Concordia University
Department of Mechanical, Industrial and Aerospace Engineering (MIAE)

Title:

Final Project Report for:

MECH 471/6621
Microprocessors and its applications

Instructor:

Prof. Brandon W. Gordon

Prepared by:

Amirhossein Dehestani (40225981)

Jalal Mahdieh (40096156)

Sourena morteza ghasemi (40171622)

Amirreza Azadnia (40198570)

Winter 2023

Amirhossein Dehestani

Student ID: 40225981

Contribution: Main body code, PWM, Analog to digital, cruise control, HIl implementation.

Amirreza Azadnia

Student ID: 40198570

Contribution: Timer interrupt, Servo control, traction control, HIl implementation.

Jalal Mahdieh

Student ID: 40096156

Contribution: Modeling and simulation of variable equations on MATLAB, Simulink, brake control.

Sourena Morteza Ghasemi

Student ID: 40171622

Contribution: Tuning the PID coefficients, Analog to digital, traction control.

Table of contents

Abstract

The goal of this project is to control the movement of a robot in a simulation space. The simulation and control are being uploaded on two different Arduino which have serial communication to each other using. The controller is divided in three different part of traction control, cruise control and brake control of the robot. At each part, the signal of speed of the motor is beeing reed frequenty using function name ADC_reed. Then after applying the PID control on this feedback signal, a servo pulse is being created using servo_setup function to control the speed of the servo motor.

The ADC_Read function:

ADC converts the voltage to a digital integer that is readable for the computer. The ADC_read program can be used to determine the speed of robot's wheels by converting Pulse-Width Modulation (PWM) signals to voltage values. PWM signals are commonly used to control the speed of DC motors and servos and are generated by a microcontroller or other digital device.

In this project, the ADC_read function is called three times, each with a different analog pin (A1, A3, A5) to measure the PWM signal from three different motors. The function measures the PWM signal 100 times using the ADC and takes the average value of the readings to get a more accurate representation of the signal.

At first, the channels are set, and the prescaler of 128 is considered for this project. The function then reads the ADC “n” times, by starting a conversion, waiting for the conversion to complete, and adding the converted value to the "sum" variable. After all “n” readings have been completed, the ADC is disabled.

The function then converts the average PWM reading to a voltage value using the formula sum * 5.0 / (n * 1023.0), where sum is the sum of the ADC readings, n is the number of readings, and 1023.0 is the maximum value of the ADC range. The resulting voltage values are stored in three separate variables, y1, y2, and y3, which represent the inputs to the PID controller for each of the three motors.

The PID controller uses these voltage values to control the speed of the motors by adjusting the PWM signal. By measuring the voltage output of the ADC_read function, the PID controller can determine the actual speed of each motor and adjust the PWM signal accordingly to maintain the desired speed. The code of this function is shown below:

```
 float ADC_reed(int channel, int n) {
  float sum = 0;

  ADMUX=0;
  // set ADC reference voltage to AVCC
  ADMUX |= BIT (REFS0);
  ADMUX &= ~BIT(REFS1);

  // set ADC input channel
  ADMUX &= 0xF0; // clear channel selection bits
  ADMUX |= (channel & 0x0F); // set channel selection bits

  ADCSRA = 0;
  // enable ADC
  ADCSRA |= BIT(ADEN);
  //set prescaler to 128
  ADCSRA |= BIT(ADPS2) | BIT(ADPS1) | BIT(ADPS0);

  // read ADC n times and accumulate the sum
  for (int i = 0; i < n; i++) {
    ADCSRA |= BIT(ADSC); // start conversion
    while (ADCSRA & BIT(ADSC)); // wait for conversion to complete
    sum += ADC; // add converted value to sum
  }

  // disable ADC
  ADCSRA &= ~BIT(ADEN);

  float voltage = sum * 5.0 / (n * 1023.0);
  // calculate average and return
  return voltage;
}
```

Servo_write function:

This function is used for generating a PWM signal with a specific frequency and duty cycle using the “Timer1” module in an Arduino. The generated PWM signal is used to control a servo motor. The serial communication with a baud rate of 115200 bauds per second is set for this project. The program sets the prescaler to 8, which means that the timer clock frequency is divided by 8. The timer constant (T_const) is calculated based on the prescaler and the CPU frequency (16 MHz). The PWM period is set to 0.5 milliseconds (2000 Hz frequency). The program also defines a function to set up the servo motor and another function to write a specific duty cycle to the servo motor.

The servo_setup function configures Timer1 for PWM generation. It would set the pin D5 as output and sets the Timer1 prescaler to 8. It enables Timer1 overflow interrupt and sets the initial value of Timer1 to T1_start, which is calculated based on the desired PWM frequency and duty cycle. The function enters first while loop that runs the Traction_mode() function until the time elapsed since t0 is less than 15 seconds. In this stage the robot should be begun to move. The function then enters next while loop that runs the Cruise_mode() function until the time elapsed since t0 is between 15 and 45 seconds and is supposed to keep the robot in a certain speed. And finally, the function enters a final while loop that runs the Brake_mode() function until the time elapsed since t0 is greater than 45 seconds and the robot stops. The function waits for one second A delay of 1000 ms is set after breaking the loop. These are the main commands that are sent to PID controller and control the robot. The servo_write function calculates the timer value (T_servo) at which the servo pin needs to be turned on and sets the output compare register (OCR1A) to this value. It also takes two arguments, pw1 and pw2, which are pulse widths in microseconds for two servo motors. It also enables the compare register interrupt to turn off the servo pin when the timer reaches the specified value.

The “ISR TIMER1_COMPA_vect” is executed two times when the timer value matches the output compare register value, which turns on the servo pin of 6 and 7 respectively. The “ISR TIMER1_OVF_vect” is executed when the timer overflows, which turns off the servo pins and restarts the timer with the initial value. The related code is as follow:

```
void servo_setup(){
cli();
    DDRD |= BIT(6); // set pin 6 to output
    DDRD |= BIT(7); // set pin 7 to output

// Reset Timer1 Controls
    TCCR1A = 0;
    TCCR1B = 0;
 // set timer1 prescaler to 8
    TCCR1B |= BIT(CS11);
//enabling the TC1 overflow
    TIFR1 |= BIT(TOV1);
    TIMSK1 = 0 ;
    TIMSK1 |= BIT(TOIE1) | BIT(OCIE1A) | BIT(OCIE1B);;
    // set timer1 initial value
    TCNT1 = T1_start;
}
void servo_write( float pw1 , float pw2 )
  {
    cli();
  // first we need to specify when we have to turn the servo pin
    long int T_servo1 = 65535 - ( T_const_inv * pw1 * 1.0e-6 );
    long int T_servo2 = 65535 - ( T_const_inv * pw2 * 1.0e-6 );

  //we need to enter the compare match interuput function
  //when the timer reach to the T1_servo1
    OCR1A = T_servo1 ;
    OCR1B = T_servo2 ;

  // enabling the compare register interrupt
    TIFR1 |= BIT(OCF1A);
    TIFR1 |= BIT(OCF1B);
    sei();
  }

ISR(TIMER1_COMPA_vect){
  // turn pin 6 on
    PORTD |= BIT(6);
    //Serial.println("a");
  }

ISR(TIMER1_COMPB_vect){
  // turn pin 7 on
    PORTD |= BIT(7);
     // Serial.println("b");
  }

ISR(TIMER1_OVF_vect){
  // turn pin 6 and 7 off
  //Serial.println("c");
  PORTD &= ~BIT(6);
  PORTD &= ~BIT(7);
  // restart the timer
  TCNT1 = T1_start;
  }
```

Tuning the PID controller

The gain of PID controller is obtained based on trial and error however the following strategy is used to modify integral gain. This makes it possible to have high value of integral without having any hunting and integral windup. This is important because a high integral value can result in a small tracking error.

Proportional gain Kp

Kp is used as an amplifier to reach the force of the actuators to the adequate amount of the robustness of their weight and the moving platform. The value of Kp will be increased till the system becomes unstable.

Integral gain

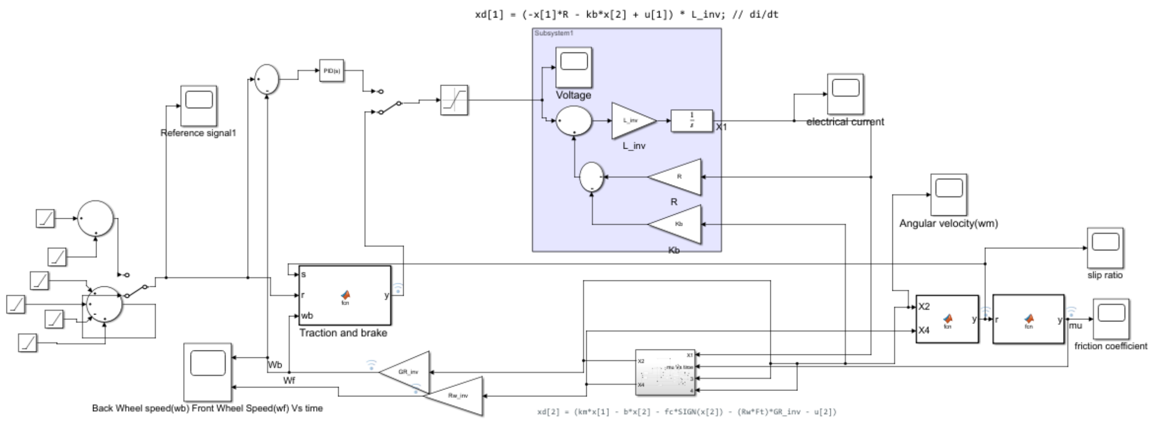
As it was discussed the main problem is reducing tracking error. Therefore, a larger value of the integrator is required. The integral windup appears when the integral gain is higher than usual. To deal with this problem one method is used allowing us to increase the integral gain. In this approach, the recommendation is to apply the following formula to modify the integral gain:

$$I = \left(k_i \int e(t)\,dt\right)\frac{\alpha}{\alpha + \dot{e}^2} \qquad (1)$$

The most imperative advantage of this method is that the integrator will shut down when integral windup occurs in the system. So, at the high velocity, the system does not have problem tracking. Hunting is another problem in the tracking. To eliminate the hunting problem, and to improve the path tracking, α is obtained by the following formula.

$$\alpha = \alpha_{max}\left(\gamma + \frac{|\dot{\theta}_d|}{\dot{\theta}_{max}}\right) \tag{2}$$

where γ is a small constant that defines the ratio between α_{max} (at maximum commanded joint velocity) and minimum α (at steady-state). Linking the nonlinear factor α to the set-point velocity, $\dot{\theta}_{max}$, as described by Eq. (8). It leads to a relatively weaker integral when the set-point is barely changing, as in the steady state, compared to the case of tracking a steep ramp. Therefore, this modification reduces the hunting effect in the steady state. First the system was modeled in SIMULINK to tune the gains.

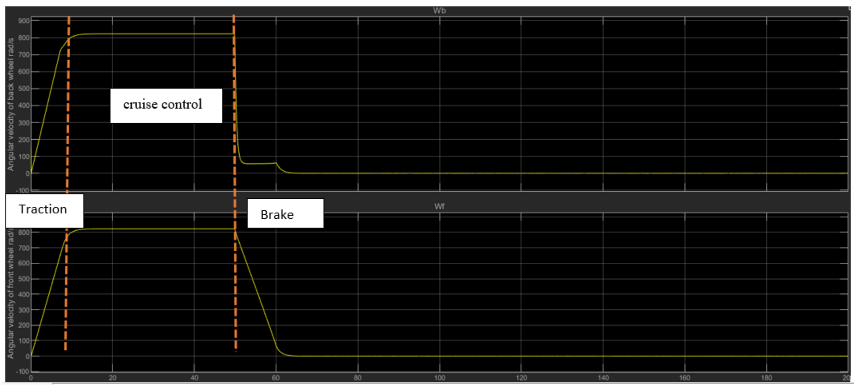


```
xd[1] = (-x[1]*R - kb*x[2] + u[1]) * L_inv; // di/dt
```

```
xd[2] = (km*x[1] - b*x[2] - fc*SIGN(x[2]) - (Rw*Ft)*GR_inv - u[2])
```

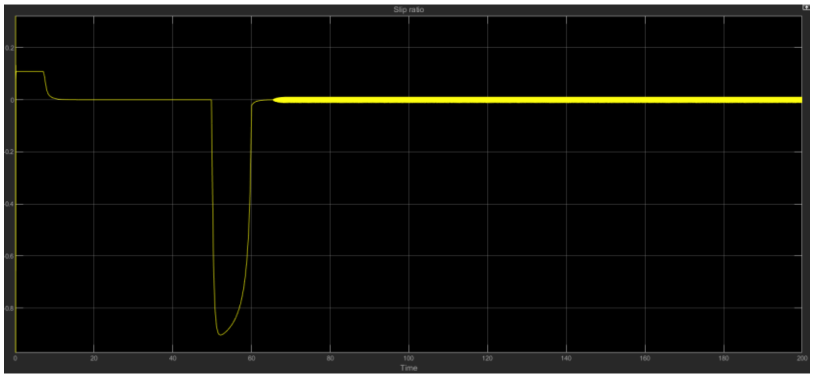
System modeled in Matlab

The following figure shows the speed of back and front wheel during the simulation. It is clear in traction and brake it almost takes some seconds for front wheels to reach velocity of back wheel.



Tracking of back and front wheels

Based these scenarios the obtained slip ratio is as follows:



The graph shows the rate of slip ratio remain at constant value during traction and brake which leads to maximum performance.

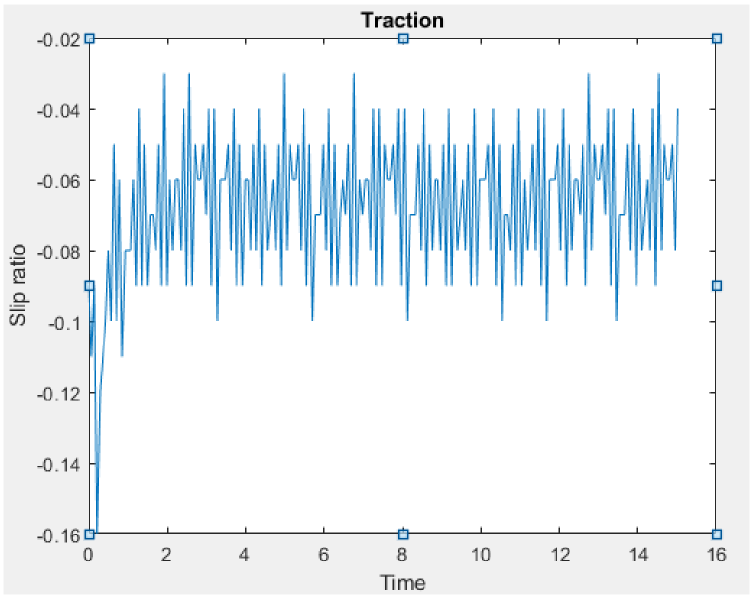
Traction

```
void traction_mode()
{
  float y1 = -1, y2 = -1, y3 = -1;
  float wb, wf, s;
  float u1, u2;
  int pw1, pw2;
  int w1, w2;
    int Alph;
  float t,t1;
  const int   PW_MAX = 2500, PW_0 = 1500, PW_MIN = 1000;
  const float wmax = 810.0; // maximum back wheel speed (rad/s)
  const float V_bat = 12.0; // lipo battery voltage
  const float V_bat_inv = 1/V_bat;
  ////////////////////
  int n;
  float s_dot;
  float dt;
  float r, v, u;
  float kp, ki, kd;
  float e, ed, z, ei_max;
  // use static floats for local variables that need to be
  // remembered between function calls.
  // static variables are like global variables that can only
  // be used by the function.
  // regular local variables get erased at the end of the function.
  static float tp = 0.0; // previous time (initial value = 0)
  static float ei = 0.0; // integral state (initial value = 0)
  static float ep = 0.0; // previous error

  // PID controller gains
  kp = 20.0;
  ki =45.345;
  kd = 0.0;

  // the control loop typically performs the following steps:
  // 1) read time
  // 2) read sensors
  // 3) calculate controller input
  // 4) write input to the actuators

  // 5) save data for plotting, etc.
  // -- be careful since this can slow the controller down
  // for the final controller don't do this
  // or maybe save it to SD card memory, etc.

  y1 = ADC_reed(1 , number_of_voltage_read);
  y2 = ADC_reed(3 , number_of_voltage_read);
  y3 = ADC_reed(5 , number_of_voltage_read);

  wb = (y1 ) * 0.4 * wmax;
  wf = (y2 ) * 0.4 * wmax;
 //   s =  (y2 - y1 ) / abs(y2);
  s = (wf - wb)/abs(wf);

  t = micros()*1.0e-6 - t0; // s


  if( t < 15 ) { // step input at t = 5 s
    r = 0.18; // A = 0.2
  } else{
    r = 0;
  }

  //r=wmax;
  // calculate dt
  dt = t - tp; // measure sampling period dt
  tp = t; // save previous sample time

  // calculate controller error
  e  = r - s;
//  ed = 0 - v; // ed = rd - yd
 s_dot= s/dt;
```

```
// finite difference approximation for ed (FYI)
ed = (e - ep) / dt;
  ep = e; // save current error for next time

ei += e*dt; // I += e*dt


if( ki > 0.0 ) {
  ei_max = 0.14*V_bat/ki; // want ki*ei < 0.14*V_bat
} else {
  ei_max = 0.0;
}

// stop integration if integral is too large and increasing
// ei += z*dt
// where z is set appropriately if integral is too large

// set z
if ( (ei > ei_max) && (e > 0) ) {
  // positive out of bounds, positive integration
  z = 0; // stop integration
} else if ( (ei < -ei_max) && (e < 0) ) {
  // negative out of bounds, negative integration
  z = 0; // stop integration
} else { // either in bounds or integration reduces integral
  z = e; // normal integration
}
  // NOTE: if you use this you must comment out ei += e*dt
ei += z*dt;
Alph = 0.006*(0.0001+s_dot/0.3);
// PID controller
u1 = kp*e + ki*ei*0.006*((Alph)/(Alph+ed*ed)) + kd*ed;
  if( u1 > V_bat )  u1 = V_bat;
if( u1 < -V_bat ) u1 = -V_bat;
 u2 = 0.0;
// convert inputs to actuator values pw1, pw2
// convert motor voltage to pulse width
// u1 = (w1 - PW_0) * PW_R * V_bat ->
w1 = u1 * V_bat_inv * (PW_MAX - PW_0) + PW_0;
  // saturate input if out of range
if(w1 > PW_MAX) w1 = PW_MAX;
if(w1 < PW_MIN) w1 = PW_MIN;

pw1 = w1;
```

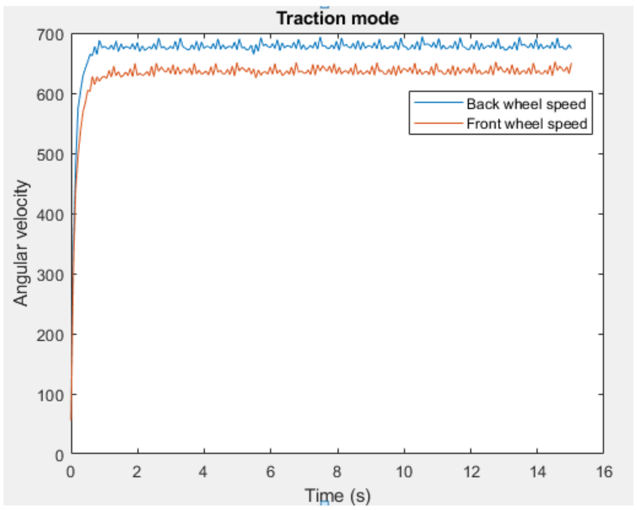
```
// set pw2 for testing purposes
pw2 = 0;
```

The slip ratio of the robot during traction is depicted in the following figure. Initially high, it decreases to an acceptable range over time.



Slip ratio

The figure presented depicts the velocity of the front and back wheels. It is evident that the front wheel reaches the same speed as the back wheel quickly during acceleration.



Traction

Cruise_mode function:

The first part of the code declares motor speed and battery voltage values, as well as PID controller gains. The control loop consists of several steps. First, the function reads sensor values from three analog input channels using an ADC_reed() function. These values are then used to calculate the motor speeds (wb and wf) based on the relationship between the input voltage and the maximum motor speed:

```
wb = (y1 - 2.5) * 0.4 * wmax;
wf = (y2 - 2.5) * 0.4 * wmax;
```

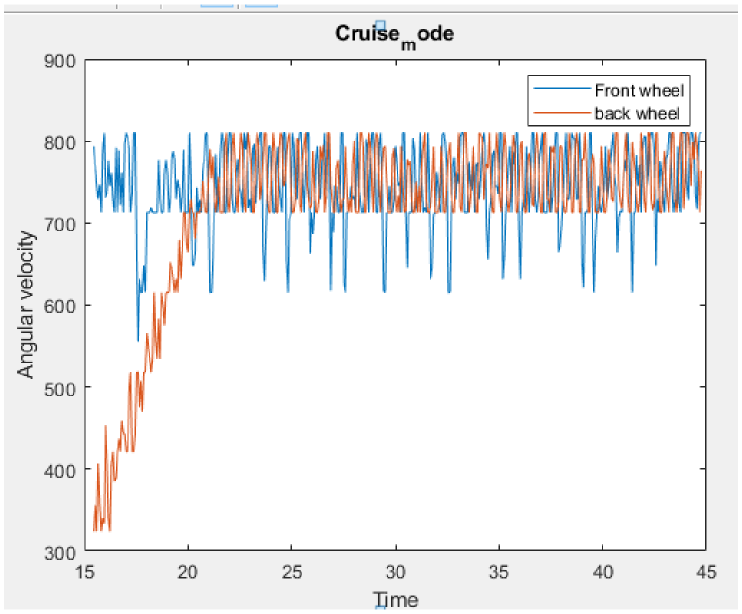
Then, the function calculates the error signal (e) between the desired reference input (r) and the measured back wheel speed (wb). The function then calculates the output of the PID controller (u1) we also put u2 equal to zero. the function converts the output of the PID controller (u1) to a pulse width value (w1) that is used to control the motor speed. The pulse width is then limited to a minimum and maximum value (PW_MIN and PW_MAX, respectively) and sent to a servo_write() function . The code below shows the abovementioned function:

```
y1 = ADC_reed(1 , number_of_voltage_read);
y2 = ADC_reed(3 , number_of_voltage_read);
y3 = ADC_reed(5 , number_of_voltage_read);

wb = (y1 - 2.5) * 0.4 * wmax;
wf = (y2 - 2.5) * 0.4 * wmax;

t = micros()*1.0e-6 - t0; // s

if( t > 15 ) { // step input at t = 5 s
    r = 810; // A = 0.5
} else {
    r = 0;
}
//r = wmax;
// calculate dt
dt = t - tp; // measure sampling period dt
tp = t; // save previous sample time
```

```
   // calculate controller error
   e = r - wb;
   //  ed = 0 - v; // ed = rd - yd

   // finite difference approximation for ed (FYI)
   ed = (e - ep) / dt;
   ep = e; // save current error for next time

   ei += e*dt; // I += e*dt

   if( ki > 0.0 ) {
       ei_max = 0.14*V_bat/ki; // want ki*ei < 0.14*V_bat
   } else {
       ei_max = 0.0;
   }

       // set z
   if ( (ei > ei_max) && (e > 0) ) {
       // positive out of bounds, positive integration
       z = 0; // stop integration
   } else if ( (ei < -ei_max) && (e < 0) ) {
       // negative out of bounds, negative integration
       z = 0; // stop integration
   } else { // either in bounds or integration reduces integral
       z = e; // normal integration
   }

   // NOTE: if you use this you must comment out ei += e*dt
   ei += z*dt;

   // PID controller
   Alph=1;

// PID controller
u1 = kp*e + ki*ei*0.006*((Alph)/(Alph+ed*ed)) + kd*ed;

   if( u1 > V_bat )  u1 = V_bat;
   if( u1 < -V_bat ) u1 = -V_bat;

   u2 = 0.0;

   // convert inputs to actuator values pw1, pw2

   // convert motor voltage to pulse width
   // u1 = (w1 - PW_0) * PW_R * V_bat ->
```

```
w1 = u1 * V_bat_inv * (PW_MAX - PW_0) + PW_0;

// saturate input if out of range
if(w1 > PW_MAX) w1 = PW_MAX;
if(w1 < PW_MIN) w1 = PW_MIN;

pw1 = w1;
```

During the simulation, the controller was able to maintain a constant velocity of the robot. The angular velocity of the back and front wheels was observed, and it took almost 20 seconds for the front wheel to reach the same velocity as the back wheel. However, the velocity of both wheels remained constant thereafter, indicating that the controller had good performance in maintaining a constant velocity for the robot.



Cruise mode