AuE 8350

Automotive Electronics Integration

Project 2 Report

Fall 2021

Submitted By:

Shubham Gupta
C38771108
gupta9@clemson.edu

Project 2: Adaptive Cruise Control and Autonomous Lane Keeping

Task 1: Adaptive Cruise Control

1.1 Problem Statement

The vehicle shall be able to exhibit Adaptive Cruise Control capabilities, i.e, it should be able to autonomously change its speed such that it stays 30 cm away from the obstacles ahead. Furthermore, if the obstacle ahead is stationary, then the vehicle should stop 30 cm away from the obstacle.

1.2 Technical Approach

The vehicle was controlled using a Proportional-Integral-Derivative (PID) controller. The PID controller provided us with an easy yet robust method of implementing adaptive cruise control, and we did not need to model the system.

The PID controller can be represented using the following equation:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{d}{dt} e(t)$$

where:

- u(t): PID control variable
- e(t): Error signal
- K_p: Proportional Gain
- K_i: Integral Gain
- K_d: Derivative Gain

However, since Arduino can only handle discrete data, we implement the PID controller on an Arduino as follows:

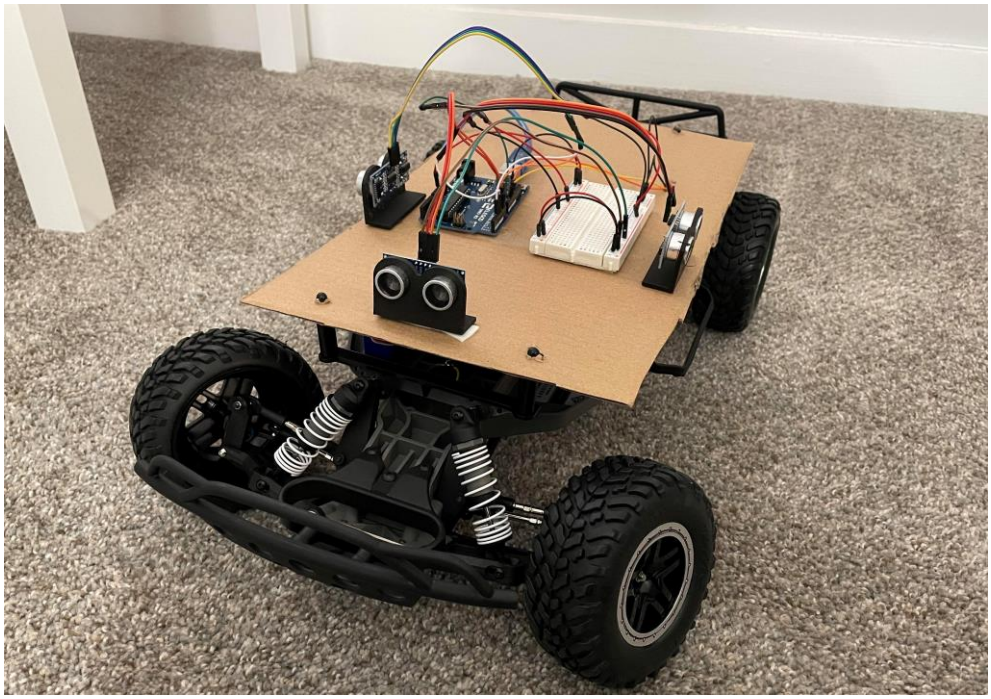
```
u(k) = u(k-1) + k1*e(k) + k2*e(k-1) + k3*e(k-2);
```

where:

- k1 = kp + ki + kd
- k2 = -kp -2*kd
- k3 = kd

1.3 Hardware and Software Implementation:

The following pictures demonstrate our setup for the project:

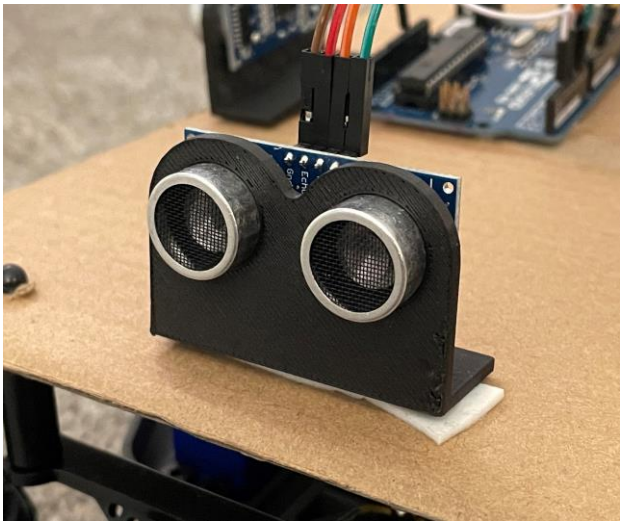


Our setup consisted of the following components

- RC Car
- Arduino Uno
- Front: One HC-SR04 sensor
- Sides: One HC-SR04 sensor each on left and right sides of the vehicle
- Breadboard
- Cardboard base to place the components
- Connecting wires

Additional Components (optional):

- 3D Printed Sensor Mounts



- Power Bank (to power Arduino)

PID Controller Implementation

The PID Controller is a non-model based controller and is implemented using the discrete data version on an Arduino. For Adaptive Cruise Control, the PID uses data from the ultrasonic sensor in the front to calculate error signal, and provide the required throttle response.

- Throttle = 90 for zero throttle
- Throttle > 90 for forward acceleration
- Throttle < 90 for backward acceleration

The PID controller parameters are as follows:

```
// Throttle Parameters
float e_thr = 0, e1_thr =0, e2_thr = 0;
float kp_thr = 0.38 , ki_thr = 0.0003, kd_thr = 0.00007;
float k1_thr = kp_thr + ki_thr + kd_thr;
float k2_thr = -(kp_thr + 2*kd_thr);
float k3_thr = kd_thr;
```

Handling the Dead Zone

The RC car motor has a noticeable **dead zone (between throttle = 86 and throttle = 94)**. Therefore, we need to handle this by starting the forward throttle values at 94 and backward throttle values at 86. Therefore, the PID controller is implemented as follows on the Arduino:

```
// Throttle PID Control
e_thr = 0;
e2_thr = e1_thr;
e1_thr = e_thr;
e_thr = -(30 - frontDistance);            // Compute current control error
if(frontDistance < 27){
  throttle = 86;
  throttle = throttle + (k1_thr*e_thr) + (k2_thr*e1_thr) + (k3_thr*e2_thr);
}
else if(frontDistance >= 27 && frontDistance <= 33){
  throttle = 90;
}
else{
  throttle = 94;
  throttle = throttle + (k1_thr*e_thr) + (k2_thr*e1_thr) + (k3_thr*e2_thr);
}
throttle = throttle + (k1_thr*e_thr) + (k2_thr*e1_thr) + (k3_thr*e2_thr);
throttle = max(min(maxThrottle, throttle), minThrottle);
```

Here, the throttle values have been restricted as follows for better control:

- Max Throttle = 100
- Min Throttle = 82

1.4 Experimental Results

The throttle control input values for front obstacle distance can be seen in the following snippet:

```
Front Distance = 30.90, e = 0.90 , e1 = 0.00, k1 = 0.38, k2 = -0.38, Throttle = 90.34
Front Distance = 30.89, e = 0.89 , e1 = 0.00, k1 = 0.38, k2 = -0.38, Throttle = 90.34
Front Distance = 31.37, e = 1.37 , e1 = 0.00, k1 = 0.38, k2 = -0.38, Throttle = 90.52
Front Distance = 34.44, e = 4.44 , e1 = 0.00, k1 = 0.38, k2 = -0.38, Throttle = 97.38
Front Distance = 37.36, e = 7.36 , e1 = 0.00, k1 = 0.38, k2 = -0.38, Throttle = 99.60
Front Distance = 37.74, e = 7.74 , e1 = 0.00, k1 = 0.38, k2 = -0.38, Throttle = 99.89
Front Distance = 39.92, e = 9.92 , e1 = 0.00, k1 = 0.38, k2 = -0.38, Throttle = 100.00
Front Distance = 36.07, e = 6.07 , e1 = 0.00, k1 = 0.38, k2 = -0.38, Throttle = 98.62
Front Distance = 28.65, e = -1.35 , e1 = 0.00, k1 = 0.38, k2 = -0.38, Throttle = 89.49
Front Distance = 23.33, e = -6.67 , e1 = 0.00, k1 = 0.38, k2 = -0.38, Throttle = 82.00
Front Distance = 21.10, e = -8.90 , e1 = 0.00, k1 = 0.38, k2 = -0.38, Throttle = 82.00
Front Distance = 20.34, e = -9.66 , e1 = 0.00, k1 = 0.38, k2 = -0.38, Throttle = 82.00
```

Task 2: Autonomous Lane Keeping

2.1 Problem Statement

The vehicle shall be able to exhibit Autonomous Lane Keeping capabilities, i.e, it should be able to autonomously follow along the center of a defined lane. Furthermore, it should be able to appropriately steer itself to maintain equal distance on either side.

2.2 Technical Approach

The Autonomous Lane Keeping assist was controlled using a Proportional-Integral-Derivative (PID) controller. The PID controller provided us with an easy yet robust method and we did not need to model the system.

The PID controller can be represented using the following equation:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{d}{dt} e(t)$$

where:

- u(t): PID control variable
- e(t): Error signal
- K_p: Proportional Gain
- K_i: Integral Gain
- K_d: Derivative Gain

However, since Arduino can only handle discrete data, we implement the PID controller on an Arduino as follows:

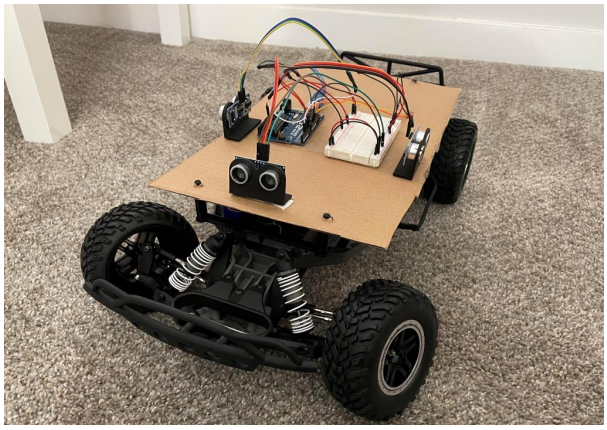
```
u(k) = u(k-1) + k1*e(k) + k2*e(k-1) + k3*e(k-2);
```

where:

- k1 = kp + ki + kd
- k2 = -kp -2*kd
- k3 = kd

2.3 Hardware and Software Implementation:

The hardware implementation is the same as that in Task 1 (Adaptive Cruise Control). The following pictures demonstrate our setup for the project:



Our setup consisted of the following components

- RC Car
- Arduino Uno
- Front: One HC-SR04 sensor
- Sides: One HC-SR04 sensor each on left and right sides of the vehicle
- Breadboard
- Cardboard base to place the components
- Connecting wires
- 3D Printed Sensor Mounts
- Power Bank (to power Arduino)

PID Controller Implementation

The PID Controller is a non-model based controller and is implemented using the discrete data version on an Arduino. For Autonomous Lane Keeping, the PID uses data from the ultrasonic sensors on the left and right to calculate error signal, and provide the required steering response.

- Throttle = 90 for neutral steering i.e no turning
- Throttle > 90 for turning right
- Throttle < 90 for turning left

The PID controller parameters for steering are as follows:

```
// Steering Parameters
float e_str = 0, e1_str =0, e2_str = 0;
float kp_str = 0.7 , ki_str = 0, kd_str = 0;
float k1_str = kp_str + ki_str + kd_str;
float k2_str = -(kp_str + 2*kd_str);
float k3_str = kd_str;
```

The PID controller for steering is implemented as follows on the Arduino:

```
// Steering PID Control
e2_str = e1_str;
e1_str = e_str;
e_str = -(leftDistance - rightDistance);            // Compute current control error
steering = steering + (k1_str*e_str) + (k2_str*e1_str) + (k3_str*e2_str);
if(abs(e_str) < 2){
  steering = 90;
}
steering = max(minSteering, min(maxSteering, steering));
```

Here, the steering values have been restricted as follows for better control:

- Max Steering = 125
- Min Steering = 55

2.4 Experimental Results

The steering control values for distances from the sides can be seen in the following snippet:

```
Left Distance = 38.03, Right Distance = 37.53, Steering = 90.00
Left Distance = 35.78, Right Distance = 37.25, Steering = 90.00
Left Distance = 35.04, Right Distance = 37.51, Steering = 90.70
Left Distance = 32.89, Right Distance = 36.86, Steering = 91.74
Left Distance = 31.76, Right Distance = 37.51, Steering = 92.99
Left Distance = 29.73, Right Distance = 36.86, Steering = 93.96
Left Distance = 28.68, Right Distance = 37.51, Steering = 95.15
Left Distance = 27.50, Right Distance = 37.51, Steering = 95.98
Left Distance = 25.50, Right Distance = 37.51, Steering = 97.37
Left Distance = 24.58, Right Distance = 37.51, Steering = 98.02
Left Distance = 22.33, Right Distance = 37.51, Steering = 99.59
Left Distance = 20.87, Right Distance = 37.41, Steering = 100.54
Left Distance = 19.42, Right Distance = 36.86, Steering = 101.17
Left Distance = 19.42, Right Distance = 37.51, Steering = 101.63
Left Distance = 19.42, Right Distance = 37.49, Steering = 101.61
```

Conclusions and Discussion

1.1 Conclusions

- Impact of Battery SOC:
    - The state of charge of the RC vehicle battery impacted the throttle response of the vehicle. At a higher state of charge, the acceleration experienced for the same throttle input was higher.
    - As a result, the response of the PID controller for the given K_p, K_i and K_d parameters changed significantly.
- Throttle Dead Zone:
    - As discussed, the throttle input experiences a dead zone. Therefore, it is essential to identify the range of the dead zone and incorporate that information into the controller.
- Data Type for Variables:
    - An interesting find was that the chosen data type for a variable can significantly impact results. For example, if the variable corresponding to throttle input is taken as an integer, the throttle values keep decreasing for a constant obstacle distance. Whereas, using a float variable avoids this issue.
- Sensor Noise:
    - The ultrasonic sensors are prone to noise and can adversely impact throttle response if measured distance has very high inaccuracy.
- Power Source for Steering Servo
    - Power the steering servo motor using the Arduino introduces noise in the ultrasonic sensor readings whenever the servo draws current from the Arduino. Therefore, it is recommended that the servo is powered from the ESC.

1.2 Discussion

Even though the PID controller has proved to be a good implementation technique, we can obtain more robust control using model-based controllers. Controllers such as Model Predictive Control (MPC) can provide better results.

Apart from the model-based controllers, other non-model-based controllers such as Reinforcement learning and Supervised learning techniques can also provide robust control. The only challenge with these approaches is that they require significant amount of training data to perform well.