchanged, this resulted in the DC Motor not working in the allotted time slot, affecting the team's performance evaluation.

3. Teamwork

Logan Wan was responsible for designing the ultrasonic sensor mount, and programming the servo motor to interact with the ultrasonic sensor.

Clare Cui was responsible for bringing up the stepper motor driver, setting relevant current-limit, and controlling the Stepper Motor with obstacle distance inputs from the Sharp IR Sensor. The transfer function of the IR Sensor is shown in Figure 3. Clare shared the electronic hardware integration part with Maitreya Naik. She was responsible for placing the Stepper Motor driver, the Motor, the IR Sensor, and fixing the DC Motor in place.

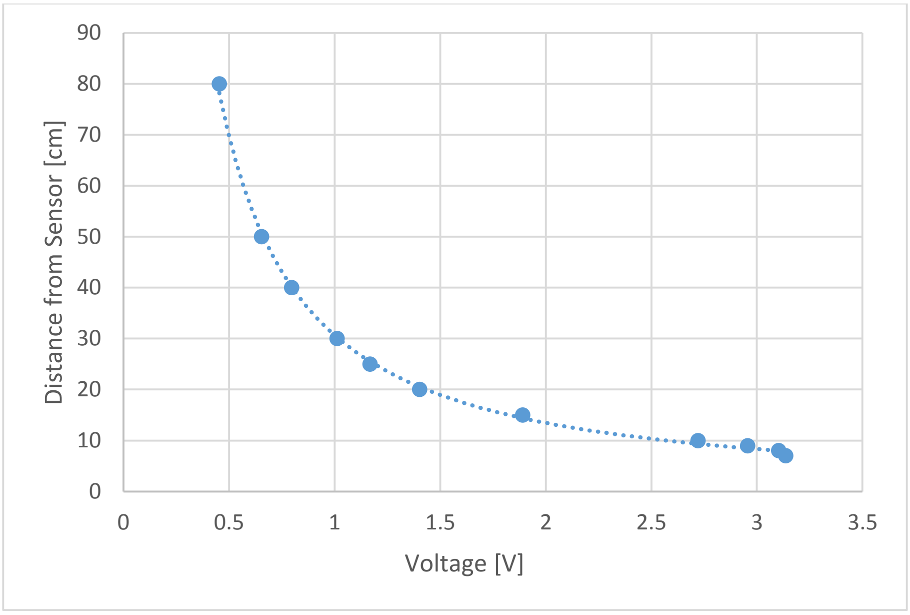

Figure 3: Sharp IR distance vs voltage transfer function *[by Clare Cui].*

Angad Sidhu was responsible for designing the GUI, integrating the independent codes, plotting relevant graphs, and enabling state-switching – providing a software complement to the push button. He also enabled Servo Motor control through a manual slider on the GUI.

4. Plan

We are done with the initial assembly of the Hexrotor. The Remote control radio was already set up by the team. I enabled the Remote Controlled flight of the Hexrotor by physically integrating the flight controller (Pixhawk [6]) with the 6 ESCs, a safety switch, a buzzer and a Telemetry Radio to enable wireless logging of Aerial vehicle data. I also calibrated the Flight Controller's Gyroscope, Accelerometer, and RC signals to enable the Remote Controlled flight. The document at [7] was followed for calibration and bringup.

We also collected the LiDAR, IMU, and GPS data of our test site according to our CoDR plan and are working with the sensor calibration to be completed by the next Progress Review.

Going ahead, I will be integrating the Pixhawk Flight Controller with our on-board computer, before finishing the sensor calibration with Angad. Me and Angad will then work on the State estimation problem of our system. Logan and Clare will be working on the sensor mount development before continuing to start the Map Generation part of the system.

References

[1] "DC Geared Motor with Encoder SPG30E-60K," Cytron Technologies, [Online]. Available: http://www.cytron.com.my/p-spg30e-60k.

[2] "Solarbotics L298 Compact Motor Driver Kit," Solarbotics, [Online]. Available: https://solarbotics.com/product/k_cmd/.

[3] "Encoder Library for Paul Stoffregen," [Online]. Available: https://github.com/PaulStoffregen/Encoder.

[4] "Arduino Discussion Forum," [Online]. Available: http://forum.arduino.cc/index.php?topic=8652.0.

[5] "Phidgets DC Motor - PID Control," [Online]. Available: http://www.phidgets.com/docs/DC_Motor_-_PID_Control.

[6] "Pixhawk Flight Controller," Px4 Autopilot, [Online]. Available: https://pixhawk.org/.

[7] "Pixhawk Quick Start Guide," PX4, [Online]. Available: https://3dr.com/wp-content/uploads/2014/03/pixhawk-manual-rev7.pdf.

Appendix

A.1. DC Motor Velocity and Position Control Code

```
// ___  _      _ _    _   _   _____  _______
///  __|   \ /   |   |  | | |_| |  _    | |__  __|
//  (__|  |\_/|  |   |_| |    _  | |_| /  __| |__
//\___|_|     |_|\___/  |_| |_| \_\|______|
//
// MRSD Task 7 Fall 2016
//
// Made by Maitreya Naik
// License: CC-BY-SA 3.0
// Website: maitreyanaik.wordpress.com

#define ENCODER_USE_INTERRUPTS true
#define LOOPTIME         1000
#define M_pot            A0
#define M_DC_Speed       5
// define pins for motor direction input to motor driver
#define M_DC_IN1              12
#define M_DC_IN2              13
#include <Encoder.h>

int M_DC_PWM_val = 0;
int M_potval = 0;
int M_potval_last = 0;
int M_potval_diff = 0;

Encoder M_DC_my_Enc(3, 2);

boolean M_motor_move = false;

int M_DC_Speed_Req = 0;
int M_DC_Ang_Req = 0;
int M_DC_Ang_Req_old = 0;

int M_DC_Speed_Act = 0;
int M_DC_Ang_Act = 0;

unsigned long M_DC_lastMilli = 0;
unsigned long M_encoderPos_UpdateTime = 0;
unsigned long M_encoderPos_LastTime = 0;

long M_encoderPos = 0;
long M_encoderPos_last = 0;

float M_DC_count_per_rot = 3*60.0*4;

// PID Speed Params Old
float M_DC_Kp_S =   0.6;                              // PID Proportional Gain for Speed
float M_DC_Ki_S =    0.02;                             // PID Integral Gain for Speed
float M_DC_Kd_S =    0.1;                              // PID Derivitave Gain for Speed

// PID Position Params
float M_DC_Kp_A =   1;                              // PID Proportional Gain for Speed
float M_DC_Ki_A =    0.01;                            // PID Integral Gain for Speed
float M_DC_Kd_A =    1.2;                             // PID Derivitave Gain for Speed
//Allowable tolerance in degrees
float M_DC_epsilon_ang = 5;

void setup() {

  pinMode(M_DC_Speed, OUTPUT);
  pinMode(M_DC_IN1, OUTPUT);
  pinMode(M_DC_IN1, OUTPUT);
  Serial.begin(9600);
  Serial.println("start");
  M_potval_last = readPot();
}

void loop() {
  //Serial.print("Encoder Count: "); Serial.println(M_DC_my_Enc.read());
  M_potval = readPot();
  ////////////////////////////////////////Position PID////////////////////////
  if(M_potval < 0 || M_potval > 1023)
  {
    M_potval = M_potval_last;
    M_motorBrake();
    M_DC_Speed_Req = 0;
    M_DC_Ang_Req = 0;
  }
  M_potval_diff = M_potval - M_potval_last;
  M_DC_Ang_Req = map(M_potval_diff, -1023, 1023, -179, 180);

  // If required degree rotation is in deadband, make required degree rotation 0
  if(-1*M_DC_epsilon_ang < M_DC_Ang_Req && M_DC_Ang_Req < M_DC_epsilon_ang)
    M_DC_Ang_Req = 0;
  // If new angle set point is greater than old angle set point, reset encoder counts
```

```
 else if(abs(M_DC_Ang_Req - M_DC_Ang_Req_old) > M_DC_epsilon_ang+2)
 {
  M_DC_my_Enc.write(0);
  M_DC_Ang_Req_old = M_DC_Ang_Req;
 }

 /////////////////////////////////////////////Simple Mapping (without PID)
 /*
 M_DC_Ang_Act = 360*M_encoderPos/M_DC_count_per_rot;
 float M_DC_Ang_move = map(M_DC_Ang_Act - M_DC_Ang_Req, -360, 360, -255, 255);
 if(M_DC_Ang_move < -1*M_DC_epsilon_ang)
  M_motorBackward(-1*M_DC_Ang_move);
 else if (M_DC_Ang_move > M_DC_epsilon_ang)
  M_motorForward(M_DC_Ang_move);
 else
  M_motorBrake();
 */
 /////////////////////////////////////////////

 //////////////////////////////// Implement PID
 if((millis()-M_DC_lastMilli) >= 0)   {                              // enter tmed loop
  M_DC_lastMilli = millis();
  M_encoderPos = M_DC_my_Enc.read();

  M_DC_Ang_Act = round(360*M_encoderPos/M_DC_count_per_rot);
  M_encoderPos_UpdateTime = millis();
  M_DC_Ang_Act %= 360;
  if(M_DC_Ang_Act > 180)
   M_DC_Ang_Act -= 360;
  else if(M_DC_Ang_Act < -180)
   M_DC_Ang_Act += 360;
  M_DC_PWM_val= M_DC_Ang_updatePid(M_DC_PWM_val, M_DC_Ang_Req,
M_DC_Ang_Act, M_encoderPos_UpdateTime, M_encoderPos_LastTime);                    //
compute PWM value

  Serial.print("Required: "); Serial.println(M_DC_Ang_Req);
  Serial.print("Actual Angle: "); Serial.println(M_DC_Ang_Act);
  if(abs(M_DC_Ang_Req - M_DC_Ang_Act) < M_DC_epsilon_ang){
   M_potval_last = M_potval;
   M_DC_my_Enc.write(0);
   M_encoderPos_LastTime = millis();
   M_DC_Ang_Req_old = M_DC_Ang_Req;
  }
  M_encoderPos_LastTime = M_encoderPos_UpdateTime;
```

```
    // If required PWM is in deadband, make PWM 0
    if(M_DC_PWM_val > 0)
      M_motorForward(M_DC_PWM_val);
    else if (0 < -1*M_DC_PWM_val)
      M_motorBackward(-1*M_DC_PWM_val);
    else
      M_motorBrake();
  }
  ////////////////////////////////// PID End
  //////////////////////////////////////////END POSITION/////////////////////////

  /////////////////////VELOCITY PID////////////////////////////////////
//  /////////////
//  ////
//  ////
//  if(M_potval < 0 || M_potval > 1023)
//  {
//    M_motorBrake();
//    M_DC_Speed_Req = 0;
//    M_DC_Ang_Req = 0;
//  }
//  M_DC_Speed_Req = map(M_potval, 0, 1023, -1*76, 76);
//  /////////////////////////////////////////////Simple Mapping (without PID)
//  /*
//  if(M_potval < 512-2*M_DC_epsilon_pwm)
//    M_motorBackward(M_potval/2);
//  else if (M_potval >= 512+2*M_DC_epsilon_pwm)
//    M_motorForward((M_potval - 512)/2);
//  else
//    M_motorBrake();
//  */
//  /////////////////////////////////////////////
//
//  //////////////////////////////// Implement PID
//  if((millis()-M_DC_lastMilli) >= LOOPTIME)   {                              // enter tmed loop
//    M_DC_lastMilli = millis();
//    M_encoderPos = M_DC_my_Enc.read();
//    M_encoderPos_UpdateTime = millis();
//    M_DC_Speed_Act = ((M_encoderPos -
M_encoderPos_last)*(60*(1000.0/(M_encoderPos_UpdateTime -
M_encoderPos_LastTime))))/(M_DC_count_per_rot);
//    Serial.print("Required: "); Serial.println(M_DC_Speed_Req);
//    Serial.print("Actual Velocity: "); Serial.println(M_DC_Speed_Act);
//    Serial.print("Encode Diff: "); Serial.println(M_encoderPos - M_encoderPos_last);
//
//    M_DC_PWM_val= M_DC_Speed_updatePid(M_DC_PWM_val, M_DC_Speed_Req,
M_DC_Speed_Act, M_encoderPos_UpdateTime, M_encoderPos_LastTime);                    //
compute PWM value
//
//    M_encoderPos_last = M_encoderPos;
//    M_encoderPos_LastTime = M_encoderPos_UpdateTime;
//
//    //Serial.print("PWM Ordered: "); Serial.println(M_DC_PWM_val);
//    // If required PWM is in deadband, make PWM 0
//    if(M_DC_PWM_val > 0)
//      M_motorForward(M_DC_PWM_val);
//    else if (M_DC_PWM_val < 0)
//      M_motorBackward(-1*M_DC_PWM_val);
//    else
//      M_motorBrake();
//  }
//  ////////////////////////////////// PID End
// put your main code here, to run repeatedly:

}

/*******************************Potentiometer
Reading*******************************/
int readPot(){
  int M_array_potval, M_potval_avg;
  M_potval_avg = 0;
  for(int i=0; i<8; i++){
    M_array_potval = analogRead(M_pot);
    M_potval_avg += M_array_potval;
  }
  M_potval_avg /= 8;
  return M_potval_avg;
}
/*******************************Motor Movement
Fix*******************************/
void M_motorForward(int PWM_val)  {
  Serial.print("Forward by "); Serial.println(PWM_val);
  analogWrite(M_DC_Speed, PWM_val);
  digitalWrite(M_DC_IN1, HIGH);
  digitalWrite(M_DC_IN2, LOW);
  M_motor_move = true;
}
```

```
void M_motorBackward(int PWM_val)  {
 Serial.print("Back by "); Serial.println(PWM_val);
 analogWrite(M_DC_Speed, PWM_val);
 digitalWrite(M_DC_IN1, LOW);
 digitalWrite(M_DC_IN2, HIGH);
 M_motor_move = true;
}

void M_motorBrake()  {
 analogWrite(M_DC_Speed, 0);
 digitalWrite(M_DC_IN1, HIGH);
 digitalWrite(M_DC_IN2, HIGH);
 M_motor_move = false;
}
/******************************PID Speed*******************************/
int M_DC_Speed_updatePid(int command, int targetValue, int currentValue, unsigned long t1,
unsigned long t2) {            // compute PWM value
  float pidTerm = 0;                                              // PID correction
  int error=0;
  double dt = (t1 - t2)/1000.0; //time change in secs
  static int last_error=0;
  static long sum_of_errors = 0;
  error = targetValue - currentValue;
  Serial.print("Last PID Error: "); Serial.println(last_error);
  Serial.print("PID Error: "); Serial.println(error);
  sum_of_errors += error;
  Serial.print("PID Sum of Errors: "); Serial.println(sum_of_errors);
  pidTerm = (M_DC_Kp_S*error) + (M_DC_Ki_S*(sum_of_errors)*dt) + (M_DC_Kd_S*(error -
last_error)/dt);
  last_error = error;
  Serial.print("*******PID Term: "); Serial.println(pidTerm);
  Serial.print("PWM: "); Serial.println(command + round(pidTerm));
  return constrain(command + round(pidTerm), -255, 255);
}

/******************************PID Angle*******************************/
int M_DC_Ang_updatePid(int command, int targetValue, int currentValue, unsigned long t1,
unsigned long t2) {            // compute PWM value
  double pidTerm = 0;                                              // PID correction
  double error=0;
  double dt = (t1 - t2)/1000.0; //time change in secs
  if (dt == 0)
    dt = 1;
  static int last_error=0;
  static long sum_of_errors = 0;
  error = (targetValue - currentValue);
  if(error > 180)
    error = int(error-360)%180;
  else if (error <= -180)
    error = int(error+360)%180;
  //Acceptable error
  if(abs(error) <= M_DC_epsilon_ang){
    sum_of_errors = 0;
    last_error = 0;
    error = 0;
    Serial.println("REEEEEEEEEEEEEEEEEEEEEEEEEEEEAAAAAAAAAAAAACCCCCCCCHHHH");
    M_motorBrake();
    return 0;
  }
  Serial.print("PID Error: "); Serial.print(error); Serial.println("Degrees");
  error = error*22.0/(7*180.0);
  Serial.print("Time Diff: "); Serial.println(dt);
  Serial.print("Last PID Error: "); Serial.println(last_error);
  Serial.print("PID Error: "); Serial.print(error); Serial.println("Rads");
  sum_of_errors += error;
  Serial.print("PID Sum of Errors: "); Serial.println(sum_of_errors);
  pidTerm = (M_DC_Kp_A*error) + (M_DC_Ki_A*(sum_of_errors)*dt) + (M_DC_Kd_A*(error -
last_error)/dt);
  last_error = error;
  Serial.print("*******PID Term: "); Serial.println(pidTerm);
  Serial.print("PWM: *********** "); Serial.println(command + round(pidTerm));
  return constrain(command + round(pidTerm), -255, 255);
}
```

A.2. Debounce Function

```
void M_push_button_debounce(){
 // Check if push button was pressed. If so, move motor to State 0 (STOP)
 int M_reading = digitalRead(M_push_button);
 if(M_reading != M_push_button_last_button_state)
 {
  if ((millis() - M_push_button_last_press) > M_push_button_debounce_delay) {

   // if the button state has changed:
   if (M_reading != M_push_button_state) {
    M_push_button_state = M_reading;

    // Toggle DC Motor state to 0 (STOP) if pushbutton is pressed
    if (M_push_button_state == LOW) {
     ////////////////////////////////// Finally uncomment
     /*
     if(M_DC_State)
      M_DC_State = 0;
     else if(!M_DC_State)
      M_DC_State = 1;
     */
     ///////////////////////////////////// Finally Comment the following line
     M_DC_State = (M_DC_State + 1)%3;  //Cycle between states for debugging
    }
   }
  }
  M_push_button_last_press = millis();
  M_push_button_last_button_state = M_reading;
 }
}
```