Line follower robot

Oct 18, 2018
Course: ENGR 22
Section: 2314

Instructor:
Tram Dang

Team members:
Zayd Sikder
Raul Gonzalez
Daniel Kaitel

Table of Contents

Team members

Zayd Sikder Daniel Kaitel Raul Gonzalez

Summary

This project focused on making a robot follow you if you were a line. The idea was to make a robot car that could follow a black line along the floor and it make go through white/black crosswalks with ease. To accomplish this, the robot car auto-adjusted its wheels through a PID loop using the arduino microcontroller and arduino software, so that more torque would act on the car if it deviated further from the desired path (a black line). We achieved this goal by breaking the task into parts, some of these parts were building circuits, setting up the car, coding, fine tuning the code, and finally adding cool extra features. At the end it worked out well, we achieved our goal and accomplished the task at hand while making the project look cool. See Figure 1 and Figure 2.

Figure 1. Iteration of the line follower robot

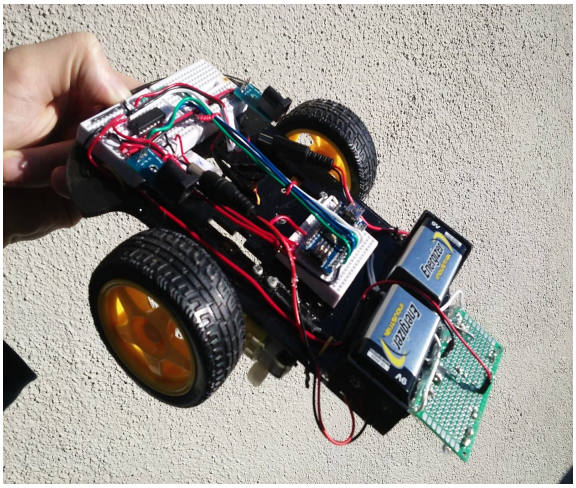
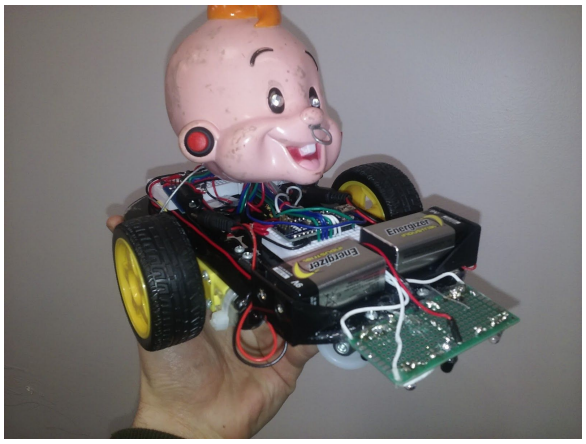


Figure 2. Final version of line follower robot.



Design problem and objectives

Our project focuses on the design and construction of a small robot capable of detecting and moving along a dark line when placed on a reflective surface.

This type of robot is know as a line follower and is widely used as an introductory tool to teach electronics and the basics of self guiding vehicles. It is based on the fact light reflects off lighter surfaces with greater intensity as compared to darker surfaces. This behaviour can be exploited to track the position of an object relative to two contrasting surfaces, and is exactly what is used by a line following robot to keep itself atop a line.

In the interest of pragmatism we limited the scope of the project by constraining the design through the following conditions:

1) The robot shall be based on a small car powered by electric motors.
2) It shall use a battery (or batteries) of readily available voltage and size.
3) The electronic components shall be of standardized size.
4) Infrared light emitting diodes and photodiodes shall be used to detect the line.
5) The robot shall be able to detect the presence of a black line, proceed to move forward, then stop once it reaches the end of the line.
6) Every component shall be off the shelf and must be easily bought off the internet.

By using electric power and off the shelf parts we limited the cost of the robot. Electric motors are simple, clean and easily controlled through a microcontroller. Likewise, alkaline batteries are clean and safe energy sources that require no special equipment or circuits to use. All of these components are inexpensive and have complementing characteristics which decreases the number of parts and thus the total cost.

Detailed design documentation

The entire project was partitioned into four separate categories:

1) Design and construction of the infrared light sensor
2) Design and implementation of an H-bridge to control the direction and speed of two motors.
3) Combining the light sensor and H-bridge with a motorized platform.
4) Programming steering and enabling line tracking through a PID control loop.
5) Programming vehicle forward speed to enable line following.
6) Testing the performance of the car with simple line circuits and iterating through different tuning parameters.
7) Finally, adding cool features such as playing sound with a huge array of tones/frequencies and adding an RGB LED that constantly changed color.

An overview of each section is described below:

1. The light sensor was tackled by considering it two seperate sections. One that would focus on the emission of infrared light (the emitter section), and one that would receive this light and generate an easily measured voltage (the receiver section).

   We first tackled the emitter section. This was no more than a simple infrared LED and a resistor. By choosing a specific resistor value, we were able to maximize the brightness of the emitted light without damaging the LED at our chosen operating voltage.

   The receiver section came next. This was a more complex problem than the emitter. The key to this circuit was understanding a photodiode and its properties. By substituting the resistor of a voltage divider with a photodiode, and then setting to the resistance of the remaining resistor to an appropriate value, we were able to create a light sensitive circuit.

   Combining these two separate circuits into one, referred to as a *module*, we had a system that would allows us to detect light that was created by the emitter, and then reflected back into the receiver. We then took two modules, separated them by the thickness of the black line we expected to track, and permanently attached them to a circuit board. This new system was the completed light sensor.

2. Our initial decision for motor control was to use a single TIP120 transistor for each motor which would allow the motors to spin in one direction, but this idea was scrapped in favor of a full H-bridge. An H-bridge allows the motors to spin in both directions by reversing the direction in which current travels through the motor by acting as a series of switches. Depending on which TIP120s were activated, the motors direction would change. We made this choice so we would have more options as to how the robot could move. The

original implementation of the H-bridge was done through the use of four TIP120 transistors per motor wired in the formation above. This method proved viable, but we later chose to swap our large and messy circuit for an integrated circuit which works on the same principles. This greatly reduced the size of our H-bridge while still maintaining differential steering on our robot.

3. Once the light sensor and motor controllers were complete we focused on attaching them to our motorized platform. We placed the light sensor at the front of the vehicle where it could predict the displacement of the line in advance of the wheels, and then attached it using hot-glue to the frame. The H-bridge was placed at the rear of the vehicle where it could be easily connected to the motors, and then attached to the frame using double sided foam tape. Lastly, we attached the Arduino microcontroller to the center of the frame using double sided tape. This way the Arduino could easily interface with the other components of the vehicle.

4. We next proceeded create a way to guide the car using differential steering. That is to say, to cause one wheel to turn faster than the other and thus induce a rotation. We did this by programming a function that sent on average a positive voltage to one of the motors, and a negative voltage to the other. Anticipating the need to set forward speed, we also added a parameter to bias the voltage of both motors towards a positive or negative voltage. This way, if commanded, both motors could spin in a common direction, albeit with one spinning faster than the other.

   With motor control complete, we proceeded to implement a simple PID loop. For our error term, we used the *difference* in the signal sent by each module of the light sensor. This way, we could determine how offset a black line would be from the mid point between the two modules. If the line was biased to one side, one module would receive more reflected light, and the difference between signals would increase. The *sign* of the difference merely specified the direction of this offset. With a proportional term ready, we created two subroutines to calculate the time derivative and time integral of the error. Once these functions were able to output smooth and consistent results, they were combined into a single value that was fed into the motor control function. The PID loop was complete.

5. With the functions required to track a line ready, we continued by programming more functions to permit forward motion. To do this it was necessary to determine whether the car was in the presence of a black line. This naturally required a comparison with a surface which was *not* black. We determined this reference with a function that took the average over an interval of time of the light sensor signals. If the immediate sensor readings dropped below this “reference value”, we knew the presence of a dark line and could begin moving forward. However, to avoid a rapid increase in speed which could damage the motors or cause unknown problems, we chose to implement a smoothing

function that would slowly increase motor speed until reaching a steady value.

6. With the line follower constructed and the essential functions completed, we could begin tuning of the PID coefficients and the forward speed. We first disabled the forward speed functions and focused solely on line tracking. This process was based on entirely trial and error and required iterating through different coefficient values. First we increased the proportional gain until oscillations were visible, then increased the derivative gain to add stability, and finally raised the integral grain. We repeated these steps many times to find a compromise between a high reaction time and a lack of rebound that was acceptable.

   Next we focused on tuning the forward speed. In this case, there were two important parameters: the rate at which the velocity rose and the final velocity of the car. The rate needed to be slow enough to provide a smooth increase in speed yet not be unnecessarily slow. After some trials an acceptable value was found. Then came the final velocity. This was much more of a compromise. If the value was too large then the car would not react quickly enough to curves and skip the line. If the forward speed was very low then it could potentially be higher and still permit successful line tracking. Through iteration, a value for the final speed was found that struct a compromise between rapidity and line tracking.

   By the end of these adjustments it became clear that a form of modulation was required to reduce the speed of the car as it entered and maintained a turn. This would enable the car to speed up on straight line sections and slow down during turns, thus providing it with more time to track said line when it most needed to. A function was coded to provide this modulation.

7. At first, putting an RGB led and a music file seemed like an easy task. However, doing this we were faced with many difficulties. Some of these difficulties was how we ran into storage space issues and how we would be able put a whole music file on the arduino. Adding an RGB LED also required that we power the leads backwards such that one pin had 3.3v going through it, while the other pins were attached to variable signals that went to ground.

8. Sadly, solving the storage space issue was not easy because we would need to invest in another part and an sd card due to the arduino only have 32kbs of storage. We are cheap so we used a very short sound sample. This was an evil laugh that lasted for 4 seconds that was converted into a 1D array with an 8kb ratio (sound quality). The sound was played by changing the frequency of a square wave fed to a small speaker. This 4 second audio clip took 55% of the storage. We were left with 45% of our storage to implement the RGB LED light changing code.

   Due to our RGB leds sharing a common anode, we were unable to directly control the led from the arduino. In order for us to control the led colors, we used NPN transistors on

each of the LED's three positive leads. This allowed us to selectively connect the LED's leads to ground, which provided us with basic color control. In order for us to get fine color adjustment, we resorted to using PWM to simulate different voltage levels through each of the LED's pins.

Throughout the design process we were assisted by a theoretical analysis of the problem. However determining the behavior of the car presented a case of *circular logic* as we were trying to predict behaviour *we had yet to program*. In light of this, we did find a case that was possible to analyse.

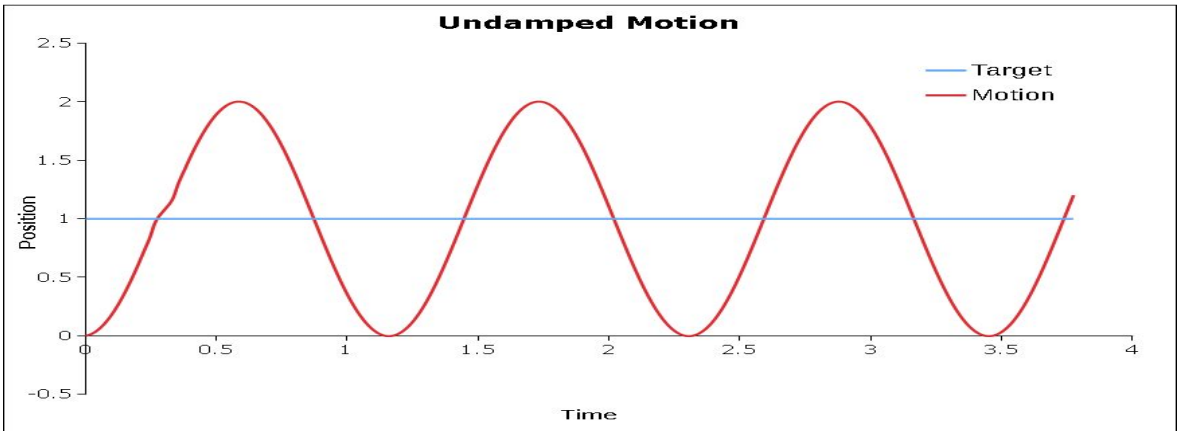
When we consider the kinematics of the car we find there are only two degrees of freedom. For translations, we can assume that static friction will act as a constraint and only the direction in which the wheels roll permits motion. Likewise, as the wheels can turn at different speeds there is an additional degree of freedom about rotation. This is to say, the car can either translate forward and aft or steer sideways. By making an additional assumption the analysis could be simplified: forward motion induced no torque on the vehicle, and so any translation had no effect on the rotation of the car. This way each degree of freedom could be analyzed separately.

Through this isolation we could apply a well known model to the rotation of the car: the step response of a PID loop as a function the term coefficients. It must be mentioned that the control loop is being applied to a *second order system* as angular acceleration is the *second derivative* of angular position. This way we could take into account the effect of inertia on the vehicle's behaviour. Additionally the mass of the car was assumed to be constant as we did not expect it to change significantly even if we added or removed components.

For a fixed mass, some cases of this model stand out:

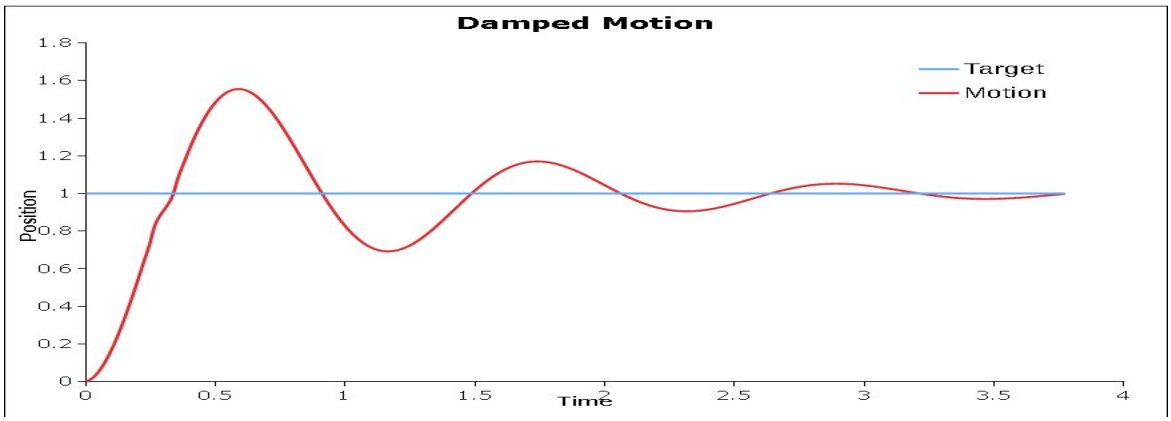
1) The derivative and integral term are zero, the proportional term is not: In this case, we see that the error (which represents position relative to the line being tracked) oscillates back and forth without decaying. See Figure 3.

Figure 3. Step response with no derivative term.



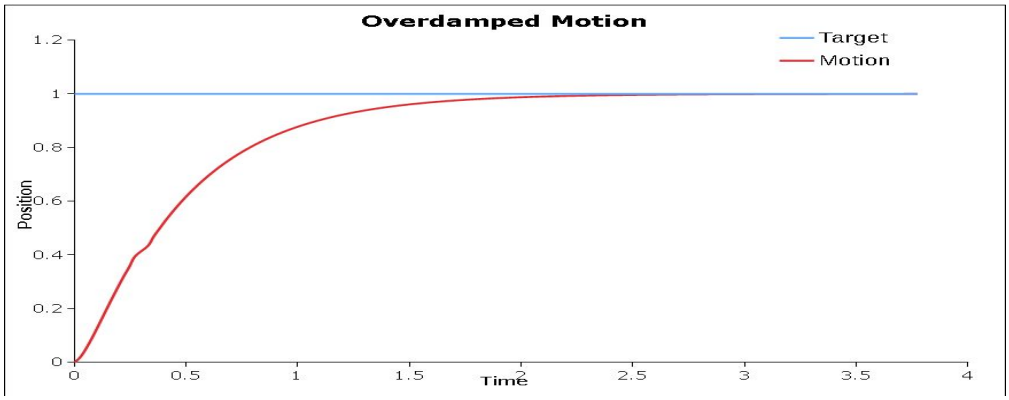
2) The integral term is zero. The proportional and derivative terms are not: We note the existence of a derivative term adds damping to the motion and the error decays with time. See Figure 4.

Figure 4. Step response with derivative term.



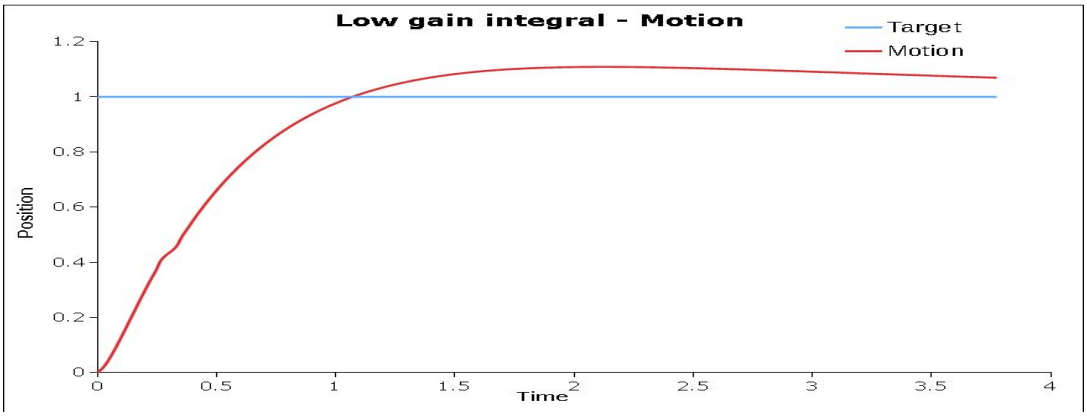
3) The integral term is zero, the proportional term is not, and the derivative term is large: Here the error decays to zero without oscillating. We note that in this case the motion is overdamped. See Figure 5.

Figure 5. Step response with large derivative term.



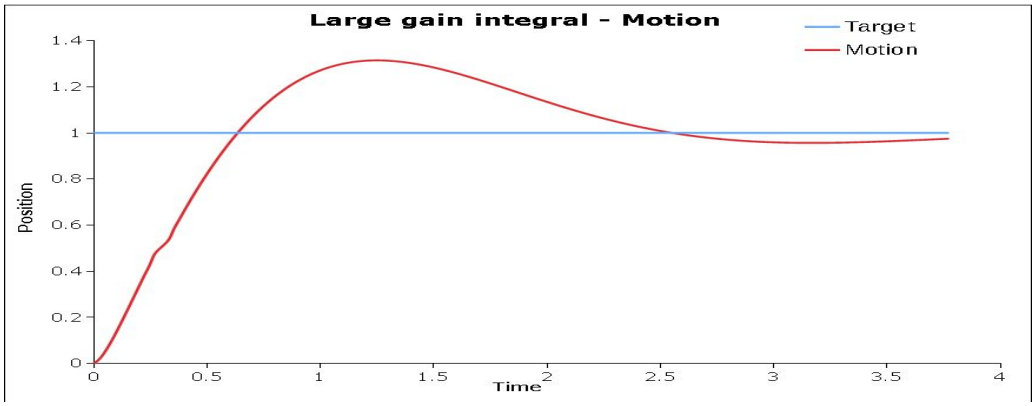
4) The integral and proportional terms are not zero. The derivative term is large: Even though the motion is overdamped, the integral term induces a slowly decaying but noticeable overshoot. See Figure 6.

Figure 6. Step response with overshooting integral term.



5) The proportional term is not zero. The integral and derivative terms are large: For this situation we notice the overshoot induced the integral is worsened and no longer decays after one long oscillation cycle. Rather, the oscillation has many cycles. See Figure 7.

Figure 7. Step response with oscillating integral term.



We can compare these cases to an ideal behaviour of our choice. While we would like overdamped motion to avoid any rebound and ensure crisp response, we can only achieve this without the presence of an integral term. Once this term is added a rebound, however small, will always be present. Another important observation is the necessity of a derivative term. Stable and decaying motion for large coefficient values is only possible if the term is present, and so it requires special attention in order to achieve maximum performance.

Additional analysis of the PID model revealed the benefit of taking the *second time derivative* of the error. It would allow to us determine the *acceleration* of the car relative to the *line* that it was tracking (as the error we were measuring was the position of the car relative to the line). The usefulness of this parameter comes from the way in which it can be incorporated into the PID loop as an additional term to reduce rebound. This becomes obvious if we consider the PID output as a force acting on the car. See Equation 1.

$$F_{PID} = \left(K_p e(t) + K_d \frac{de(t)}{dt} + K_I \int_0^t e(t)dt \right) + K_{d2} \frac{d^2 e(t)}{dt^2} \qquad \text{: Equation 1}$$

If this force is the only external force acting on the car, then by using Newton's second law we have: (See Equation 2)

$$F_{PID} = \sum_{i=1}^{n} F_i = ma = m\frac{d^2 x}{dt} \qquad \text{: Equation 2}$$

If we assume that the position of the car *x* is the same as the measured error *e(t)* then: (See Equation 3)

$$m\frac{d^2 e(t)}{dt^2} = K_p e(t) + K_d \frac{de(t)}{dt} + K_I \int_0^t e(t)dt + K_{d2} \frac{d^2 e(t)}{dt^2} \qquad \text{: Equation 3}$$

↓

$$0 = K_p e(t) + K_d \frac{de(t)}{dt} + K_I \int_0^t e(t)dt + (K_{d2} - m)\frac{d^2 e(t)}{dt^2}$$

We can see that addition of a *second order derivative* to the PID loop has the same result as *adding or removing mass* from the system (so long as the error is a faithful representation of position). By *effectively removing mass* from the system we can expect the *rebound to decrease* as the damping will be able to decelerate the *actual mass* faster, leading to a smoother and more damped response. The effect of an *effectively* lighter system is clearly seen in a plot of the step response. See Figure 8 and Figure 9.

Figure 8. High mass response.

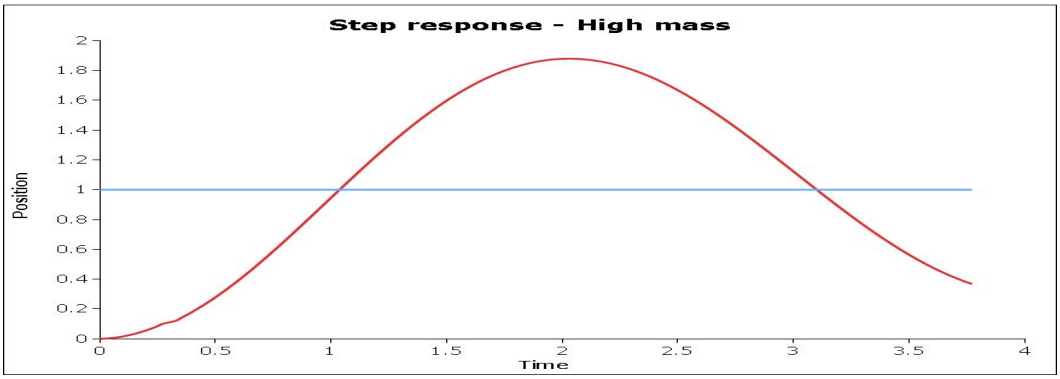
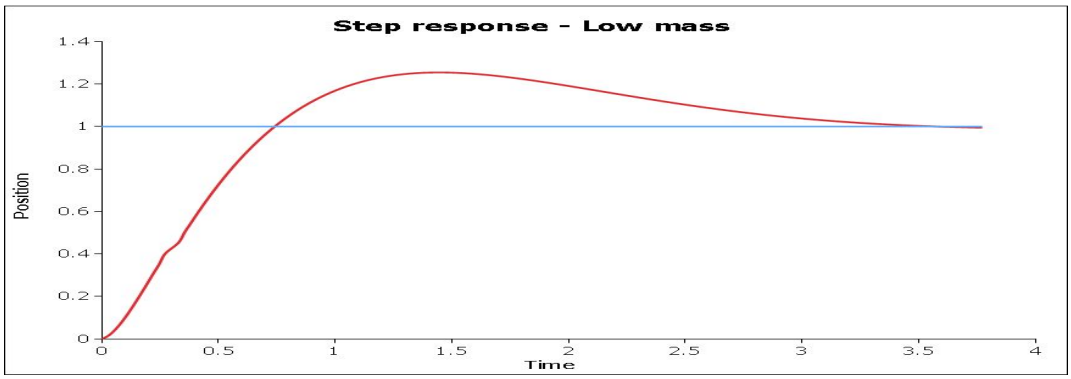


Figure 9. Low mass response.



With a new appreciation of the physical behaviour we could expect of the car, we focused our efforts on designing the circuits required to control it. These were mainly the light sensor and H-bridge, in addition to the connections to the arduino microcontroller. Conceptually the system was designed to work as follows:

The light sensor generated two signals which were fed to the Arduino. The microcontroller then interpreted this data and generated a series of output signals that were fed to the H-bridge. The H-bridge then responded to these signals by causing the motors to turn at a specific rate. The

signals that were interpreted and sent through the entire system were then completely dependant on the measurements taken by the sensor. This process is best illustrated with a schematic of the whole circuit. See Figure 10 and Figure 11.

Figure 10. Connections between Arduino and H-bridge.

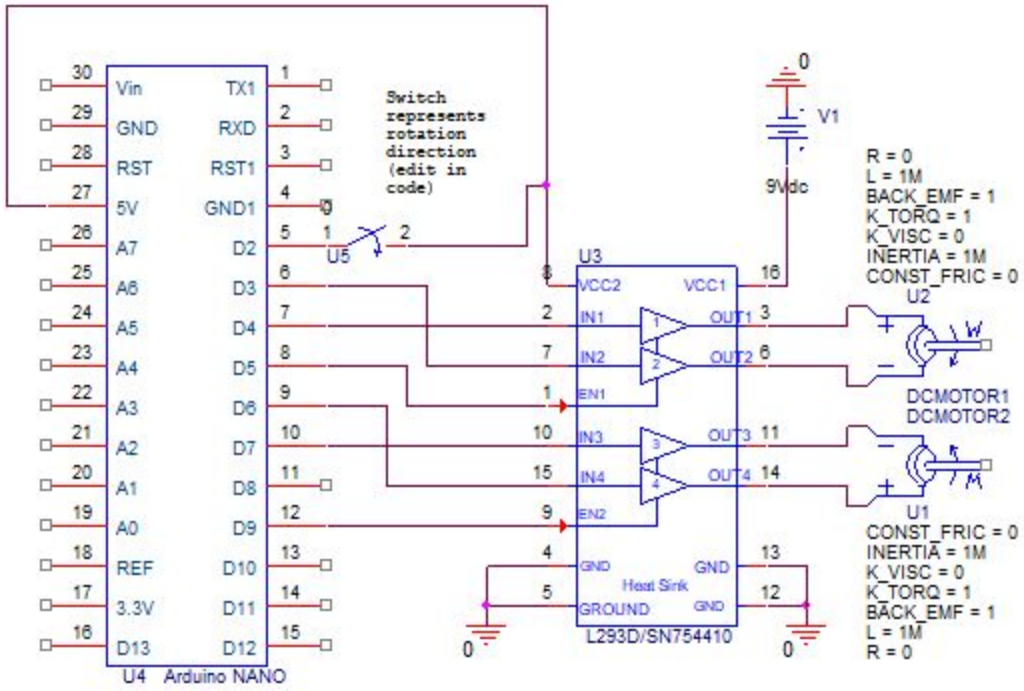
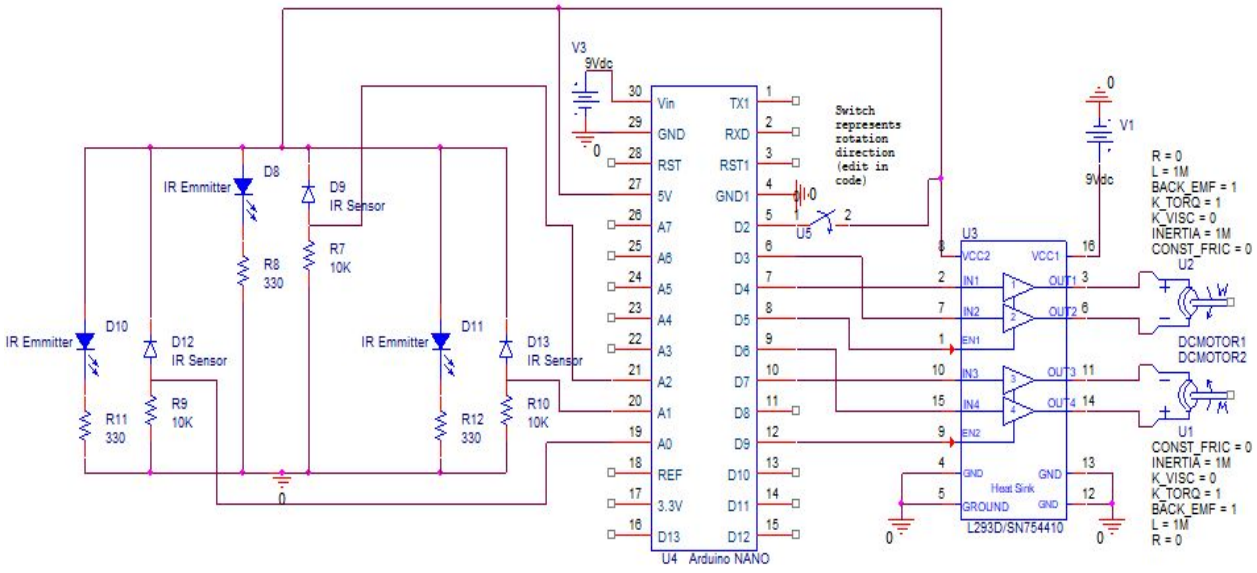


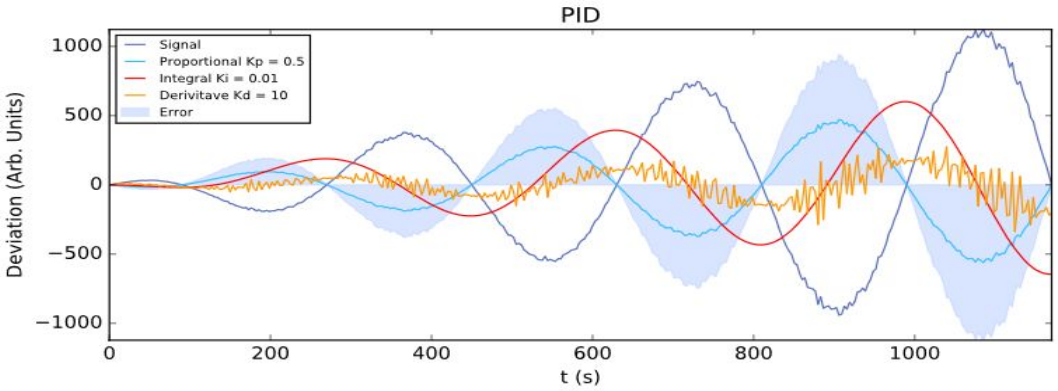
Figure 11. Connection between line sensor, arduino and H-bridge.



Laboratory test plans and results

The process of designing a PID controller is superficially a simple task. However, its actual implementation soon faces important challenges. Foremost in these is constructing calculus functions that output the required integral and derivative terms. We felt confident using standard numerical methods, like *finite differences* and *Newton-Cotes quadratures*, to *approximate* the derivative and integral of our assumed input function. While all seemed well in theory, in practise the results were completely different. When subroutines were finally coded and tested using these methods the output was very different from what we expected. See Figure 12.

Figure 12. Signal quality of simple differentiation and integration.



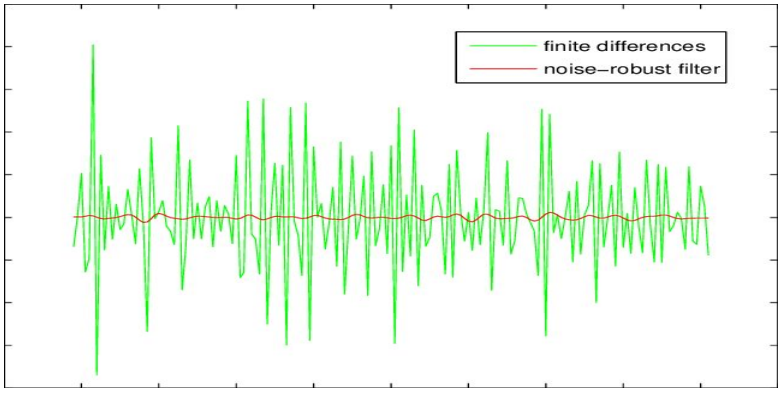
It was very obvious that the derivative function was extremely noisy could barely be described as continuous. By contrast, the integral function was much smoother and its output had few spikes and distortions. It became clear that differentiation *amplified* the noise in the input signal while integration *suppressed* the noise.

In order to overcome to the problem of noise amplification from differentiating, an effort was made to find an easy to implement yet effective solution. This culminated in the use of a *finite impulse response filter*, more specifically a *Savitzky-golay filter*. In brief, it functions by taking a set of data and finding the least squares regression of a degree *N* polynomial, and then taking the *derivative* of this polynomial and returning the output of the derivative at some input value. By doing so, the problem of noise sensitivity is greatly reduced as the least squares regression naturally suppresses sudden fluctuations.

These characteristics were very appealing but so was the implementation of the filter due to its simplicity. Being a *finite impulse response* filter, it is based on the concept of *discrete convolution*, which is simply the summation of different data samples multiplied by known coefficients. For the Savitzky-Golay filter these coefficients are constant and so the implementation was even simpler, being nothing more a *weighted average*.

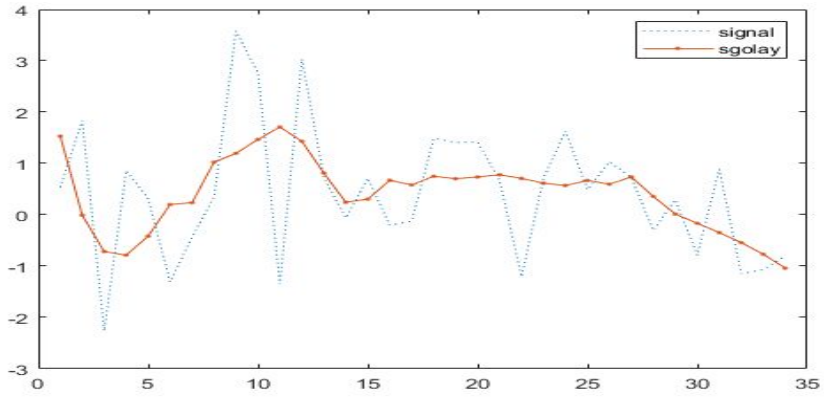
Once this was understood, we coded two variants of the filter: one to *smooth* the raw data and remove high frequency chatter, and one to take the time derivative. The results were *far better* than anything *finite differences* returned. There were *very* few sudden spikes in the derivative that were *not* expected. See Figure 13.

Figure 13. Finite difference equation compared to Savitzky-Golay derivative.



The smoothing variant of the filter also had welcome results. It clearly removed a great deal of the noise in the raw data and appeared much better suited as an input for the PID loop. See Figure 14.

Figure 14. Effect of Savitzky-Golay smoothing filter.



With the PID terms having acceptable behaviour, we shifted towards using them to guide the car. We accomplished this tuning the control loop through a process of *iteration*. That is to say, a series of short tests followed by a small modification to the programming. These steps were repeated in quick succession in order to converge towards a result that satisfied the design objectives. We made the deliberate choice to focus the iteration solely on programming changes rather than mechanical or electrical modifications. These type of changes required far more work and drastically slowed down the process. Consequently we kept work on the physical

components to a minimum so it did not become a major distraction. We then proceeded by separating the iteration process into two sections: Line tracking and line following.

Line tracking focused entirely on the robot's steering and its ability to keep itself above a black line. As we assumed that the rotation of the car was independent of its translation, we were comfortable adjusting the steering without forward motion. By performing a series of disturbance tests, where we abruptly moved the black line relative to the robot, and then observing the results, we were able to fine tune the PID coefficients. The abrupt displacement mimicked a step input and so the motion of the car would be similar to the step response we had previously studied. We compared the behaviour with our idealized cases and adjusted the coefficients accordingly, to achieve either a faster or slower response during the *next* test.

Next came the line tracking. For this section we adjusted the forward speed of the robot by testing it on a simple closed circuit. We placed the car over a section of the circuit and allowed it to move forward. If the car, for whatever reason, failed to complete the circuit, the parameters dictating forward speed were modified. We repeated these steps until the car could smoothly and consistently go around the entire track.

If the car accelerated too quickly at the beginning of the circuit and skipped the line, we programmed it to accelerate slower. Likewise, if the car was unable to enter and maintain a turn successfully, we either programmed it to brake harder during turns or decreased its maximum speed altogether. Contrary to this, if the car seemed capable of moving faster, we programmed it to increase its speed or to decreased its braking. If it appeared possible to increase the speed, we kept doing so until the car was no longer able to *consistently* follow the circuit, at which point we reverted to the last successful iteration and stopped modifying the code.

Bill of Materials

Category	Component	Quantity	Unit cost ($)	Cost ($)
Light sensor	¼ w 10KΩ Resistor	3	0.58	1.74
	¼ w 330Ω Resistor	2	0.58	1.16
	5mm Infrared LED (10 pcs)	1	12.99	12.99
	5mm Infrared receiver (10 pcs)			
Motor control	SN754410NE dual H-bridge	1	2.35	2.35
	Arduino Nano Clone	2	3.0	6.0
Miscellaneous	Robot car kit (includes motors)	1	14.99	14.99
	9v Alkaline battery	2	0.89	1.78
	Multithreaded jumper wires (20 pcs)	1	1.95	1.95
	Insulated hook up wires (Kit - 4pcs)	1	12.99	12.99

	Miniature breadboard (3 pcs)	1	6.99	6.99
Lighting	1w NPN transistor	3	0.3	0.9
	5mm (multicolor) RBG LED	2	0.39	0.78
Construction	Hot glue stick (30 pcs)	1	3.97	3.97
	Hot glue gun	1	7.99	7.99
Music	10mm Buzzer (5 pcs)	1	5.00	5.00
Total cost				81.58

Conclusion

The line follower was able to effectively follow a smooth circuit draw on a highly reflective background. In addition to this, the speed of the vehicle could be raised or lowered and it would still be able to follow the circuit. However for very sharp turns, such as 90 degree bends, the car would skip the line if moving fast enough.

The car was also able to react to sudden disturbances (such as being tapped by hand) and return to a steady path without oscillating or becoming unstable. This was especially evident during sharp turns where the line itself was the source of disturbance. The car would smoothly enter and exit a turn without jittering or oscillating. Additionally if the line formed an open circuit, the car was able to smoothly detect the line, accelerate, follow it, and then bring itself to rest once it reached the end of the circuit.

Even though we completed our assignment, there were many problems we did not solve. One of these challenges is what to do when the light sensor is no longer on the black line. If we added an opto interrupter and traced back the position of the car, it would be possible for it to return to where it started. By also adding ultrasound sensors the car could detect impending collisions and prevent them. Additionally, we could have added an optoisolator/optocoupler to isolate the arduino from potential back EMF from the motors and prevent any long term damage.

Acknowledgements

We would like to thank professor Dang for teaching us to the concepts necessary to succeed at this project. Her instruction was invaluable at piecing together the electronic components to make a working circuit. We also thank her patience with our many questions and greatly appreciate her efforts.

References

Denizen. “PID Simulator - Free Tools Collection.” *Automation Forum*, 8 Nov. 2015, automationforum.in/t/pid-simulator-free-tools-collection/557.

“PID Demo .” *EStuffz*, Google Sites, sites.google.com/site/fpgaandco/pid.

“PID Tutorial.” *KLiK ROBOTICS*, www.klikrobotics.com/PIDTutorial.html.

Lohninger, H. “Moving Average .” *Fundamentals of Statistics*, 8 Aug. 2012, www.statistics4u.com/fundstat_eng/cc_moving_average.html.

Lohninger, H. “Filters - Mathematical Background.” *Fundamentals of Statistics*, 10 Aug. 2012, www.statistics4u.com/fundstat_eng/cc_filter_math.html.

Lohninger, H. “Savitzky-Golay Filter” *Fundamentals of Statistics*, 10 Aug. 2012, http://www.statistics4u.com/fundstat_eng/cc_filter_savgolay.html.

“Savitzky–Golay Filter.” *Wikipedia*, Wikimedia Foundation, 2 Dec. 2018, en.wikipedia.org/wiki/Savitzky–Golay_filter.

Holoborodko, Pavel. “Smooth Noise-Robust Differentiators.” *Applied Mathematics and Beyond*, Wordpress, 2008, www.holoborodko.com/pavel/numerical-methods/numerical-derivative/smooth-low-noise-differentiators/.

Holoborodko, Pavel. “Central Differences.” *Applied Mathematics and Beyond*, Wordpress, 2009, http://www.holoborodko.com/pavel/numerical-methods/numerical-derivative/central-differences/

Paramonov, Andrey. “Noise-robust smoothing filter.” *Applied Mathematics and Beyond*, Wordpress, 2015, http://www.holoborodko.com/pavel/numerical-methods/numerical-derivative/central-differences/

“Finite Difference.” *Wikipedia*, Wikimedia Foundation, 30 Oct. 2018, en.wikipedia.org/wiki/Finite_difference.

“Finite Difference Coefficient.” *Wikipedia*, Wikimedia Foundation, 22 Nov. 2018, en.wikipedia.org/wiki/Finite_difference_coefficient.

“Newton–Cotes Formulas.” *Wikipedia*, Wikimedia Foundation, 19 Nov. 2018, en.wikipedia.org/wiki/Newton–Cotes_formulas.

Holoborodko, Pavel. “Stable Newton-Cotes Formulas.” *Applied Mathematics and Beyond*, Wordpress, 2011,

http://www.holoborodko.com/pavel/numerical-methods/numerical-integration/stable-newton-cotes-formulas/

Adeed, Samer. "*Introduction to Numerical Analysis: Numerical Integration"*, University of Alberta, sameradeeb-new.srv.ualberta.ca/introduction-to-numerical-analysis/numerical-integration/.

"Numerical Integration." Uni Study Guides, Wikimedia, 23 Oct. 2015, www.unistudyguides.com/wiki/Numerical_Integration.

Appendices

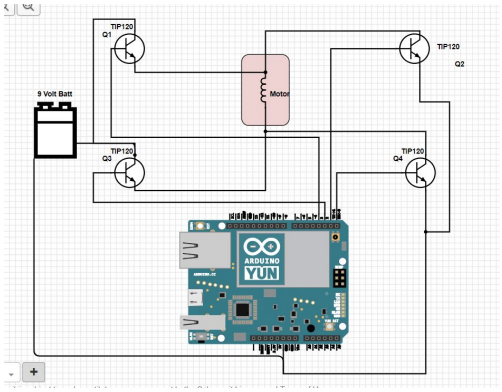
Miscellaneous: RBG LED with 3.3V running through it: Red pin has resistor value 165 ohms and the Blue pin/Green pin has resistor value of 110 ohms.

LED and Sound generation code:

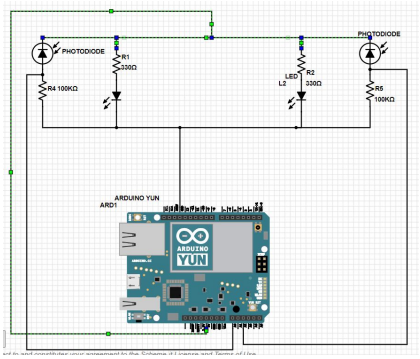
```
#include <PCM.h>

int greenLedPin1 = 3;              // Pin Green LED is connected
to

int blueLedPin1 = 5;       // PIN Light Sensor is connected to

int redLedPin1 = 6;



const unsigned char sample[] PROGMEM = {

ARRAY OF CODE

};

void setup()

{

   pinMode(greenLedPin1, OUTPUT);

 pinMode(blueLedPin1, OUTPUT);

 pinMode(redLedPin1, OUTPUT);

 }

void loop()

{

 while(true){

  startPlayback(sample, sizeof(sample));

  analogWrite(greenLedPin1, (sin( millis()/500.0 ) +
1)*255/2.0);

  analogWrite(blueLedPin1, (sin( millis()/500.0 + PI/3) +
1)*255/2.0);

  analogWrite(redLedPin1, (sin( millis()/500.0 + 2*PI/3) +
1)*255/2.0);

  delay(4000);

 }

}
```

Initial circuit diagrams:

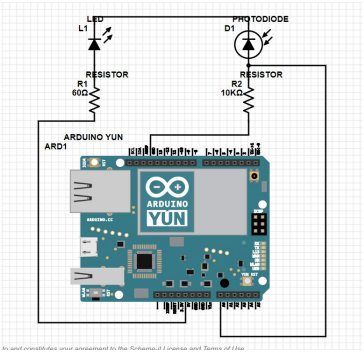
Controlling one motor with tip120:



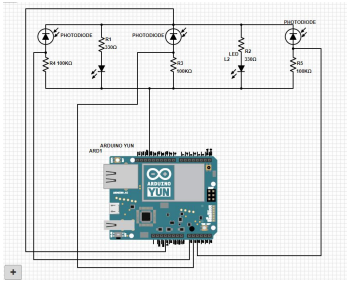
Photodiode second iteration circuit:



Photodiode first iteration circuit:



Photodiode Final iteration circuit:



Sensor interpretation and motor (line following) code:

```
//------ Output Pin numbers -------

        //Left motor:
const int INPUT_PIN_L = 6;
const int PIN_A_L = 7;
const int PIN_B_L = 8;
        //Right motor:
const int INPUT_PIN_R = 3;
const int PIN_A_R = 4;
const int PIN_B_R = 5;
        //Analog inputs: [sensor]
const int PIN_SENSOR_A = A0;
const int PIN_SENSOR_B = A1;


//-- Motor control:
const int PWM_MAX = 255;
const uint32_t ON_INTERVAL = 7000;
const float INPUT_MIN = 5.0;
const int MOD_TRANSITION = 120.0;


//-- Proportional - Derivative parameters:
const float EXP_SMOOTH = 0.4;
        //Derivative filter:
const float COEFF_DEV[] = {5, 8, 27, 48, 42, 0, -42, -48, -27,
-8, -5};
const float DENOM_DEV =  512.0;
const int N_STORED = sizeof(COEFF_DEV)/sizeof(
COEFF_DEV[0] );
        //2nd derivative filter:
const float COEFF_DEV_2[N_STORED] = {15, 6, -1, -6, -9,
-10, -9, -6, -1, 6, 15};
const float DENOM_DEV_2 = 429.0 ;
        //Proportional filter
const float COEFF_PRO[N_STORED] = {-4, -20, -20, 80,
280, 392, 280, 80, -20, -20, 4};
const float DENOM_PRO = 1024.0;
        //Integration of sensor [Quadrature]
const float COEFF_INT[] = { 7, 12, 15, 16, 15, 12, 7 };
const float DENOM_INT = 14.0;
const int N_INT = sizeof(COEFF_INT)/sizeof(
COEFF_INT[0] );

        //Control coefficients:
const float K_PROP = 0.25; // 0.25;
const float K_DEV = 5.0; // 6.0;
const float K_DEV_2 = 20.0; // 20.0;
const float K_INT = 0.0005;
// Anti wind-up limit:
const float INT_LIMIT = 100.0;
// Threshold to enable integration
const float INT_ENABLE = 0.1;


//-- Forward throttle parameters:
const int N_INITIAL = 1000;
const int FOR_THROTTLE = 150;
const float DECAY = 0.01;
const float PERCENT_INITIAL = 0.9;
const float MOTOR_BIAS = 0.01;
// Threshold when throttle is reduced
const float TURN_LIMIT = 20000;


        //-- Global variables:
float past_average[N_STORED] = {0};
int input_initial = 0;
int diff_initial = 0;
float forward = 0;
//------> Global pulse classes <------ [See below the variables]



//------ Functions -------


//----- Motor control: --

//-- Class for pulse-frequency modulation:
class pulse {

  public:

        int freq_mod;

        //Constructor
        pulse( void ) {
        last_time = 0;
        freq_mod = 0;
        off_interval = 0;
        }
        //Destructor
        ~pulse ( void ) {}

        //---

        void set_period( float input ) {
        if( input < INPUT_MIN ) {
        off_interval = ON_INTERVAL*(
PWM_MAX/INPUT_MIN - 1 );
        } else {
        off_interval = ON_INTERVAL*( PWM_MAX/input -
1 );
        }
        }

        //---

        void check_time( void ) {

        float interval = micros() - last_time;

        if( interval <= (ON_INTERVAL + off_interval) ) {
        if( interval <= ON_INTERVAL ) {
        freq_mod = PWM_MAX;
        } else {
        freq_mod = 0;
        }
        } else {
        last_time = micros();
        }

        }
```

```
  private:
            uint32_t last_time;
            uint32_t off_interval;

};


//-- Function to set motor throttle and direction:
void set_motor( int input, int pin_input, int pin_a, int pin_b ) {

  analogWrite( pin_input, abs(input) );

  if( input > 0 ) {
            digitalWrite( pin_b, HIGH );
            digitalWrite( pin_a, LOW );
  } else {
            digitalWrite( pin_b, LOW );
            digitalWrite( pin_a, HIGH );
  }
}

//-- Function to identity sign:
int sign( int input ) {
  if( input < 0 ) {
            return -1;
  } else {
            return 1;
  }
}
            //Note: Turn > 0 implies right turn (positive X axis)

            //------> Global pulse classes <------
pulse wave_L, wave_R;

//-- Function to combine forward and directional inputs:
void set_speed( int forward, int turn ) {

  int input_L = constrain(  (1.0 + MOTOR_BIAS)*(forward +
turn ), -PWM_MAX, PWM_MAX );
  int input_R = constrain(  (1.0 - MOTOR_BIAS)*(forward -
turn), -PWM_MAX, PWM_MAX );

  wave_L.set_period( abs(input_L) );
  wave_R.set_period( abs(input_R) );

  if( abs(input_L) < INPUT_MIN ) {
            input_L = 0;
  }
  else if( abs(input_L) < MOD_TRANSITION ) {
            input_L = sign(input_L)*wave_L.freq_mod;
  }

  if( abs(input_R) < INPUT_MIN ) {
            input_R = 0;
  } else if( abs(input_R) < MOD_TRANSITION ) {
            input_R = sign(input_R)*wave_R.freq_mod;
  }

  set_motor( input_L, INPUT_PIN_L, PIN_A_L, PIN_B_L );
  set_motor( input_R, INPUT_PIN_R, PIN_A_R, PIN_B_R );
}



//----- Proportional-Derivative components: --

//-- Function to output difference of sensor inputs:
int sensor_diff( void ) {
  return ( analogRead(PIN_SENSOR_A) -
analogRead(PIN_SENSOR_B) );
}


//-- Function to get exponential average of sensor difference:
float exp_average_diff( void ) {
  static float val = 0;
  val += ( (sensor_diff() - diff_initial) - val )*EXP_SMOOTH;
  return val;
}


//-- Function to store averaged values in array:
void store_average( void ) {

  for( int index = (N_STORED-1) ; index > 0 ; index -= 1 ) {
            past_average[index] = past_average[index - 1];
  }
  past_average[0] = exp_average_diff();
}


//-- Function to calculate time derivative of difference
[averaged values]:
float smooth_deriv( void ) {
  float deriv = 0;
  for( int index = 0; index < N_STORED ; index += 1 ) {
            deriv += past_average[index]*COEFF_DEV[index];
  }
  return deriv/DENOM_DEV;
}


//-- Function to calculate second time derivative of difference
[averaged values]:
float smooth_deriv_2( void ) {
  float deriv_2 = 0;
  for( int index = 0; index < N_STORED ; index += 1 ) {
            deriv_2 +=
past_average[index]*COEFF_DEV_2[index];
  }
  return deriv_2/DENOM_DEV_2;
}


//-- Filter to remove noise from sensor difference [averaged
values]:
float smooth_prop( void ) {
  float prop = 0;
  for( int index = 0; index < N_STORED ; index += 1 ) {
            prop += past_average[index]*COEFF_PRO[index];
  }
  return prop/DENOM_PRO;
}


//-- Function to get quadrature (time integral) of the average
sensor difference: [Note: can be disabled through the
enabled input]
float quadrature( float enable ) {

  static float short_term_int  = 0;
```

```
  static float long_term_int = 0;
  static int count = 0;

  //- Checking if integration enabled
  if( enable > INT_ENABLE ) {

            // Trapezoidal integration for immediate samples
            short_term_int += K_INT*( past_average[0] +
past_average[1] )/2.0 ;
            count += 1;
            // Newton-cotes quadrature for previous samples
[long term integral]
            if( count == (N_INT-1) ) {
            count = 0;
            short_term_int = 0;

            float accum = 0;
            for( int index = 0; index < N_INT ; index += 1 ) {
            accum +=
past_average[index]*COEFF_INT[index];
            }
            long_term_int += K_INT*accum/DENOM_INT;
            }

            //- Anti Wind-up code: Limits integral from
increasing beyond bounds
            float integral = (short_term_int + long_term_int);

            if( abs(integral) > INT_LIMIT ) {
            long_term_int = sign(integral)*INT_LIMIT;
            return long_term_int;
            }
            else {
            return integral;
            }
            //- Default if integration is disabled
            } else {
            short_term_int = 0;
            long_term_int = 0;
            count = 0;
            return 0;
  }
}



//----- Sensor calibration and determining forward motion: --

//-- Average of sensor inputs:
float av_input( void ) {
  return  ( analogRead(PIN_SENSOR_A) +
analogRead(PIN_SENSOR_B) )/2.0;
}


//-- Blinking sequence for Pin 13 LED:
void blink_led( int count ) {
            for( int index = 0; index < count ; index += 1 ) {
            digitalWrite( LED_BUILTIN, HIGH );
            delay( 100 );
            digitalWrite( LED_BUILTIN, LOW );
            delay( 100 );
  }
}
```

```
//-- Function to generate calibration value for sensor
average:
void initialize( void ) {

  blink_led( 10 );

 //- Calibrating sensor average:
  int64_t average = 0;
  for( int index = 0; index < N_INITIAL ; index += 1 ) {
            average += av_input();
  }
  input_initial = average/N_INITIAL;

 //- Calibrating sensor difference:
  average = 0;
  for( int index = 0; index < N_INITIAL ; index += 1 ) {
            average += sensor_diff();
  }
  diff_initial = average/N_INITIAL ;
  blink_led( 5 );
}


//-- Function to smoothly enable or disable forward speed:
void set_forward( void ) {
  if( av_input() < input_initial*PERCENT_INITIAL ) {
            forward = DECAY + forward*(1 - DECAY);
  } else {
            forward = forward*(1 - DECAY);
  }
}


//- Function to reduce forward speed during sharp turns
float modulate_forward( void ) {
  float modulation;
  float var = abs( past_average[0] )*FOR_THROTTLE ;

  if( var > TURN_LIMIT ) {
            modulation = TURN_LIMIT/var;
  } else {
            modulation = 1;
  }
  return (forward*FOR_THROTTLE)*modulation ;
}

//------ Main functions:

void setup() {

  pinMode(LED_BUILTIN, OUTPUT);

  pinMode( PIN_SENSOR_A, INPUT );
  pinMode( PIN_SENSOR_B, INPUT );

  for( int index = 2 ; index <= 8 ; index += 1 ) {
            pinMode( index, OUTPUT );
  }

 initialize();
}

//-

void loop() {
```

```
 wave_L.check_time();
 wave_R.check_time();

 store_average();

 set_forward();
 set_speed( modulate_forward(), -smooth_deriv()*K_DEV -
smooth_prop()*K_PROP - smooth_deriv_2()*K_DEV_2 -
quadrature(forward) );

}
```