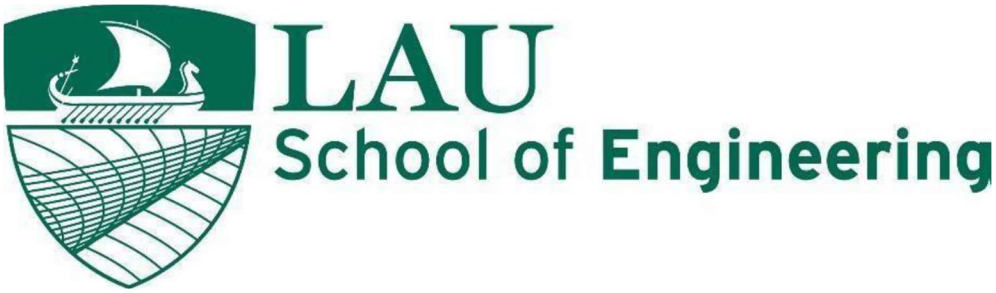
Lebanese American University

Electrical and Computer Engineering Department



Capstone Design Project II

Automated Bicycle

Advisor: Dr. Zahi Nakad

Edwin Odeimi 201601044

Charbel El Khoury 201603822

Ahmad Bitar 201507093

June 2nd, 2021

Abstract

The project we developed is a third-party instalment on any 6-gear bicycle as an external kit that extends the abilities of a regular bike significantly, namely adding a large motor as well as an automatic gear shifting system to relieve the user from pedalling and shifting gears all the time when commuting every day. Another feature is the ability for the bike to cruise at a user-defined speed to further relax responsibilities from the user. The autonomous bike also has an “Assistive” mode which senses when the user is pedalling, and gives him a little push from the motor, as well as having autonomous gear-shifting while in this mode of operation. This report aims at explaining our development process as well as our solutions when dealing with contingencies.

Acknowledgement

We would like to thank our Capstone II advisor Dr. Zahi Nakad for being with us throughout the semester whenever we needed help, and for being patient and professional whenever we needed to meet and discuss further developments in the project. His constructive comments and advice guided us and motivated us whenever we encountered a problem when developing the project. We would also like to thank the Electrical, Computer and Mechatronics Engineering departments faculty on their hard work ensuring that the students benefit from every course and advice through their university careers.

Table of Contents

Table of Figures

Table of Tables

Introduction

The project we built is composed of a user (pre-owned) bicycle, a large motor used for propulsion, a gear shifting mechanism, and a well-organized dashboard that should guide the user through his selection of mode of operation, as well as all pre-sets needed for the system to be autonomous.

E-Bikes are gaining popularity recently although their need for the consumer has never been agreed on by leading companies in the industry. This kit will attempt to help its consumers in their everyday commute to work or school, all the while attempting to reduce CO2 emissions since all components are electrical and the overall price of the product will be significantly appealing to both bike enthusiasts, as well as people who cannot afford a car. Till this day, we can find E-Bikes on the market that provide the regular manual motor speed and gear shifting controls, all the while being as expensive as a small car, or a large motorcycle. One example of such products that also implements autonomous gear shifting is KOGA E-Nova Automatic developed by KOGA [9], this E-Bike's retail price is marked as 27,999 Danish Krone, which converts to around 4,574 USD:

Figure 1 KOGA E-Nova Automatic by KOGA [9]

Our product will also help solve some local problems Lebanese citizens face daily, for example, one significant problem we might be reducing is the need for public transportations. As mentioned before, the kit we will provide might be appealing to people who cannot afford a car but need it at the same time. Therefore, it will reduce the need to use the local public transportation mediums that are unsanitary (certainly in these times with the Coronavirus Pandemic) and unsafe.

We achieved this final product by attaching a large motor (that will be discussed further in the background section as well as its own section) as a propulsion mechanism, linked to the rear wheel through a chain opposite to the gear shifting side. The motor's power is controlled electronically, and it will be powered with a 36 Volts power supply attached to the back of the bike along with the needed electronics on a cargo rack.

We are also using two sensors to detect each revolution of both the rear wheel and the pedals in order to find the system's speed and the user's rate of pedalling respectively (the sensors will be further discussed in the background and in the electronics preparations sections). These sensors will prove valuable when the system wants to autonomously shift gears or assign a new speed in an attempt to smoothly achieve a cruising speed that the user will specify on the interface panel provided to him on the dashboard (the dashboard will be discussed in the dashboard section).

The original gear shifting mechanism is still used the same way, but we altered the dynamic portion using our electronic and mechanical modifications to change gears electronically, allowing autonomous gear shifting. We implemented a mechanism to detect the current gear position as well.

Project Constraints

Power Constraint

We are using a large motor (the motor will be introduced in the background section and further discussed in its own section); thus, we can neglect the power consumption of the other components (electronics, relatively small stepper motors that we used for the dashboard and gear shifting – discussed in the background section). Since we are using a 450W Brushed motor, and three 12V batteries in series (discussed in the background section) with a capacity of 15 AH, then we can approximate that the system will last around 1 hour and 12 minutes (1.2 hours) at constant, full power use on a full charge.

Weight Constraint

Our product classifies as a vehicle; therefore, all components should be safely available on board when being used. We can neglect the weight of the electronics and 3D printed mounts (discussed in each section separately), and we can consider that the main weighing items are the power supply and the main large motor. Initially, we promised to use three 12V car batteries as a power supply, but we found a better alternative which consists of smaller 12V SLA batteries (discussed in the background section) that added 12 KG to the system (4 KG each). The main motor weighs at 3.2 KG bringing the total weight of the product to around 15.2 KG.

After thorough testing, the bike with our mounted kit (product) was easily able to carry a person that weighs 86 KG.

Financial and Quality Constraints

Due to the rising exchange rates of the Lebanese Pound, we found it challenging to keep up with additional costs needed to cover all the components of the project, as well as all the contingencies we faced throughout the development process. We initially aimed at keeping all costs under 300 USD, in the hopes of selling the product for a reasonable profit, but after the economic crisis, the components started skyrocketing.

This also affected our choice of motors, electronics and even the batteries.

Time and Scope Constraints

Initially, we wanted to create a pleasant dashboard comprised of high-quality potentiometers with numerical and semantic markings to make it easier for the user to change speeds and operation modes, but due to time limitations, as well as other factors, we decided to make a simple 3D printed dashboard and speedometer setup.

Time also affected an important feature we wanted to include but had to abstain from adding, which is a break detection system. This detection could have added the ability to enable an additional breaking technique using the 450W motor, as well as to disengage the cruise control system whenever the breaks are squeezed.

Standards Utilized

Battery

The battery is further covered in the background section of the report. As for the standards we must abide:

- Lead-Acid battery safe use, code of conduct - Reference: BS 6133:1995
- General information - Reference: IEC 60086-2, BS

Background

In this section, we will discuss each externally manufactured product we used in our project, and the reason we used each compared to other similar products.

MY1018 Brushed DC Motor [3]

The choice of the motor was not an easy one. We had many options, including a very compact Chinese model that encompasses the motor inside the wheel rim making it very easy to install, and more symmetrical in terms of design. We decided to go with the MY1018 Brushed motor from AliExpress since the other model had a shipping cost of over 1000 USD for some reason:

Figure 2 BDC Motor [3]

This motor comes in two configurations as illustrated by the table below:

motor data

Specification	MY1018	
Rated output Power	450W	450W
Rated Voltage	24V DC	36V DC
Rated speed	3000 RPM	3000 RPM
No load speed	4000 RPM	4000 RPM
Full load Current	≤24.70A	≤16.50A
No load Current	≤2.5A	≤2.2A
Rated Torque	1.43 N.m	1.43 N.m
Efficiency	≥78%	≥78%
Gear Ratio	7.18:1	
Application	Light E.V./ E-bike	

Figure 3 BDC Specs [3]

Both options generate an output power of 450W. It is rated for speeds of 3000 RPM. The torque generated is 1.43 N.m with a gear ratio of 7.18:1, which means that the torque on the rear wheel will be 7.18 times that of the motor shaft. The difference between both configurations is that the 36V motor will require less current and therefore a smaller battery (less capacity needed), which also means a cheaper power supply. The lower current consuming motor appealed more to us since the price difference is significantly negligeable, certainly compared to the money saved by using a smaller capacity power supply.

A3144 Hall-Effect Sensor [4]

The Hall-Effect sensor is a small electronics component that changes its output based on external magnetic fields exerted on it. We needed it to detect the exact time the wheel (or the pedal) completed a full rotation. At first, we wanted to use a cheaper option, the HAL301, but after testing thoroughly, we found the analog fluctuations to be very inconsistent and to fix this would mean building a large circuit to convert it into the desired square wave signal. The A3144 Hall-Effect's voltage output will result in a square wave since the voltage will vary every time the rotating object completes a full revolution due to the variation of the magnetic field and the internal comparator. By finding the period to complete 1 revolution, the frequency which is the reciprocal of this period will be the angular velocity.

Knowing the time needed for one rotation we can calculate the number of rotations per minute. Then we get the distance travelled in a minute (speed) by multiplying the rotations per minute with the diameter of the bicycle.

Time taken to rotate once = T
RPM =1/T
Distance travelled in a minute = RPM * D
Speed = distance / time = RPM * D
Where T = time in minutes and D = Diameter of the wheel

450W High Power PWM Motor Controller [7]

In the research stage of the project, we attempted to build our own circuit for PWM control over the main 450W motor. We tried using Power MOSFETS and an IR2110 MOSFET driver, to convert the 5V PWM signal from the microcontroller to a 36V PWM signal. This technique proved to be significantly unreliable due to the components needing to be of very close proximity to eliminate the inductive hum and overheating of the MOSFETS (even with dead-time control from either a low-pass filter or directly from the microcontroller).

We decided to buy a manufactured circuit from AliExpress that solved all the problems we had with the previous circuit. The only model that shipped to Lebanon is the H-Bridge controller shown below:

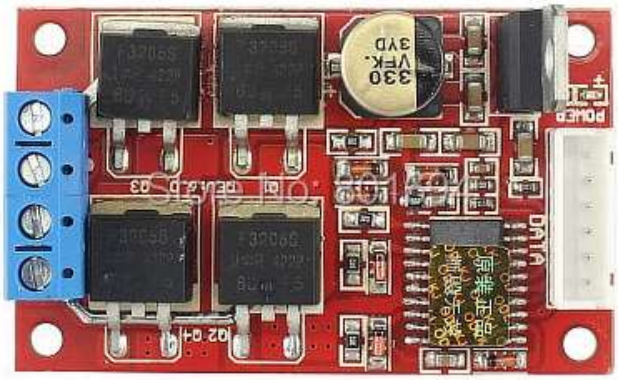


Figure 4 450W PWM Motor Controller [7]

DC-DC Converter [8]

We needed a way to convert the 36V coming from the power supply into 12V that can be used to power the stepper motors and the microcontroller after converting it into 5V using a voltage regulator. After much research, we found that the best option was to purchase it locally since the costs were accumulating at this stage of the development and we were limited in that regard. We opted for the “5A DC/DC Step Down Module with Voltmeter” from Katranji:

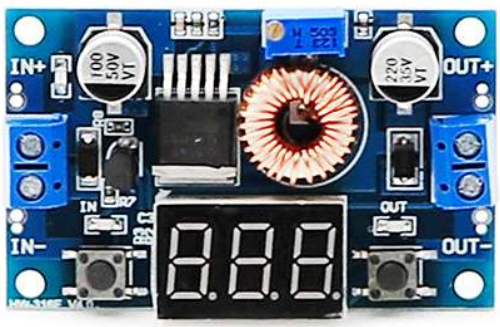


Figure 5 DC-DC Converter [8]

28-BYJ Stepper Motor [2]

We needed a small stepper motor to display the speed, this motor will be used as a speedometer by changing the position of a small arrow. Since we had this motor in our inventory, we decided to go with it, but we could have chosen any stepper motor of similar size:

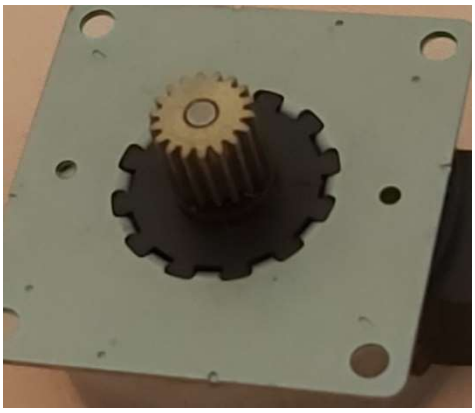


Figure 6 28-BYJ Stepper Motor

L298N Stepper Motor Controller [10]

Along with the 28-BYJ stepper motor, we needed a controller to convert the 4-pin 5V signal into a 12V signal that the motor can use. We could have used MOSFETS to do the conversion, but due to the availability of the L298N driver in our inventory, we decided to use it:

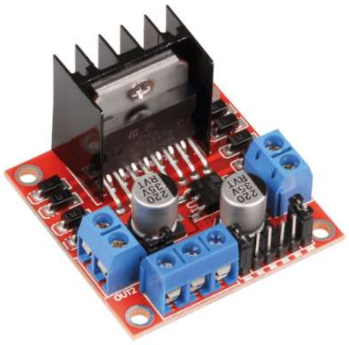


Figure 7 L298N Stepper Motor Driver

17HD48002-22B Stepper Motor [1]

We needed a stepper motor with high torque to accommodate the gear shifting technique we went with. This motor can provide 560mN.m at 1.8◦ step resolution which is more than enough:

Figure 8 17HD48002-22B Stepper Motor

A4988 Stepper Motor Driver [5]

Along with the gear shifting stepper motor, we needed an accurate motor driver. We found one locally that gives high precision, meaning it works with our 1.8◦ step stepper motor. The choice of the controller was given to us as advice from the retailer we purchased the 17HD48002-22B Stepper motor from. After researching the datasheets of each, we found it to be compatible:

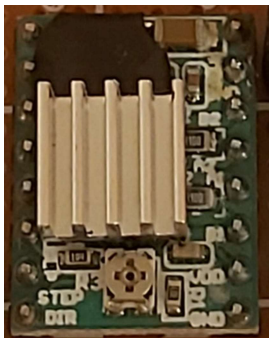


Figure 9 A4988 Stepper Motor Driver

L7812CV 12V-5V Step-Down Voltage Regulator [6]

After converting the power source's voltage (36V) to 12V using the DC-DC converter we purchased from Katranji [8], we still needed to bring it down to 5V in order to power the microcontroller, thus we opted for an L7812CV voltage regulator that was available in our inventory (we could have used any 12V to 5V voltage regulator).

PIC18F4550 Microcontroller [11]

Since we knew most about the PIC18F4550, and we already had one available from the Embedded Systems Course, we decided to go with it.

TNE12-15 12V Sealed Lead Acid Batteries (SLA) [12]

Preferably, we would have used a 36V Lithium-Ion battery instead of three 12V SLA batteries. Initially, we wanted to use car batteries instead of SLA batteries, but since we found a retailer locally, we were able to purchase them at a reasonable price (scooter shop in Beirut); thus, we decided to go with this option. These batteries have a capacity of 15Ah, which would still allow over an hour of continuous full power usage of the motor (around 1.2 hours):

Figure 10 TNE12-15 SLA Battery [12]

Miscellaneous

We also needed a way to allow the user to choose motor and gear values that the microcontroller can later use. We decided to implement an interface panel composed of two potentiometers for each user variable value input (speed/target cruise speed, and gear). We also included an ON-OFF-ON switch to allow the user to choose the mode of operation of the system (ON(Assist)-OFF(Manual)-ON(Cruise)).

As for most of the rest of these electronics, we acquired them from the Engineering Labs:

- 10K Potentiometers
- Switch
- Breadboard wiring
- PICKit 3

We also purchased a 10k slider (linear) potentiometer from a local electronics store, to be used in the gear shifting detection system we built (our choice of this potentiometer is further discussed in the gear detection section).

Main Motor Overview

In this section, we will discuss the functionality of our main motor, as well as the setup that will be used to drive the rear wheel using the motor. The actual implementation will be discussed in the next section (main motor attachment section).

The sprocket on the shaft of the motor is connected to a larger sprocket placed on the rear hub of the wheel by using the chain that comes with the kit. Looking into the motor specifications, we were able to deduce the torques and speed generated by the motor on the rear wheel. The rated torque represents the torque of the motor at rated output power and speed. Thus, the torque generated on the sprocket fixed on the rear wheel can be calculated as follows:

$$T_r = T_m \, x \, \frac{r_r}{r_m}$$

T_r: torque on the rear wheel

T_m: torque generated by the motor

r_m: radius of the sprocket mount to the motor shaft

r_r: radius of the rear sprocket

This gives us the torque generated on the rear wheel at the motor's rated conditions, which is when the speed of the motor is 3000 RPM. To get the torques generated at lower speeds, we simply divide the output power (in Watts) by the speed (in rad/s), then obtain the corresponding torque on the rear wheel by multiplying the motor torque by 7.18 (gear ratio). This phenomenon is illustrated in the below graph:

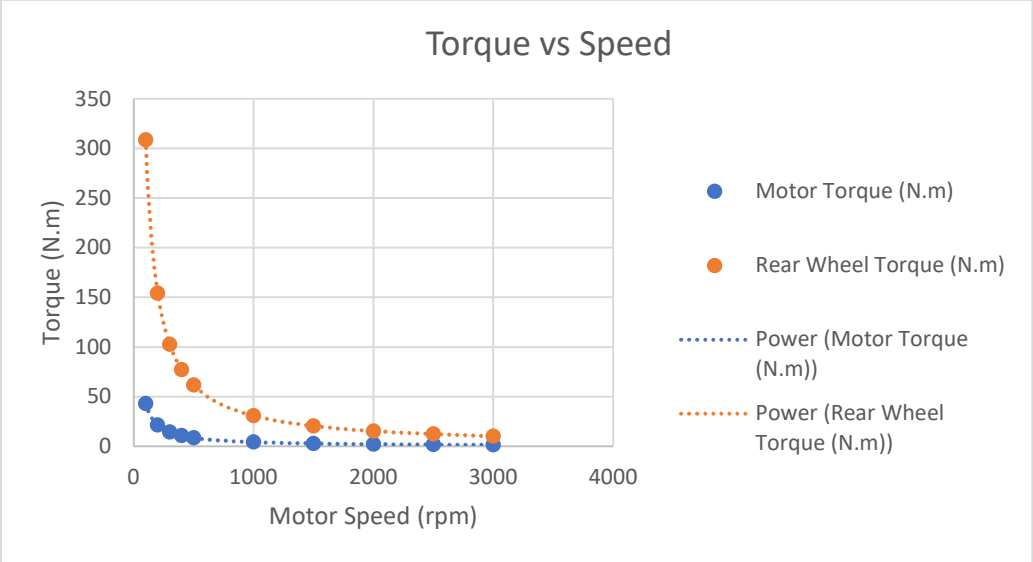


Figure 11 Torque vs Speed Graph

Main Motor Attachment

The motor came with an instalment kit that had a few problems. The motor's attached free-wheel gear was mis-aligned with its counterpart on the rear wheel's rim. Another problem we faced was that the chain was touching the bicycle's chassis, due to the unusually large chassis radius that ours had. Both these problems were causing the chain to derail from the rim's gear:

Figure 12 Motor's Chain touching the chassis

As we can see in the picture above, the chain is very slightly bent at the point where it meets the chassis. To resolve that problem, we grinded a small portion of the chassis' tube and freed the chain. We also filled in the gap with epoxy glue:

Figure 13 Chassis cut and infill

Figure 14 Motor's chain freed from the chassis

We still had to find a solution to align the rim's gear and the motor's gear. We decided to 3D print a cylinder with screw holes with TPU, which is a flexible material, to place behind the rim's gear. This solved the problem perfectly:

Figure 15 Rim's gear adjustment 3D Print

Figure 16 Rim's gear adjustment

Another problem we encountered was the motor's chain length. The chain that came in the kit was slightly longer than desirable, it also contributed to the derailing when the alignment of the gears was not fixed (as discussed previously). We solved this problem by using a chain-breaker we purchased from a local bicycle store.

Gear Shifting

The original gear shifting control of the bike was based on a cable, the user would pull and release the cable through a handle in order to change gears. To transform it to an electronic mechanism we decided to remove the cable and attach a stepper motor in addition to a L-shaped metal handle which would move the rear derailleur to change the position of the chain to the appropriate gear.

Figure 17 Original Gear Shifting Mechanism

In the figure above we can observe the original mechanism (cable).

Shifting from one gear to another was possible through the rotation of the motor, which was converted into a translational motion of the handle through the rotation of the 8mm bolt that was connected to the shaft of the motor using a metal coupler. The handle is shaped as an “L”, locally manufactured (local metal specialist) and threaded to slide into the 8 mm bolt. Then we fixed the L-shape handle on the bolt where the cable was originally fixed.

Old Design

At first, we decided to go with a mount which will put the shaft of the motor in a perpendicular direction with respect to the gears. After printing the mount, installing it and testing it, we deduced two main problems:

- First, the design of the mount was not compatible with our derailleur, as it was blocking the way and could not move the lower part of the derailleur, making it impossible to move the chain.
- Second, the direction and angle used to fix the shaft of the stepper motor ended up producing an undesired tension on the bolt which made the motion of the handle harder and impossible to move the derailleur sometimes.

Figure 18 First design of the electrical gear shifting mechanism

New design

After deducing the main problems, we redesigned a new stepper motor's mount. An angle of 14◦ was added to the motor mount in order to align the motor shaft with the derailleur exactly. We also allowed some room where the mount attaches to the chassis.

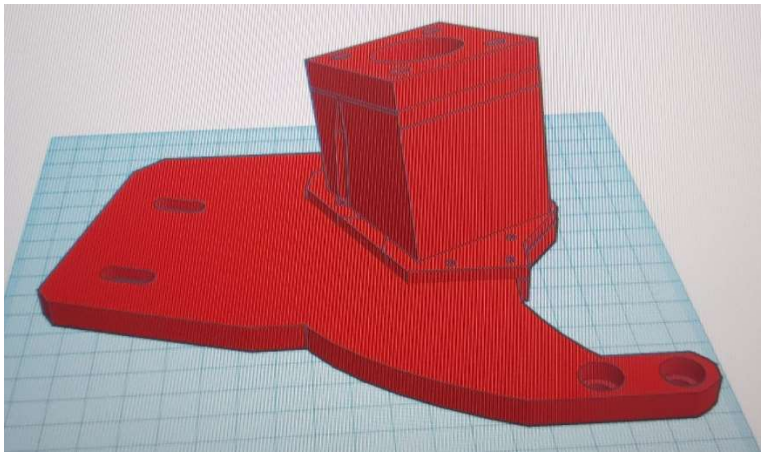


Figure 20 Second design of the electrical gear shifting mechanism using TinkerCad (Part1)

Figure 19 Second design of the electrical gear shifting mechanism using TinkerCad (Part2)

Figure 21 Second design of the electrical gear shifting mechanism (mounted)

Gear Detection

Old Design

First, we decided to use an ultrasonic sensor to sense the position of the handle, which we concluded from the position of the derailleur and finally the position of the chain. After installing it and observing it, the value given by the ultrasonic where sometimes unreliable.

In the following figure we can observe the location of the ultrasonic sensor on the stepper motor mount:

Figure 22 Ultrasonic mount on the old design

New Design

After removing the ultrasonic sensor out of the picture, we decided to go with a slider potentiometer. The slider fixed to the moving L-shaped metal handle will change the resistive value of the potentiometer, taking these values and saving them into our system we can locate the position of the derailleur. After printing the mount to connect it to the stepper mount and testing it, we found out that there is a forward/backward offset, i.e., the same gear could have multiple positions based on the motor direction of rotation, and the forces exerted on the system at certain points, which made the sensing unreliable and inaccurate. We fixed this problem by attaching the L-shaped metal attachment to the bottom of the derailleur, making the offset constant:

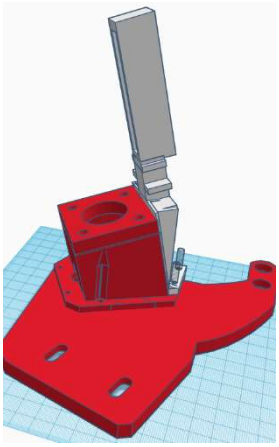


Figure 23 Gear shifting mount with potentiometer mount 3D model using TinkerCad

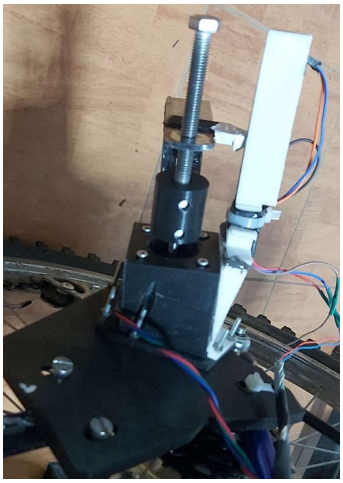


Figure 24 Gear Shifting mount with potentiometer mount (mounted) (Part 1)

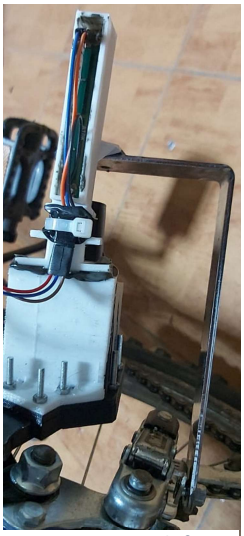


Figure 25 Gear Shifting mount with potentiometer mount (mounted) (Part 2)

Dashboard

We needed a way to represent two potentiometers, a ON-OFF-ON switch, and the speedometer system that will reveal the current speed. To make the speedometer, we 3D printed a case which would hold the 28-BYJ stepper motor which had a gear (18 teeth, 1cm diameter) attached to its shaft.

Since we are using the stepper motor to show the current speed, then we assumed we had less than 180◦ to show the whole spectrum, and since the motor needs 12 steps to make a full rotation, then we needed to find a way to increase the resolution of the readings. We decided to add to the case, a 3D printed gear four times bigger than the motor's (72 teeth, 4cm diameter). This allowed us to show the full spectrum using 45◦ with a resolution of 1KM/H. We also included AA-batteries' spring metal contacts on the case and on the arrow at 0KM/H to be used when calibrating. Finally, we 3D printed an attachment out of TPU (flexible material) to mount the speedometer:

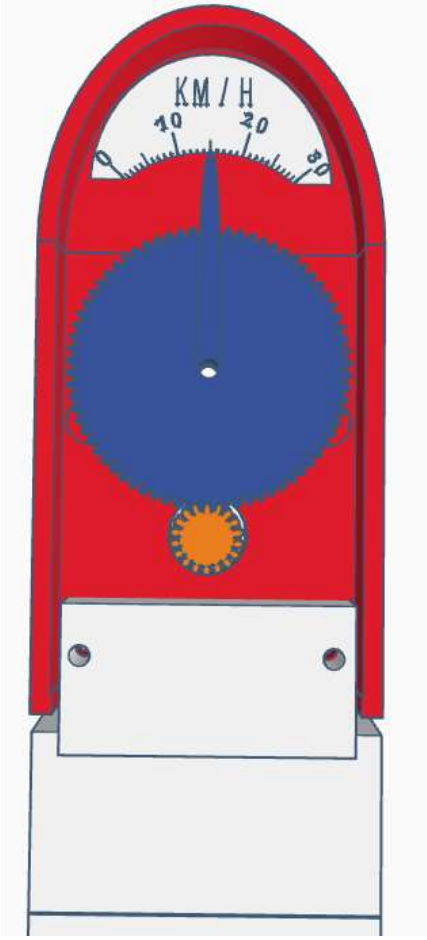


Figure 26 3D model of the speedometer system with TinkerCad

Figure 28 3D printed and assembled speedometer (closed)

Figure 27 3D printed and assembled speedometer (opened)

The same technique was used to mount the potentiometers and the switch in a 3D printed case:

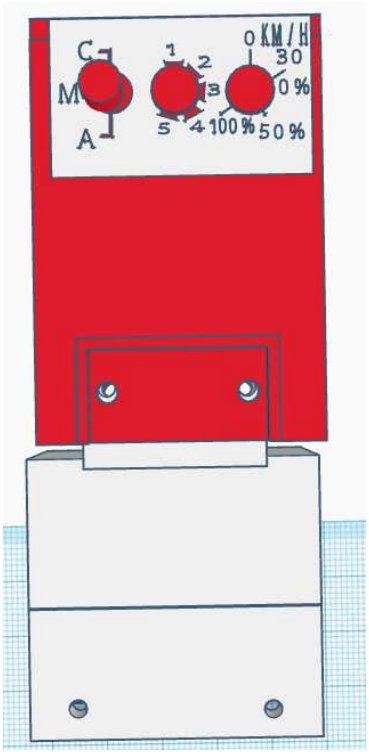


Figure 30 Potentiometers and switch mount 3D model using TinkerCad

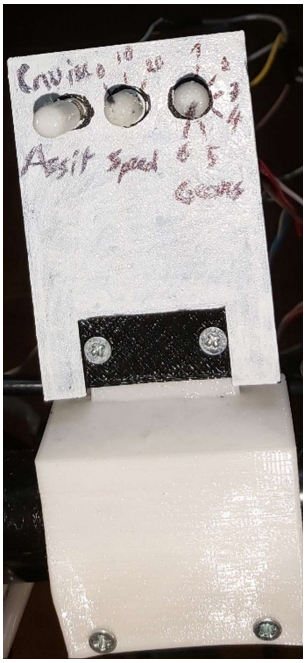


Figure 29 Potentiometers and switch mount (mounted)

Electronics Preparations

After building the circuits and testing them, we needed to transfer them to perforated boards in order to make it more compact and presentable than a regular breadboard, especially since the system will be shaking on the road and a breadboard's connections are not always reliable. We decided to include in one board the PIC18F4550, and the voltage regulation portion (both discussed in the background section), and a separate board for the stepper motor driver circuit (A4988 from the background section):

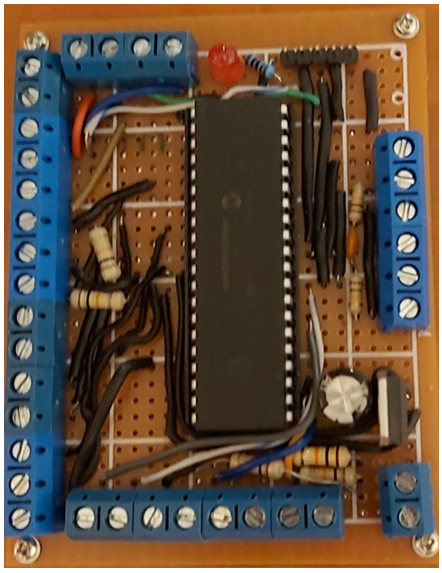


Figure 32 Main Perforated Board (Front)

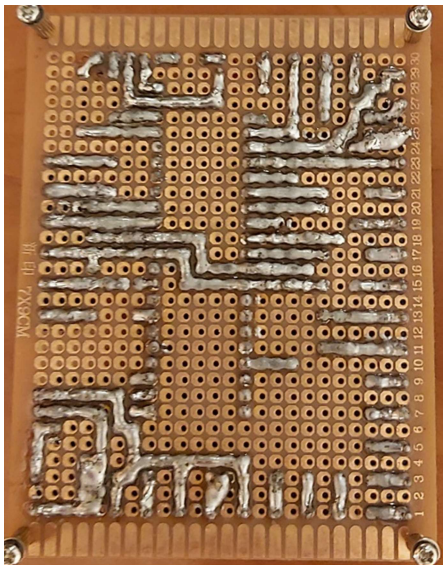


Figure 31 Main Perforated Board (Back)

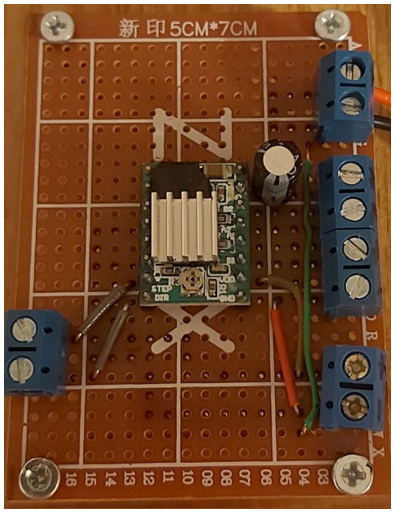


Figure 34 Stepper Motor Perforated Board (Front)

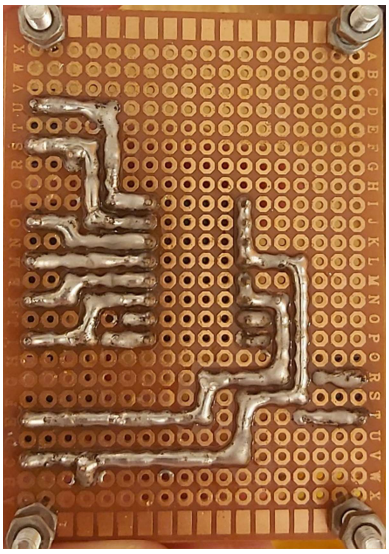


Figure 33 Stepper Motor Perforated Board (Back)

Next, we 3D printed in TPU (flexible material) two mounts for each Hall-Effect sensor (discussed in the background section) on the bike's chassis. We also glued small neodymium magnets less than a centimetre in front of each:

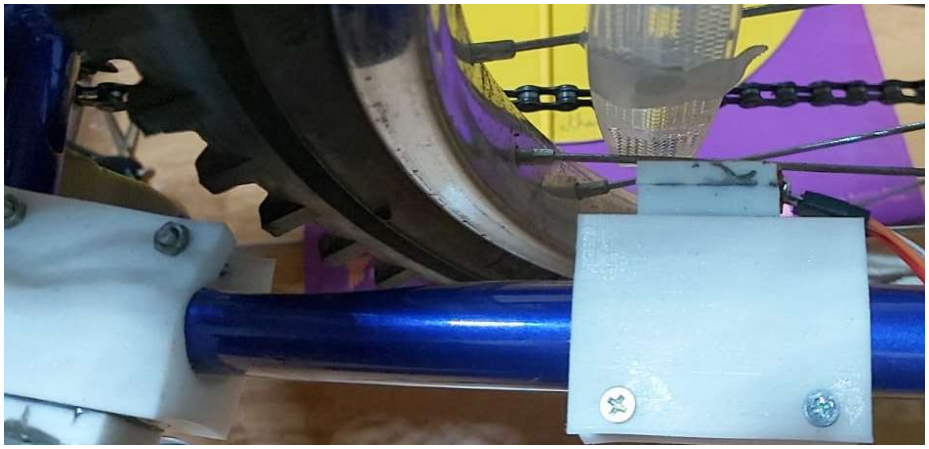


Figure 35 Hall-Effect sensors mounted (Left for cadence, right for system speed)

Figure 36 Hall-Effect sensor front view (mounted)

Batteries and Electronics Placement

The best way to store the batteries is at the back using a cargo rack. We also wanted to include all the electronics to keep the whole system compact. We purchased locally a rear bike rack from a bicycle store, and modified (by a local metal specialist) it to be able to slide in the whole package later:

Figure 38 Cargo rack unmodified

Figure 37 Cargo rack modified

Next, we needed a case that would hold everything in it, thus we made a wooden box with the same sliding mechanism that we used on the rack above. This mechanism allows us to add a screw at the end to secure it in place:

Figure 40 Wooden box for batteries and electronics (mounted)

Figure 39 Rack sliding mechanism

Figure 41 Sliding mechanism (unmounted)

We also needed to prevent shaking and breaking any of the batteries or the electronics, thus we 3D printed in TPU (flexible material) a case for every battery, and a mounting surface for every PCB. We also printed a bottom for the PCBs to screw in, therefore giving us the ability to remove them whenever we wanted without unscrewing anything (press-fit the screwed part):

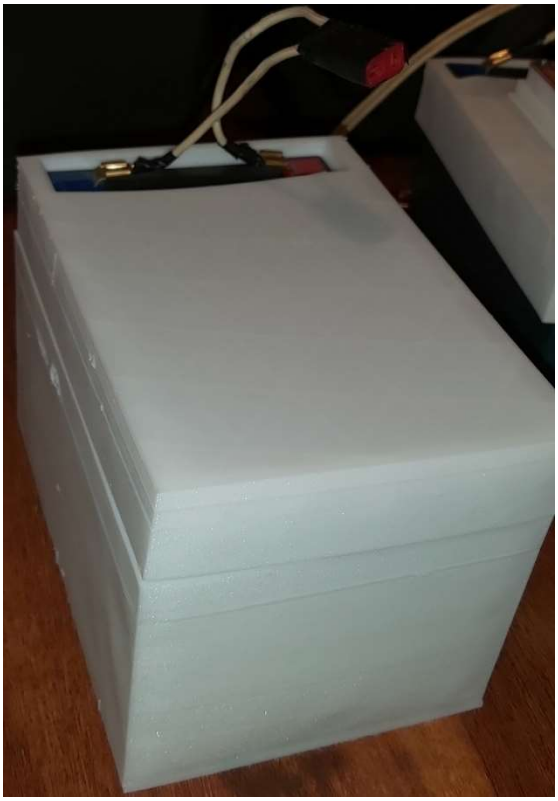


Figure 43 Battery in its case

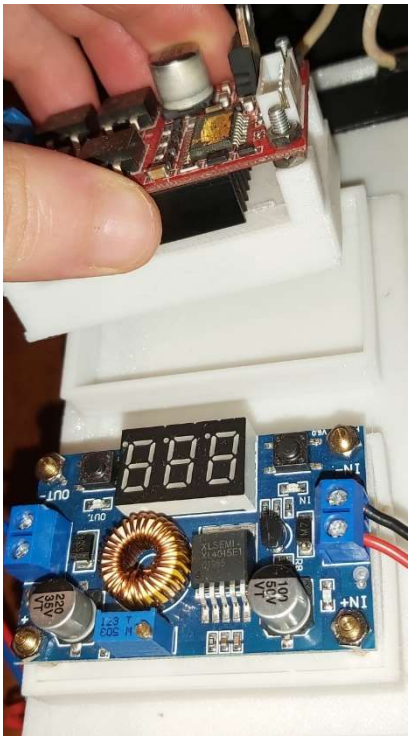


Figure 42 PCB mounting system

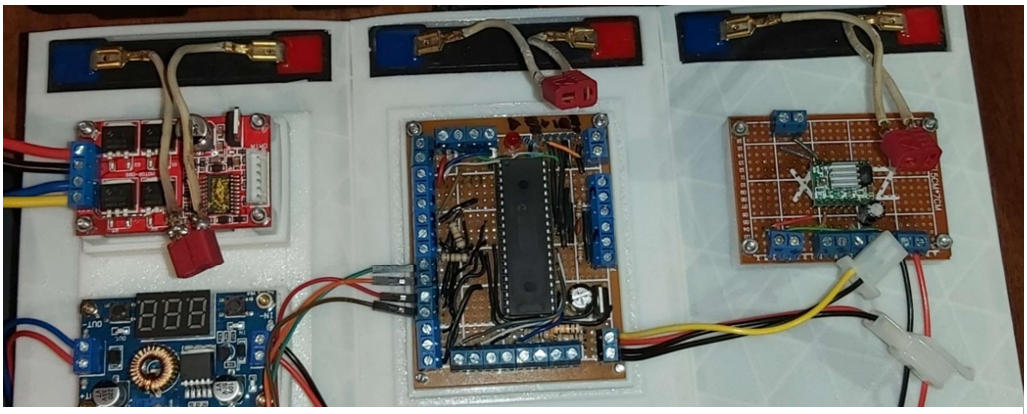


Figure 44 Batteries and electronics in their cases (missing from the picture is the L298N motor driver for the speedometer)

Wiring and Routing

In order to link the dashboard, gear shifting and detection, Hall-Effect sensors discussed in the background section, and the main motor we needed to route some wires around the chassis of the bike. In our case, we only needed around AWG 22 wire for the electronics, thus we used Cat5 Ethernet cables to solve this problem. We needed two (8 wires each) long from the dashboard (keeping in mind to leave a loose portion to allow rotation when steering), one wire from the sensors side, and one from the gear shifting side. We then attached them to the chassis using zip ties, and soldered them and wrapped them in heat shrink:

Figure 45 Wiring attached and routed (Red arrow pointing to the loose section needed for steering)

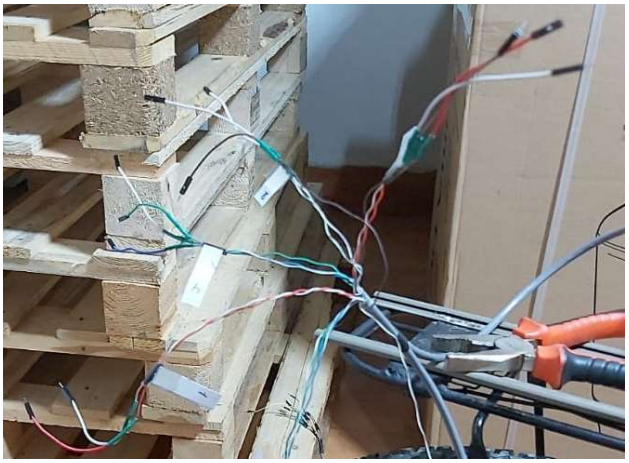


Figure 47 Wire preparation before connecting, tucking and organising together (back-side)

Figure 46 Wire preparation before connecting, tucking and organising together (front-side)

PID Controller

In order to add a PID block to our system, initially we need to model the system accurately as is. After giving it some thought, we considered the system as a low-pass filter for throttle variations of the main motor (PWM duty-cycle). This can represent our system since it takes a significant amount of time (2/f -> half a period) to achieve a speed because of the physical limitations of the motor, friction and drag. We can consider a frequency f_0 which represents the cut-off frequency of the throttling, i.e., below that frequency the speed follows the input (PWM) variation, and above, the speed follows f_0.

We can now write the transfer function of the E-Bike system we made: $TF = \frac{1}{1+\frac{s}{w0}} = \frac{w0}{s+w0}$

We can calculate f_0 by performing some physical testing on the bike using the 450W motor:

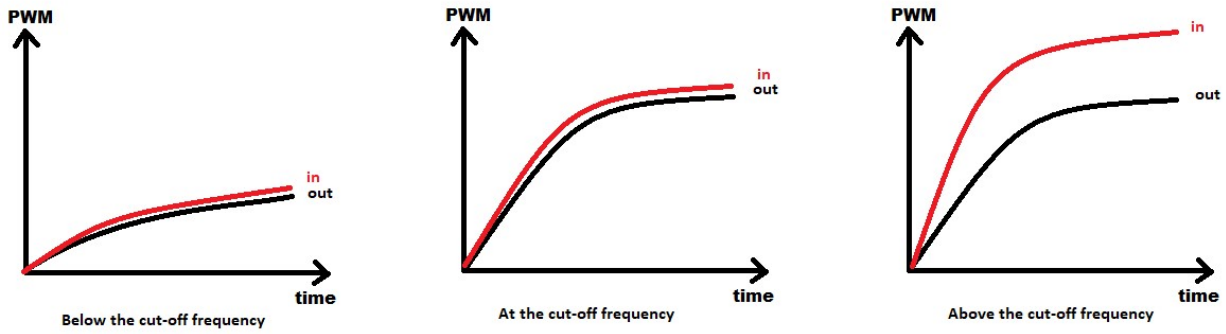


Figure 48 Relation between throttle and speed drawn using MS Paint

First, we calculate the rise time of the speed T_1, by applying a step input and finding the time to reach 63.3% of the steady state speed:

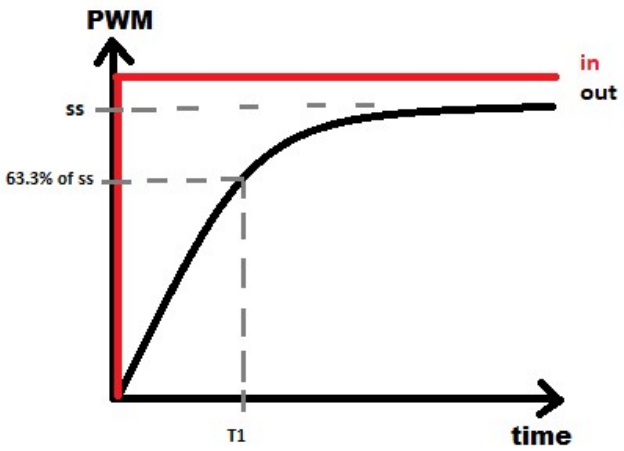


Figure 49 Visualization of the process for finding T1 using MS Paint

Next, we calculate the fall time of the speed T_2, by releasing throttle after reaching steady state and finding the time to reach 36.67% of the steady state speed:

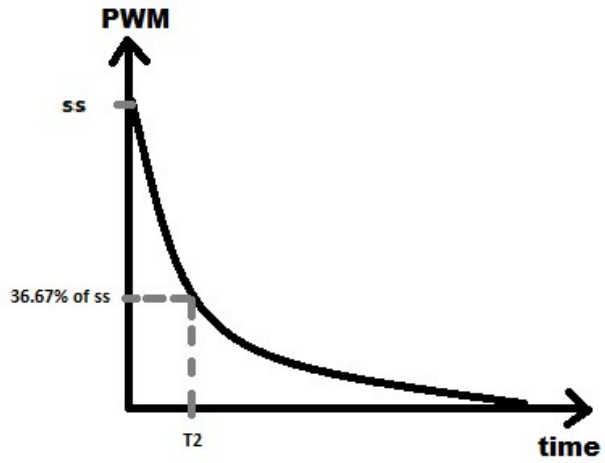


Figure 50 Visualization of the process for finding T2 using MS Paint

Now, we can say that $T_0 = T_1 + T_2$ with $T_1 = 18$ s and $T_2 = 23$ s, thus $T_0 = 41$ s.

From T_0 we can find $f_0 = 0.024$ Hz, and $w_0 = 0.151$ rad/s.

Next, we can start to develop our PID controller. We are aiming at a unit step response with not steady state error, and overshoot below 5%. All other properties (rise time, settling time, etc) do not affect the system significantly and can be excluded when finding our constants:

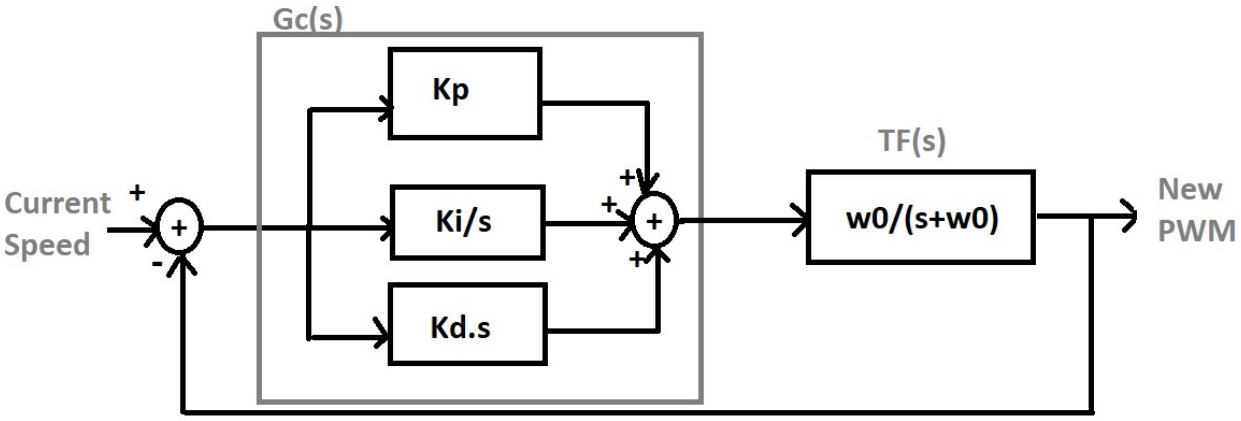


Figure 51 System Block Diagram after PID controller added

Where: $Gc(s) = \frac{Kd}{s}\ (s^2 + \frac{Kp}{Kd}\ s + \frac{Ki}{Kd})$

The three constants can be simplified into one, since we are using a specific unified sample period:

$Kp = K$ $\quad Ki = Kd = \frac{K}{\tau i}$ where $\tau\ i = 2.5\ s/repeat$ (chosen after trial and error in the software)

Then, $Gc(s) = \frac{K}{2.5}\left(\frac{s^2+2.5\ s+1}{s}\right)$

The open-loop transfer function becomes: $OLTF(s) = (\frac{w0}{2.5})\ (\frac{(s+0.5)(s+2)}{s\ (s+w0)})$

In the above equation, K is considered to be a separate block in series with the OLTF.

We can see that we have s^1 in the denominator, meaning that our system is of type 1, i.e., unit step response will not have any steady state error. We should also note that we will not be able to do better in terms of velocity error using this system configuration, but that is not a problem in our case:

$$Kv = \lim_{s\to 0}(s.OLTF) = \frac{K}{2.5}$$

We solved the problem of the steady state error, now we should choose a value of K to reduce overshoot. To do so we drew the root locus of our system using MATLAB:

We can write our characteristic equation that will be used in MATLAB to draw the root locus:

➔ $K\ .\ OLTF(s)\ .\ 1 = Kn\ \frac{(s\ +\ 0.5)(s\ +\ 2)}{s(s+0.151)}$ ➔ Let $\mathrm{Kn}\ =\ 0.0604\ .\ \mathrm{K}$

```
>> % Building the TF Object
>> s = tf('s')

s =

  s

Continuous-time transfer function.

>> s = tf((((s+0.5) * (s+2)) / (s * (s+0.151))))

s =

  s^2 + 2.5 s + 1
  ---------------
   s^2 + 0.151 s

Continuous-time transfer function.

>> rlocus(s)
```

Figure 52 MATLAB root locus code preparation

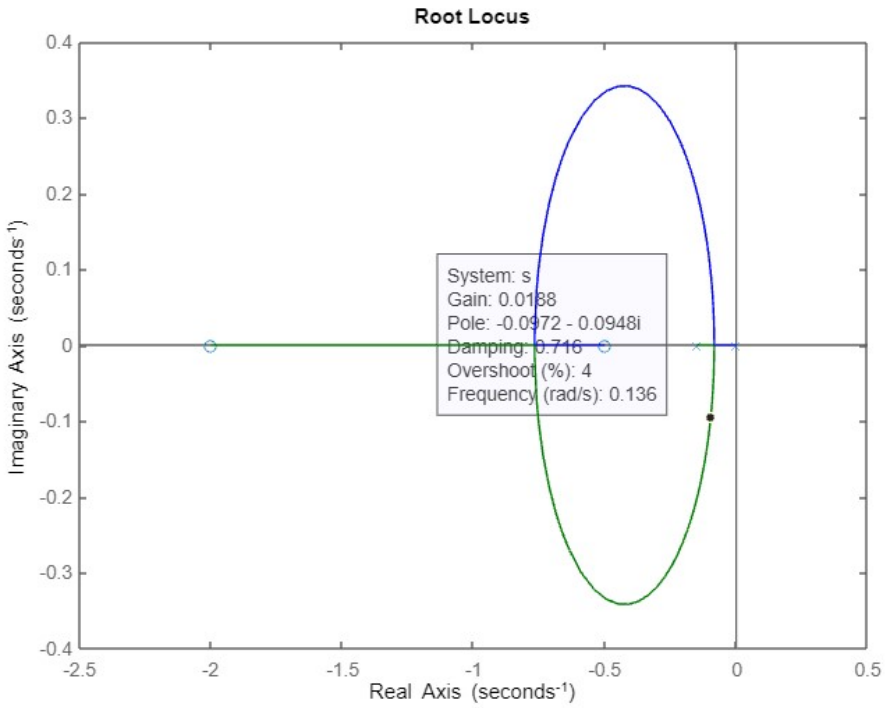


Figure 53 Root Locus and K value using poles on MATLAB

We tried multiple values of K and we found a good value shown in the Root Locus plot above. With Kn = 0.0188, we get 4% overshoot which is acceptable.

$$K = \frac{Kn}{0.0604} = 0.311$$

After using this value of K, the system was able to achieve the speeds we wanted in a reasonable time frame (depending on the target speed but usually rise time around 21 seconds, and settling time around 76 seconds), without any overshoot (or negligeable to the point we did not detect it) or steady state error.

Software

Since we are using Microchip's PIC18F4550, we decided to use MPLab as an IDE for the project. Initially, we had to define the microcontroller's configurations (Appendix A). The only important configuration here is the internal clock of 8MHz.

Next, we need to initialize our software by assigning all registers with the right values for timers, PWM, pin declarations, interrupts, etc. (Appendix B). In the initialization stage, we also need to calibrate our speedometer, since the arrow might not be pointing at 0 initially making all later arrow position assignments wrong. The "calibrate_speedometer_stepper" method operates as described in the pseudo-code below:

```
calibrate_speedometer_stepper(limit) {
    while limit is not zero or below {
        if zero position reached {
            Move the arrow away from zero position (Appendix D)
            Move the arrow towards the zero position (Appendix D)
            // This is done to make sure we are at zero position
            return
        }
        Move the arrow towards zero position 1-step (Appendix D)
        limit ← limit - 1
    }
    Mark Speedometer as out of service
}
```

Next, we need to define the "update_gear_stepper" and "calculate_automated_gear" methods that can be described following the below pseudo-codes:

```
update_gear_stepper(target_gear) {
        if pedalling {
                Read current gear position (Appendix G)
                If target_gear > current_gear {
                        Lower gear position by stepping backwards (Appendix E)
                } else if target_gear < current_gear {
                        Raise gear position by stepping forward (Appendix E)
                }
        }
}

calculate_automated_gear() {
        // Constant Thresholds can be found in Appendix C
        if (current speed is between gear 1 and 2 thresholds OR below gear 1 ‘s threshold) AND (pedalling is above its threshold) {
                return gear 1
        } else if (current speed is between gear 2 and 3 thresholds) AND (pedalling is above its threshold) {
                return gear 2
        } else if (current speed is between gear 3 and 4 thresholds) AND (pedalling is above its threshold) {
                return gear 3
        } else if (current speed is between gear 4 and 5 thresholds) AND (pedalling is above its threshold) {
                return gear 4
        } else if (current speed is between gear 5 and 6 thresholds) AND (pedalling is above its threshold) {
                return gear 5
        } else if (current speed is above  gear 6 threshold) AND (pedalling is above its threshold) {
                return gear 6
```

```
        }
        // These constants were chosen after extensive trial and error
}
```

Next, we will introduce the “update_speedometer_stepper” method, which can be described in the pseudo-code below:

```
update_speedometer_stepper() {
        if speedometer is out of service { return }
        let delta ← current speed – old speedometer position  // same resolution of 1KM/H for 1 step
        if delta > 0 {
                Move the arrow away from zero position 1-step (Appendix D)
        } else if delta < 0 {
                Move the arrow towards the zero position 1-step (Appendix D)
        }
        Save current speedometer position
}
```

Next, we need an infinite loop that is used to always check the user's choice of operation mode and to update motor speed and gear position accordingly. In this loop we will also check, and update the speedometer based on the current speed at any given time. The below pseudo-code describes the infinite loop:

```
Always Loop {
    If in assistive mode {
        If pedalling {
            Read user's input on motor speed (Appendix G)
            Set PWM based on user's input (Appendix F)
            update_gear_stepper(calculate_automated_gear())
        }
    } else if in cruise control mode {
        Read user's input on gear position (Appendix G)
        update_gear_stepper(user's input)
        // PID is calculated and implemented in the interrupt subroutine
    } else (manual mode) {
        Read user's input on gear position (Appendix G)
        update_gear_stepper(user's input)
        Read user's input on motor speed (Appendix G)
        Set PWM based on user's input (Appendix F)
    }
    update_speedometer_stepper()
}
```

We should also define the methods for speed and PID calculations that can be described in the below pseudo-codes:

```
process_speed_sample() {
        revolutions_per_hour ← RPH_CONST / time from last revolution (Appendix C for RPH_CONST)
        current speed ← revolutions_per_hour * circumference (Appendix C)

        revolution_time__in_seconds ← time from last revolution / RPS_CONST (Appendix C)
        append revolution_time_seconds to a global variable holding the total_time_elapsed

        if total_time_elapsed >= sampling period (Appendix C as “samples_timeout_in_s”) {
                process_and_update_PID()  // described below
        }
        time from last revolution ← 0
}
process_and_update_PID()  {
        Read user’s target_speed input (Appendix G)
        If target_speed has changed { reset cumulative integrator of the PID }
         // K can be found in Appendix C which has a wrong value that has been update in the
        // implementation to 0.311

        // proportional of the error
        Let pid_p ← K * (target_speed – current_speed)
        // sample_period in Appendix C as “samples_timeout_in_s”
        // derivative of the error
        Let pid_d ← (K/sample_period) * (target_speed – current_speed)
        // integration of the error
        Let pid_i ← pid_i + ((K/sample_period) * (target_speed – current_speed))
        Let pid ← pid_p + pid_d + pid_i
        Set PWM based on “pid” value above (Appendix F)
}
```

Finally, the below pseudo-code describes the interrupt subroutine:

```
isr() {
    if wheel sensor triggered {
        Read Timer 3's value and set as the time for one revolution
        Reset Timer 3 and start it
        process_speed_sample()          // discussed above
    } else if Timer 3's overflow interrupt triggered {
        Reset Timer 3 and stop it
        Set time for one revolution to Null
    } else if pedal sensor triggered {
        // Timer 0 was used for better resolution with slow pedalling
        Read Timer 0's value and set as the time for one revolution
        Reset Timer 0 and start it
        Let cadence ← RPM_CONST / time for one revolution  (Appendix C for RPM_CONST)
    } else if Timer 0's overflow interrupt triggered {
        Reset Timer 0 and stop it
        Let cadence ← 0
    }
}
```

MPLab can also show us the memory used following this code, in terms of Program (code itself) and Data memory (variables and constants) in Bytes:

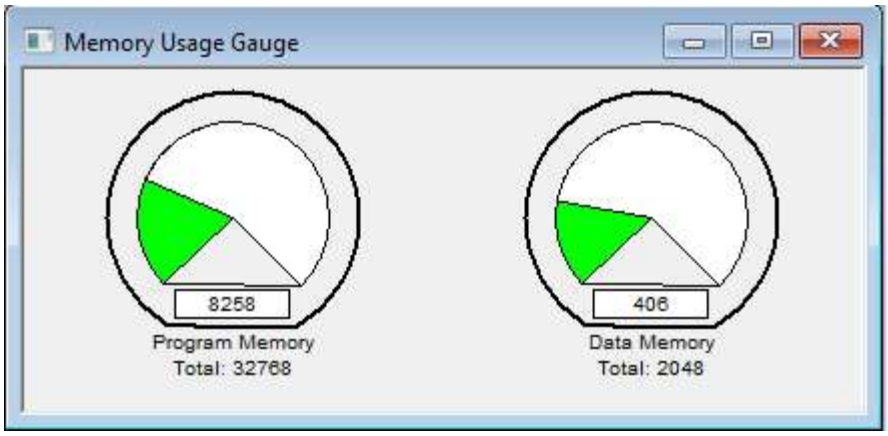


Figure 54 Program and Data memory consumption (in Bytes)

Results

After completing all the parts of the project described in this report, we ended up with a regular bicycle, modified to have an electrical propulsion system with automated gear shifting. The system operates in three modes:

- Manual Mode:
  - Motor PWM based on potentiometer input.
  - Gear position based on potentiometer input.
- Assistive Mode:
  - Motor PWM based on speed and cadence.
  - Gear position based on speed and cadence.
- Cruise Control Mode:
  - Motor PWM based on PID calculation and user input of target speed using a potentiometer.
  - Gear position based on potentiometer input.

We also have two sensors that detect wheel and pedal revolutions, as well as a functional dashboard comprised of an analog speedometer, two potentiometers (speed, target speed, and gear selections – depending on the operation mode), and an ON-OFF-ON switch to choose the mode of operation.

The assistive mode's response was not discussed earlier since the motor's output followed the input exactly considering we are switching its throttle under the cut-off frequency we found and thoroughly discussed in the PID section. As for the cruise control mode's response, we were able to achieve an average rise time of 21 seconds, and a settling time that is always below 76 seconds which only affected the initial system (no PID control) by less than 16 % increase on average in response time (considering the same settling time for both systems – with or without PID control).

We also did not show or discuss the steady state value of the system (maximum speed possible if full throttle is applied – constant speed achieved) since it will vary from bicycle to another, not to mention the surfaces (concrete sidewalk, asphalt road, rocky or muddy path, etc.). This value will vary, which will cause the rise time to vary. We found our steady state constant speed to be 22 KM/H when tested on a regular (not very maintained) asphalt road, and any assumptions and calculations using the rise time can be considered approximations.

Regarding the gear shifting mechanism, we were able to shift gears (assuming there is pedalling or any rotation of the rear wheel) relatively quickly, at an average rate of 1.4 gears/second. In case the rate was slow or, in contrast, too fast to have enough torque, we could have changed the delay between every step the stepper motor is doing.

The pricing table of all components purchased is shown below:

Table 1 Pricing Table

Product	Quantity	Price in USD
450W/36V Brushed DC Motor	1	63.7
12V SLA Battery	3	30
Hall Effect Sensor	2	1
28-BYJ Stepper Motor	1	2
17HD48002-22B Motor	1	16
A4988 Stepper Driver	1	5
L298N Stepper Driver	1	0.75
High Power Motor Controller	1	27
36V-12V DC-DC Converter	1	5.4
12V-5V DC-DC Regulator	1	1.5
Miscellaneous Electronics	-	45
3D Printing TPU Roll	1	30
Shipping	-	172.5
	Total	460.85

The above calculated total cost is merely an approximation of the product's cost and is certainly due to change slightly when considering some components were already in our inventory, and the possibility of mass-production in case the product gains popularity.

Here are some pictures of the final assembled product:

Figure 57 Final Assembled Product (Motor Side)

Figure 56 Final Assembled Product (Gear Shifting Side)

Figure 55 Final Assembled Product (Dashboard)

Figure 58 Final Assembled Product (Motor and Sensors)

Conclusion

Finally, we were able to build a product that helps with the user's day to day commute, while keeping the cost of the raw materials relatively low compared to other forms of transportations or other E-Bike companies' products. The E-Bike works using electronics, thus reducing CO2 emissions significantly compared to the alternatives.

We would like to say that the project could have been built with better quality and faster completion time, if not for the economic crisis and the coronavirus restrictions. We were able to complete most of the promised features, but we were hoping to add more features instead of eliminating some, for example, the break detection system we discussed in the Project Constraints section.

In the future we would have wanted to include the latter feature as well as more ideas we had in mind. One idea we wanted to implement was an inclination sensor. That way we could alter the PID constants dynamically and help the user go up or down slopes even when in the cruising mode of operation. We would also like to continue working on the project abroad, in hopes of acquiring a stable paying job with any currency other than the Lebanese Pound, not the mention the cheaper components and shipping costs.

References

[1] *17HD48002-22B Motor Datasheet pdf - Stepper Motor. Equivalent, Catalog*. (n.d.). Datasheetspdf. Retrieved February 22, 2021, from https://datasheetspdf.com/pdf/1147974/Busheng/17HD48002-22B/1

[2] *28BYJ-48 Stepper Motor Pinout Wiring, Specifications, Uses Guide & Datasheet*. (n.d.). Components101. Retrieved May 27, 2021, from https://components101.com/motors/28byj-48-stepper-motor

[3] 450w DC 24V 36V gear motor, brush motor electric tricycle, DC gear brushed motor, Electric bicycle - AliExpress. (n.d.). Retrieved January 11, 2021, from https://www.aliexpress.com/item/32606970144.html?spm=a2g0o.productlist.0.0.56a349c5iQ47Ci&algo_pvid=034cfbb7-51aa-4ad4-96dd-56b38310e5f3&algo_expid=034cfbb7-51aa-4ad4-96dd-56b38310e5f3-9&btsid=0bb0620316103997431654194e0ab7&ws_ab_test=searchweb0_0%2Csearchweb201602_%2Csearchweb201603_

[4] alldatasheet.com. (n.d.-a). *A3144 Datasheet(PDF) - Allegro MicroSystems*. ALLDATASHEET. Retrieved May 27, 2021, from https://www.alldatasheet.com/datasheet-pdf/pdf/55092/ALLEGRO/A3144.html

[5] alldatasheet.com. (n.d.-b). *A4988 Datasheet(PDF) - Allegro MicroSystems*. Alldatasheet. Retrieved February 22, 2021, from https://www.alldatasheet.com/datasheet-pdf/pdf/338780/ALLEGRO/A4988.html

[6] alldatasheet.com. (n.d.-c). *L7812CV Datasheet(PDF) - STMicroelectronics*. Alldatasheet. Retrieved February 6, 2021, from https://www.alldatasheet.com/datasheet-pdf/pdf/206457/STMICROELECTRONICS/L7812CV.html

[7] *DC 12V/24V/36V 2 Way PWM Motor Driver Board Module 450W High Power Controller|pwm - AliExpress*. (n.d.). Aliexpress. Retrieved March 4, 2021, from https://www.aliexpress.com/item/32488761340.html?spm=a2g0s.9042311.0.0.5bc84c4ddKhwKJ

[8] *EKT: 36 CONVERTER DC/DC PCB DOWN 20W 40V to 37V DISPLAY*. (n.d.). Katranji. Retrieved March 12, 2021, from https://www.katranji.com/Item/CONVERTER-DC-DC-PCB-DOWN-20W-40V-to-37V-DISPLAY

[9] KOGA. (2019). KOGA E-Nova Automatic. Retrieved January 12, 2021, from https://www.koga.com/en/bikes/e-bikes/collection/e-nova-automatic-lemoncurry.htm?frame=D

[10] *L298N Datasheet*. (n.d.). ST. Retrieved March 12, 2021, from https://www.st.com/content/st_com/en/products/motor-drivers/brushed-dc-motor-drivers/l298.html

[11] *PIC18F4550 Datasheet*. (n.d.). MicroChip. Retrieved August 14, 2020, from https://ww1.microchip.com/downloads/en/devicedoc/39632c.pdf

[12] *Tianneng Lead Acid Battery Tne 12–15*. (n.d.). www.Made-in-China.Com. Retrieved April 26, 2021, from https://cnbmjacky.en.made-in-china.com/product/vSbQockCSORi/China-Tianneng-Lead-Acid-Battery-Tne-12-15.html

Appendix

Appendix A – Microcontroller Configurations

```
#pragma config PLLDIV = 5               // prescaler is not used but is still set for completeness
#pragma config CPUDIV = OSC1_PLL2       // postscaler is not used but is still set for completeness
#pragma config USBDIV = 2               // USB clock selection is not used but is still set for completeness

#pragma config FOSC = INTOSCIO_EC       // internal clock with pin14 as IO pin and pin13

#pragma config FCMEN = OFF              // fail-safe monitor disabled
#pragma config IESO = OFF               // internal/external oscillator switchover bit disabled
#pragma config PWRT = OFF               // power-up timer disabled
#pragma config BOR = OFF                // brown-out reset disabled in hardware and software
#pragma config BORV = 3                 // brown-out reset voltage bits - does not matter since voltage disabled
#pragma config VREGEN = OFF             // USB voltage regulator disabled
#pragma config WDT = OFF                // watchdog timer disabled
#pragma config WDTPS = 32768            // watchdog timer postscale - does not matter since watchdog timer is disabled
#pragma config CCP2MX = ON              // RC1 as CCP2 mux
#pragma config PBADEN = OFF             // RBs set as digital pins
#pragma config LPT1OSC = OFF            // timer 1 as oscillator disabled
#pragma config MCLRE = OFF              // master clear disabled//////////////////////////////////////////////
#pragma config STVREN = ON              // stack full - underflow causes reset
#pragma config LVP = OFF                // ICSP disabled
#pragma config ICPRT = OFF              // ICPORT disabled
#pragma config XINST = OFF              // instruction set extension disabled
#pragma config CP0 = OFF, CP1 = OFF, CP2 = OFF, CP3 = OFF               // code protection bits OFF
#pragma config CPB = OFF                // boot block code protection OFF
#pragma config CPD = OFF                // data EEPROM code protection OFF
#pragma config WRT0 = OFF, WRT1 = OFF, WRT2 = OFF, WRT3 = OFF           // write protection bits OFF
#pragma config WRTC = OFF               // config registers write protection OFF
#pragma config WRTB = OFF               // boot block write protection OFF
#pragma config WRTD = OFF               // data EEPROM write protection OFF
#pragma config EBTR0 = OFF, EBTR1 = OFF, EBTR2 = OFF, EBTR3 = OFF       // table read protection bits OFF
#pragma config EBTRB = OFF              // boot block table read protection OFF
```

Figure 59 PIC18F4550 Configurations List

Appendix B – Microcontroller Initializations

```
// Initializations

OSCCONbits.IDLEN = 0b0;             // sleep mode when SLEEP is called
OSCCONbits.IRCF = 0b111;            // 8MHz INTOSC selected as is
OSCCONbits.SCS = 0b10;              // internal oscillator enabled

while(!OSCCONbits.IOFS) {}          // wait until internal clock is stable

TRISBbits.TRISB0 = 0b1;             // set RB0/INT0 pin as input as external interrupt
TRISBbits.TRISB1 = 0b1;             // set RB1/INT1 pin as input as external interrupt

INTCONbits.GIE = 0b1;               // enable external global interrupts

INTCONbits.INT0IE = 0b1;            // enable external interrupt pin INT0
INTCON2bits.INTEDG0 = 0b0;          // set external interrupt INT0 as falling edge triggered for wheel hall effect sensor detection

INTCON3bits.INT1IE = 0b1;           // enable external interrupt pin INT1
INTCON2bits.INTEDG1 = 0b0;          // set external interrupt INT1 as falling edge triggered for cadence near fall effect sensor detection

INTCONbits.INT0IF = 0b0;            // clear interrupt0 flag
INTCON3bits.INT1IF = 0b0;           // clear interrupt1 flag

INTCONbits.PEIE_GIEL = 0b1;         // enable peripheral interrupts for timers

T3CONbits.RD16 = 0b1;               // set timer3 as 16-bit timer
T3CONbits.TMR3CS = 0b0;             // set timer3 clock source from internal clock (Fosc/4)
T3CONbits.T3CKPS = 0b11;            // set timer3 presecaler value to 1:8
T3CONbits.TMR3ON = 0b0;             // set timer3 initially OFF
PIE2bits.TMR3IE = 0b1;              // set timer3 overflow interrupt enabled
PIR2bits.TMR3IF = 0b0;              // clear timer3 overflow interrupt flag initally
TMR3H = 0x00;                       // clear timer3 higher nibble
TMR3L = 0x00;                       // clear timer3 lower nibble

wheel_overflow_count = 0;           // set wheel revolution timer overflow count to 0 initally
wheel_rev_elapsed_timer_count = 0;  // set revolution total elapsed time to 0 initially
cur_KMPH = 0;                       // set current speed to 0 initially
prv_KMPH = 0;                       // set previous speed to 0 initially
cadence = 0;                        // set current cadence 0 initially
total_pid_integration = 0;          // set integration 0 initially
total_sampling_time_elapsed = 0;    // set total sampling time 0 initially
previous_dynamic_target_speed = 0;  // set previous target speed 0 initially
cruise_emergency_stop = 0;          // cruise emegency stop boolean initially False
```

Figure 61 Software Initialization (Part 1)

```
T0CONbits.T08BIT = 0b0;             // set timer0 to 16 bit timer
T0CONbits.T0CS = 0b0;               // set timer0 clock source as internal clock(Fosc/4)
T0CONbits.T0SE = 0b0;               // set timer0 as rising edge increment timer
T0CONbits.PSA = 0b0;                // enable timer0 prescaler
T0CONbits.T0PS = 0b111;             // set timer0 prescaler as 256
T0CONbits.TMR0ON = 0b0;             // timer0 OFF initially
INTCONbits.TMR0IE = 0b1;            // enable timer0 overflow interrupt
INTCONbits.TMR0IF = 0b0;            // clear timer0 overflow interrupt flag initally
TMR0H = 0x00;                       // clear timer0 higher nibble
TMR0L = 0x00;                       // clear timer0 lower nibble


TRISCbits.TRISC2 = 0b0;             // set Main Motor PWM pin as output
PR2 = pwm_period_register;          // set PWM frequency as 10KHz (-> [8,000,000/(10,000 x 4 x 1)]-1)
normalize_duty_cycle(0);            // set duty cycle initially setting the motor OFF
T2CON = 0b00000010;                 // setup Timer 2 -> No postscaler, initally OFF, prescaler set to 1
TMR2 = 0;                           // clear Timer 2 initally
CCP1CON = 0b00001100;               // setup CCP1 module -> Single Output mode, .0 duty cycle decimal, PWM mode (P1A & P1B are active-high)
T2CONbits.TMR2ON = 0b1;             // turn Timer 2 ON, thus starting the PWM output

// A/D resolution is 10 bits (i.e. [0, 1023])
ADCON0bits.ADON = 0b0;              // analog to digital converter OFF initially
ADCON0bits.GO_NOT_DONE = 0b0;       // analog to digital converter not in progress initially
TRISAbits.RA0 = 0b1;                // set RA0/AN0 as input pin
TRISAbits.RA1 = 0b1;                // set RA1/AN1 as input pin
TRISAbits.RA2 = 0b1;                // set RA2/AN2 as input pin
ADCON1bits.PCFG = 0b1100;           // set AN0, AN1 and AN2 as analog pins
ADCON1bits.VCFG = 0b00;             // set voltage references as VDD and VSS
ADCON2bits.ADFM = 0b1;              // set A/D output as right justified
ADCON2bits.ADCS = 0b101;            // set A/D clock to Fosc/16 (i.e. Tad = 2us > 0.8us which is the minimum period required by the datasheet)
ADCON2bits.ACQT = 0b001;            // set acquisition time to 2xTad (i.e. 4us > 2.45us which is the minimum acquisition time required by the datasheet)
ADRESH = 0;                         // clear A/D output higher nibble register initially
ADRESL = 0;                         // clear A/D output lower nibble register initially

gear_position = 0;                  // AN0 reading initially 0
dash_gear_pot = 0;                  // AN1 reading initially 0
dash_motor_pot = 0;                 // AN2 reading initially 0

TRISDbits.TRISD6 = 0b0;             // set gear shift stepper motor direction as output
LATDbits.LATD6 = 0b0;               // set gear shift stepper motor direction OFF initially

TRISDbits.TRISD4 = 0b0;             // set gear shift stepper motor step as output
LATDbits.LATD4 = 0b0;               // set gear shift stepper motor step OFF initially
```

Figure 60 Software Initialization (Part 2)

```
TRISAbits.TRISA3 = 0b1;             // set assist mode selection switch pin as input
TRISAbits.TRISA4 = 0b1;             // set cruise mode selection switch pin as input

TRISDbits.TRISD3 = 0b1;             // set rear brake flag as input
                                    // set front brake flag as input - RC4 does not need to be set as input pins since it are always input

TRISDbits.TRISD2 = 0b1;             // set speedometer stop flag as input (used for calibration)

TRISAbits.TRISA5 = 0b0;             // speedometer stepper (white) -> closest to power supply input on the driver board
TRISEbits.TRISE0 = 0b0;             // speedometer stepper (brown)
TRISEbits.TRISE1 = 0b0;             // speedometer stepper (white)
TRISEbits.TRISE2 = 0b0;             // speedometer stepper (grey)
LATAbits.LATA5 = 0b0;               // initially OFF
LATEbits.LATE0 = 0b0;               // initially OFF
LATEbits.LATE1 = 0b0;               // initially OFF
LATEbits.LATE2 = 0b0;               // initially OFF

calibrate_speedometer_stepper(40);          // calibrate speedometer position and set current speedometer step 0 initially with limit 40 steps (38 is the whole visible range of the speedometer)
```

Figure 62 Software Initialization (Part 3)

Appendix C – Microcontroller Constants

```
// Global Constants
#pragma udata constants
const unsigned char pwm_period_register = 249;
const unsigned char pwm_break_thresh = 85; // in %
const double potentiometer_to_pwm_CONST = 0.097752; // 100/1023
const unsigned char speedometer_calibration_recheck_steps = 3;  // this will be the number of steps taken away and
//const unsigned int gear_stepper_step_size = 200;  //200steps=180deg
const double samples_timeout_in_s = 2.5; // 2.5 or 5.0
const float K = 1;      // PID constant
const unsigned char target_speed_reading_error_margin = 2;  // target_speed will be considered the same in the ran
const unsigned char gear_error_margin = 20; // correct gear will be set between [target-err, target+err]
const unsigned int gear_1 = 0;              // gear 1 analog reading from gear position slider(10k potentiometer)
const unsigned int gear_2 = 256;            // gear 2 analog reading from gear position slider(10k potentiometer)
const unsigned int gear_3 = 512;            // gear 3 analog reading from gear position slider(10k potentiometer)
const unsigned int gear_4 = 768;            // gear 4 analog reading from gear position slider(10k potentiometer)
const unsigned int gear_5 = 1023;           // gear 5 analog reading from gear position slider(10k potentiometer)
const unsigned int gear_1_thresh_KMPH = 3;              // gear 1 threshold speed
const unsigned int gear_2_thresh_KMPH = 6;          // gear 2 threshold speed
const unsigned int gear_3_thresh_KMPH = 8;          // gear 3 threshold speed
const unsigned int gear_4_thresh_KMPH = 12;         // gear 4 threshold speed
const unsigned int gear_5_thresh_KMPH = 15;         // gear 5 threshold speed
const unsigned int cadence_threshold = 10;  // RPM threshold when we assume that below it the user is pedaling ligh
const double RPH_CONST = 900000000;     // (1,000,000s x 3,600h) / 4us  --> (i.e. 4x10^(-6) to convert to seconds,
const unsigned short long RPM_CONST = 468750;// each timer increment is (4x256)/8,000,000 = 128us --to convert to s
const unsigned short long RPS_CONST = 250000; // timer3 -> (1,000,000s/4us) = 250,000
const float circumference = 0.002042;   // considering radius is 32.5cm, circumference = 2 x pi x 32.5cm x 10^(-5)
const unsigned char max_wheel_overflow_count = 7;       // how many wheel revolution count overflows are allowed b
                        // max elapsed t = (65,535 + [65,535 x max(overflows)]) x (1/[8,000,000/(4 x prescaler
                        // (4 with 8 prescaler -> a little over 1 second elapsed) (20 with 8 prescaler ->  5.5
```

Figure 63 Global Constants

Appendix D – Speedometer's Stepper 1-Step Method

```
void speedometer_stepper_step(unsigned char direction) {    //12steps = 360deg -> turn on every other coil of the stepper in the below patterns

    unsigned char d = 25; // 25

    if(direction) {
        if(LATAbits.LATA5 && !LATEbits.LATE0 && !LATEbits.LATE1 && LATEbits.LATE2) {
            LATAbits.LATA5 = 0b1;
            LATEbits.LATE0 = 0b0;
            LATEbits.LATE1 = 0b1;
            LATEbits.LATE2 = 0b0;

            Delay10KTCYx(d);
        } else if(LATAbits.LATA5 && !LATEbits.LATE0 && LATEbits.LATE1 && !LATEbits.LATE2) {
            LATAbits.LATA5 = 0b0;
            LATEbits.LATE0 = 0b1;
            LATEbits.LATE1 = 0b1;
            LATEbits.LATE2 = 0b0;

            Delay10KTCYx(d);
        } else if(!LATAbits.LATA5 && LATEbits.LATE0 && LATEbits.LATE1 && !LATEbits.LATE2) {
            LATAbits.LATA5 = 0b0;
            LATEbits.LATE0 = 0b1;
            LATEbits.LATE1 = 0b0;
            LATEbits.LATE2 = 0b1;

            Delay10KTCYx(d);
        } else if(!LATAbits.LATA5 && LATEbits.LATE0 && !LATEbits.LATE1 && LATEbits.LATE2) {
            LATAbits.LATA5 = 0b1;
            LATEbits.LATE0 = 0b0;
            LATEbits.LATE1 = 0b0;
            LATEbits.LATE2 = 0b1;

            Delay10KTCYx(d);
        } else {
            LATAbits.LATA5 = 0b1;
            LATEbits.LATE0 = 0b0;
            LATEbits.LATE1 = 0b1;
            LATEbits.LATE2 = 0b0;

            Delay10KTCYx(d);
        }
```

Figure 64 Speedometer 1-step method (Part 1)

```
    } else {
        if(LATAbits.LATA5 && !LATEbits.LATE0 && !LATEbits.LATE1 && LATEbits.LATE2) {
            LATAbits.LATA5 = 0b0;
            LATEbits.LATE0 = 0b1;
            LATEbits.LATE1 = 0b0;
            LATEbits.LATE2 = 0b1;

            Delay10KTCYx(d);
        } else if(!LATAbits.LATA5 && LATEbits.LATE0 && !LATEbits.LATE1 && LATEbits.LATE2) {
            LATAbits.LATA5 = 0b0;
            LATEbits.LATE0 = 0b1;
            LATEbits.LATE1 = 0b1;
            LATEbits.LATE2 = 0b0;

            Delay10KTCYx(d);
        } else if(!LATAbits.LATA5 && LATEbits.LATE0 && LATEbits.LATE1 && !LATEbits.LATE2) {
            LATAbits.LATA5 = 0b1;
            LATEbits.LATE0 = 0b0;
            LATEbits.LATE1 = 0b1;
            LATEbits.LATE2 = 0b0;

            Delay10KTCYx(d);
        } else if(LATAbits.LATA5 && !LATEbits.LATE0 && LATEbits.LATE1 && !LATEbits.LATE2) {
            LATAbits.LATA5 = 0b1;
            LATEbits.LATE0 = 0b0;
            LATEbits.LATE1 = 0b0;
            LATEbits.LATE2 = 0b1;

            Delay10KTCYx(d);
        } else {
            LATAbits.LATA5 = 0b0;
            LATEbits.LATE0 = 0b1;
            LATEbits.LATE1 = 0b0;
            LATEbits.LATE2 = 0b1;

            Delay10KTCYx(d);
        }
    }
}
```

Figure 65 Speedometer 1-step method (Part 2)

Appendix E – Gear Shifting's Stepper 1-Step Method

```
void gear_stepper_step(unsigned char direction) {   //200steps = 180deg -> D6 chooses the stepper driver's direction, and D4 the step intialization
    LATDbits.LATD6 = direction;

    LATDbits.LATD4 = 0b1;
    Delay10TCYx(100);
    LATDbits.LATD4 = 0b0;
    Delay10TCYx(100);
}
```

Figure 66 Gear Shifting Stepper 1-Step Function

Appendix F – 450W H-Bridge Driver PWM Normalization

```
void normalize_duty_cycle(unsigned char duty_cycle_percentage) {

    if(duty_cycle_percentage == 255) {
        CCPR1L = pwm_period_register;   // BREAK
        return;
    }

    duty_cycle_percentage = 100 - duty_cycle_percentage; // H-Bridge used needs the pwm flipped: 100% -> 0V & (0+)% -> 36V
    if(duty_cycle_percentage > pwm_break_thresh) {
        duty_cycle_percentage = pwm_break_thresh;
    }

    CCPR1L = ((double)duty_cycle_percentage * ((double)pwm_period_register / 100.0));

}
```

Figure 67 Duty Cycle Normalization

Appendix G – Analog to Digital Conversion for Potentiometer Readings

```
void read_AN0(void) {

    ADCON0bits.ADON = 0b1;              // analog to digital converter ON to find potentiometer value
    ADCON0bits.CHS = 0b0000;            // set A/D to channel 0 (i.e. AN0)
    ADCON0bits.GO_NOT_DONE = 0b1;       // analog to digital converter in progress
    while(ADCON0bits.GO_NOT_DONE){ }    // wait till the conversion is done
    gear_position = ADRESH;
    gear_position = gear_position << 8;
    gear_position += ADRESL;        // assign output by assigning higher nibble, shifting 8 bits and adding lower nibble
    ADRESH = 0;
    ADRESL = 0;
    ADCON0bits.ADON = 0b0;              // analog to digital converter OFF when finished

}

void read_AN1(void) {

    ADCON0bits.ADON = 0b1;              // analog to digital converter ON to find potentiometer value
    ADCON0bits.CHS = 0b0001;            // set A/D to channel 1 (i.e. AN1)
    ADCON0bits.GO_NOT_DONE = 0b1;       // analog to digital converter in progress
    while(ADCON0bits.GO_NOT_DONE){ }    // wait till the conversion is done
    dash_gear_pot = ADRESH;
    dash_gear_pot = dash_gear_pot << 8;
    dash_gear_pot += ADRESL;        // assign output by assigning higher nibble, shifting 8 bits and adding lower nibble
    ADRESH = 0;
    ADRESL = 0;
    ADCON0bits.ADON = 0b0;              // analog to digital converter OFF when finished

}

void read_AN2(void) {

    ADCON0bits.ADON = 0b1;              // analog to digital converter ON to find potentiometer value
    ADCON0bits.CHS = 0b0010;            // set A/D to channel 2 (i.e. AN2)
    ADCON0bits.GO_NOT_DONE = 0b1;       // analog to digital converter in progress
    while(ADCON0bits.GO_NOT_DONE){ }    // wait till the conversion is done
    dash_motor_pot = ADRESH;
    dash_motor_pot = dash_motor_pot << 8;
    dash_motor_pot += ADRESL;       // assign output by assigning higher nibble, shifting 8 bits and adding lower nibble
    ADRESH = 0;
    ADRESL = 0;
    ADCON0bits.ADON = 0b0;              // analog to digital converter OFF when finished

}
```

Figure 68 Analog Read of the Potentiometers