Final Documentation (FD)

Project: Autonomous car

Company: REVO

Place, date: Enschede, 22nd March 2024

Drawn up by: Illia Guzerya

Anton Bucataru

Obidkhon Akhmadkhonov

Oleh Bielous

Version: 1.0

Place, date:	Enschede, 31st May 2024		
Made by:	Project group 5		
	Illia Guzerya	547753	547753@student.saxion.nl
	Oleh Bielous	558059	558059@student.saxion.nl
	Anton Bucataru	538412	538412@student.saxion.nl
	Obidkhon Akhmadkhonov	553160	553160@student.saxion.nl
Version number:	[1.0]		
Clients:			
Waseem Elsayed			
Asif Khan			

Table of Contents

Table of figures

1 Abstract

This report presents the systematic approach Team REVO used to design, build, and optimize an autonomous race car capable of competing in autonomous motor racing, called "REVO." In a project that affected hardware and sophisticated software components, the vehicle was designed to meet specific competition standards with a strong focus on speed, accuracy and for hitting the objectives of the project, methodology, key discoveries, and contributions to the development of autonomous running technology.

2 Introduction

Constructing an autonomous racing car proficient in line tracking is challenging, and many considerations must be addressed. These considerations include the project's budget, the race duration, the optimal selection of a processing platform, the appropriate motor, and the choice of programming language for software development. The team conducted several rigorous discussions to address these questions. The main dilemma was prioritizing speed or accuracy. However, united by the goal of securing victory in the race, both elements were integrated into the design strategy.

This project is implemented by a team of computer science students at Saxion University of Applied Sciences. The autonomous racing car "REVO" aims to not only compete but to win the race against other students from different faculties.

At the beginning of the project, the team outlined several critical **objectives**:

1. To optimize speed management.

2. To develop and refine line-tracking/object-avoidance systems.

3. To develop structured, highly factored software code.

These objectives guided the project's planning and execution phases.

To meet these objectives, the project was structured around a systematic development process. Initially, the background and context of self-driving racing cars were researched to build a strong theoretical base for all the team members. Aharari & Ueda (2019) showed that the colour sensor is not very precise because of different factors that can appear during testing, such as sun or laps light dust.

Following this research, it was determined that a stable colour-checking procedure is needed for the line-following part of the project. Additionally, for the construction of the machine, reference was made to Instructible (2018) because its creator described in detail all the functionalities and methods of its programming, making it a valuable source of information for the team. For example, Parekh et al. (2022) emphasized the importance of focusing on safety when creating an obstacle detection system, noting that deficiencies in such systems can lead to catastrophic outcomes. Although the project is focused on competitive racing, it similarly prioritizes safety, extending these considerations beyond the immediate racing context.

Following this foundational research, the project defined a clear problem statement: to create a vehicle that effectively balances accurate line tracking and speed management.

The team REVO used the MoSCoW methodology to approach the stated problem, which allowed continuous improvements based on the results of testing. Enhanced communication during the execution phase facilitated software development and sensor integration into the project. To conclude this segment, the organization of the report will be outlined. Following this section, the project requirements, design, and development phases, testing procedures, and final outcomes will be discussed.

3 Execution of assignment

3.1 Project Activities

Project setup

1. Project plan.
2. User requirements.
3. Acceptance test plan.

System Requirements

1. Functional requirements.
2. Technical requirements.
3. System test plan.

Functional Design

1. Concepts.
2. Choosing a concept.
3. Functional design concept chosen.

Technical design

1. Technical block diagram.
2. Calculating and selecting essential components.
3. 3D mechanical design.
4. Module/Unit test plan.

Realisation

1. Mechanical – Making components and module assembling.
2. Electrical-Electronic.
3. Software – module/unit coding.

Module/Unit Test

1. Module integration.
2. Test.

Finalization project

1. Final report & Documentation and Final presentation

3.2 Project Boundaries

Start day: 6th of May 2024
End day: 31 of May 2024
Length of the project: 25 days+
Budget: 50€

REVO needs to decide on the technical approach to be chosen. The preliminary design of the vehicle must be planned for the prototype, and a prototype of the product needs to be created for testing purposes. All important documents must be completed and submitted. The parts list should include the required parts, their prices, and links to the parts.

What doesn't belong in this project includes a website promoting the company and product, as well as personnel other than the Revo team working on the project.

3.3 Quality assurance

The desired quality will align with the project objectives outlined in the Introduction section under Objectives. Prior to product production, an order list, project plan, and both functional and technical designs must be developed and presented for feedback. This allows for potential improvements or alterations to the product's conceptualization.

Product quality will undergo phase tests, including Module/Unit Test: hardware and software.

External advice may be sought from other parties or experts to assess intermediate product results.

Some tools that were used:

1. Arduino IDE;
2. Google;
3. ChatGPT;
4. FabLab;
5. Autodesk software for modeling 2D/wood;
6. Blender software for 3D modeling;
7. Multimeter.

Costs and benefits

Project costs:

- Labor: 160 hours per person.
- Main budget: Maximum of 50 euros, less is better.

Benefits

The project offers opportunities to apply the specific Body of Knowledge and Skills of ACS , gain proficiency in Arduino and various Shields and Modules for data processing and control, conduct calculations, simulations, and circuit design, perform electrical and mechanical measurements, software coding, and test measurements.

Additionally, it involves implementing various electronic modules and constructing an autonomous driving electrical car.

Finally good to mention, that members of team Revo fulfil roles as chairman and different engineers in group meetings, demonstrating personal knowledge and skill improvement to evaluators.

4 Functional Design overview

The project aims to create a vehicle capable of autonomously following a line. This is achieved through the integration of hardware and software, resulting in a car that moves independently along a predetermined path. The core mechanism involves a sensor detecting light reflections off the line, which, after processing, guides the car forward and steers it along any turns in the path by adjusting the wheels.

This document outlines the functional design of the project, elucidating fundamental principles and the vehicle's conceptual framework. It includes detailed descriptions of both the software and hardware components involved. Additionally, it covers the requirements analysis and the electronic aspects essential for project realization. Furthermore, it discusses the project's management and conceptual strategies. All pertinent information and project details are contained herein.

4.1 Analysis of requirements

This analysis must aim at proving the principal feasibility of the functional and technical requirements as formulated in the SR- This chapter presents the objectives and methodologies for developing an autonomous vehicle project as stipulated by the stakeholders and our team.

The focus is on creating a self-guiding car that operates within specific physical dimensions and timeframes, incorporates eco-conscious practices, and remains within the team's capabilities and budgetary constraints.

Requirements Set by Stakeholder:

1) Ride autonomously.
   Approach: Develop a program that autonomously guides the car using sensor feedback.
2) Work for at least 30 minutes.
   Approach: Ensure through calculations and measurements that the vehicle's electrical features meet performance standards.
3) Follow the track line (20mm).
   Approach: Implement sensors to detect and follow the line.
4) Stop at the end of the track.
   Approach: Devise a method to detect the end of the track and signal the program to halt.
5) Avoid obstacles.
   Approach: Integrate sensors to detect obstacles and program the car to take appropriate actions, such as navigating around them.
6) Find the track line.
   Approach: Use sensors or a camera to locate and follow the track line.
7) Have a body (Separate parts of a car shield).
   Approach: Produce an wooden model assembled by pieces to ensure ease of process and its perspective in testing (wood cutting is tremendously faster than 3d printing). Finish with the shield that can be attached to the body of RC car.

4.2 Elaboration of chosen principle to functional design

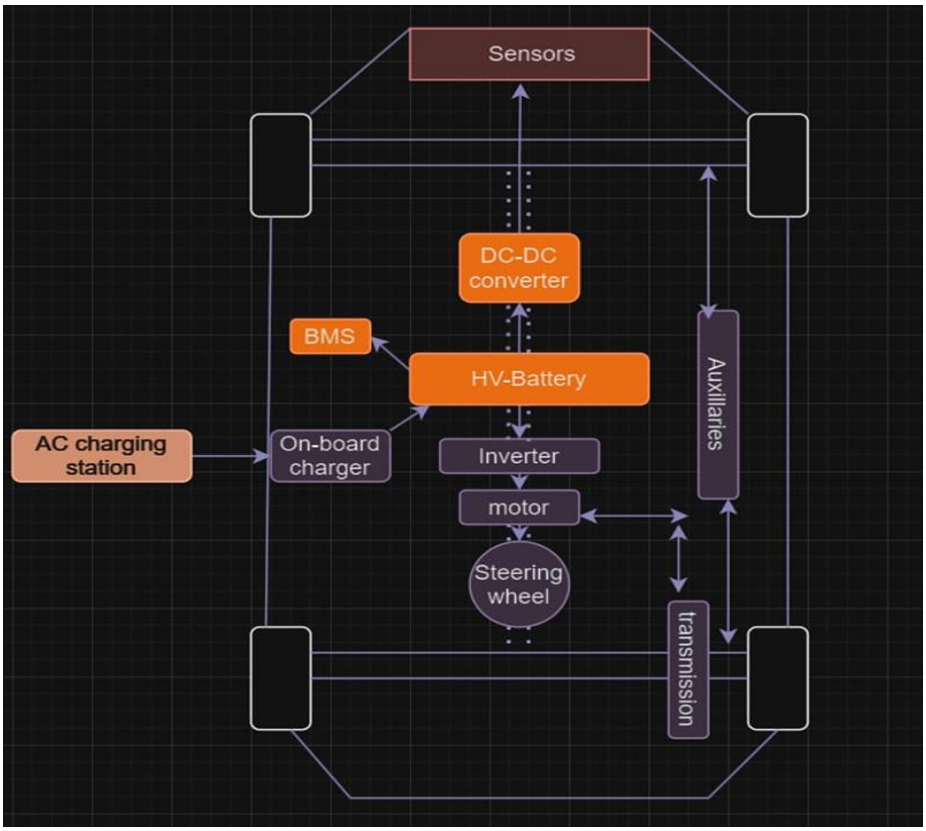

Figure 1 - Block diagram

The chosen design concept features a four-wheel drive system powered by a DC motor, controlled by a microcontroller. The system uses a combination of ultra-sonic and AR sensors for navigation and obstacle detection. The chosen design of the car is divided into 4 sub-categories: mechanical, electrical, sensors and microcontroller.

4.3 Mechanical layout

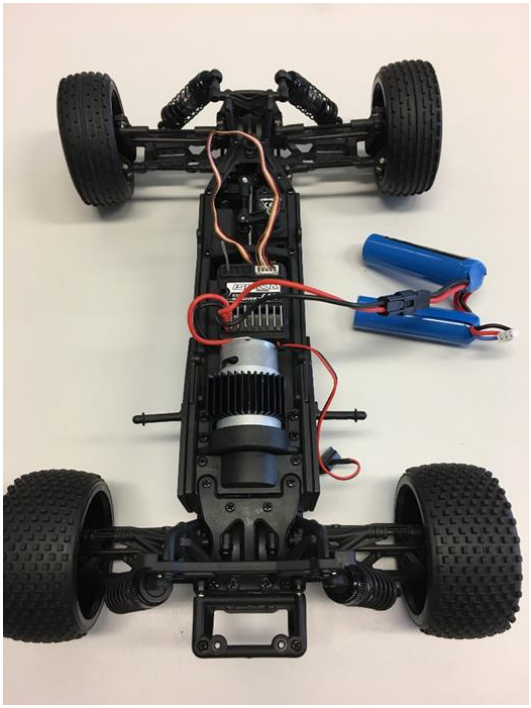

Figure 2 - ISH-010 base frame

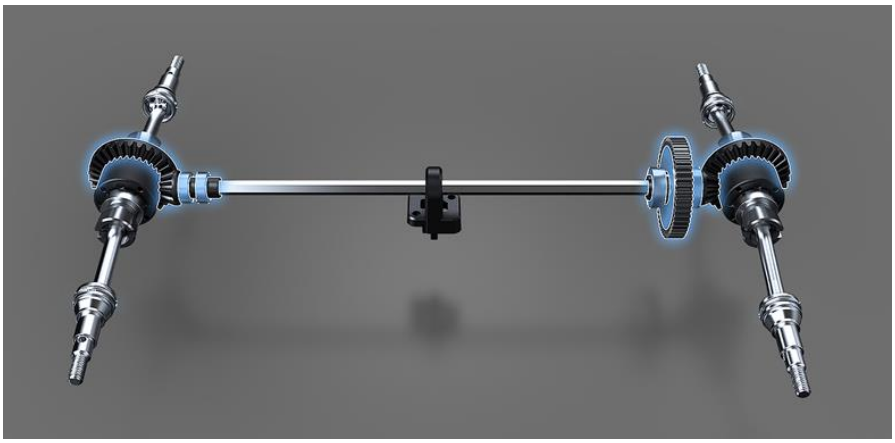

Figure 3 - Transmission and steering wheel

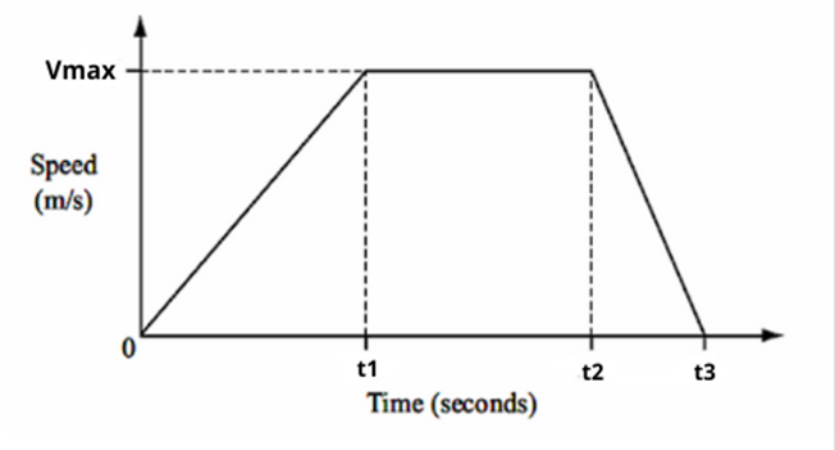

Figure 4 - Velocity to Time relationship

The mechanical component of our design is built for durability and precise control. Here's a breakdown of the key mechanical parts:

- Motor - converts DC to mechanical energy to produce motion.
- Steering wheel - connected to the motor and rotates in Z-axis to move.
- Transmission - connected to the steering wheel and rotates the wheels in X-axis.
- Auxiliaries - to provide better bounce-tolerance.
- ISH-010 – base frame provided by Saxion. It's the foundation to which all mechanical parts are anchored.

4.4 Electrical part

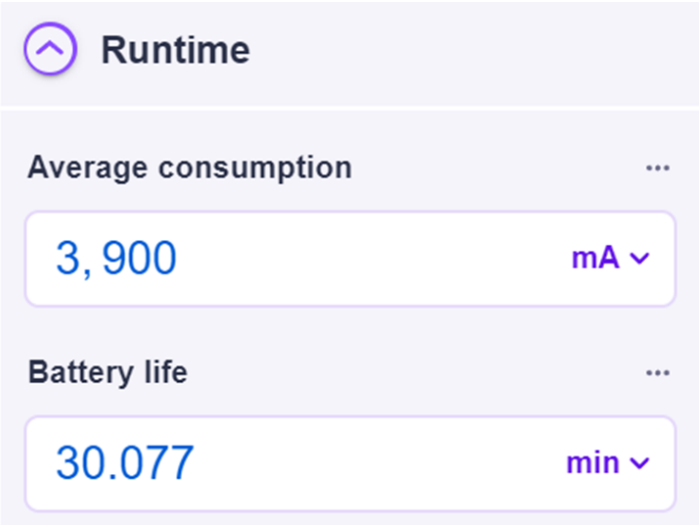

Figure 5 - Runtime of the battery calculator

The electrical system is centered around a NiMh 8,2V 2400mAh battery, ensuring a minimum operational time of 30 minutes. The discharge safety is set to 15% to avoid over-discharge. Voltage conversions for various components are carefully planned to ensure compatibility and efficient power distribution.

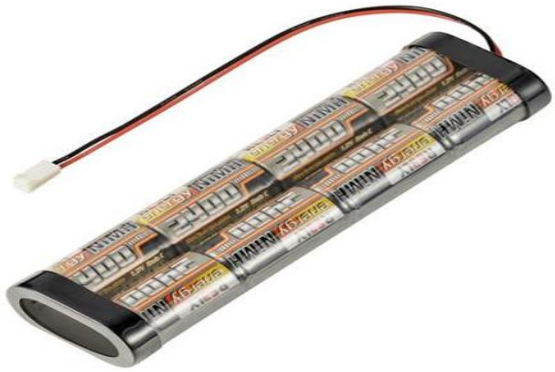

Figure 6 – REELY - NiMh Battery 8.2V,2400mAh

The electrical system layout arranges all the parts and wires to make sure everything gets the right amount of power.

- HV Battery: It's the main power source and comes with a smart system to keep it running 30 minutes non-stop.
- BMS - carefully monitors and controls the battery's charge and discharge processes to prevent damage and extend the battery's life.
- DC-DC convertor- steps down the high voltage to lower voltages suitable for different parts of the system.
- Motor Driver - transforms direct current (DC) from the battery into AC for the motor.
- On-board charger - manages the charging of the HV battery from an AC charging station.
- AC charging station - provides an interface for the vehicle to connect to the electrical grid.

4.5 Electronic

There are lots of options that we could choose as a base for our project. For example:

Usage of personal computer, implementing project using Raspberry Pi single boarded computer, or maybe use USP-32. Also, there is a broad field on choosing the way to implement main purpose of autonomy of our car – we could use Camera for both line tracking and obstacle avoidance or use different kinds of sensors.

Nonetheless we have stopped on the Arduino as we concluded that it will be easiest and most efficient option regarding flow of information and its communication. In such a case we would only additionally use sensors and would only need only USB connection for Arduino to communicate with it and update the code. A switch or button could be used to start the main algorithm when the system is turned on.

4.6 Sensors

Sensors are a crucial aspect of the electrical design, providing the necessary data for navigation and interaction with the environment. Line Tracking Sensor: The vehicle is equipped with infrared light

sensor and infrared LED responsible for detecting the white line on the track. The IR LED illuminates a surface with infrared light; the sensor then picks up reflection and based on intensity distinguishes the contrast between the track and the line, providing input to the control system to ensure that the car stays on course. Obstacle Avoidance Sensor: To navigate around obstacles, ultrasonic sensors are placed around the vehicle. These sensors produce high-frequency sound waves that reflect from the objects in the vehicle's path. The time it takes for the echoes to return is calculated to determine the distance to potential obstacles. Finally, the accelerometer/gyroscope sensor will use this information to navigate the car avoiding collisions or to detect the upcoming ramp.

The information from these sensors is sent to the car's main control unit (Arduino), where the software processes the data to make decisions about the vehicle's movements.

4.7 Software

Software approaches and options used:

- Integrated development environment: Arduino IDE, Clion, Visual Studio Code.
- Programming languages: C/C++
- Arduino features such as Interrupts, external libraries.
- Main algorithms – line detection, obstacle detection, ramp detection.

Main principle of the software part: code is implemented within a loop that iterates through the whole program while Arduino is working, what lets us to periodically update all information within the system to successfully adjust the trajectory and acceleration of our car in real time to complete the track.

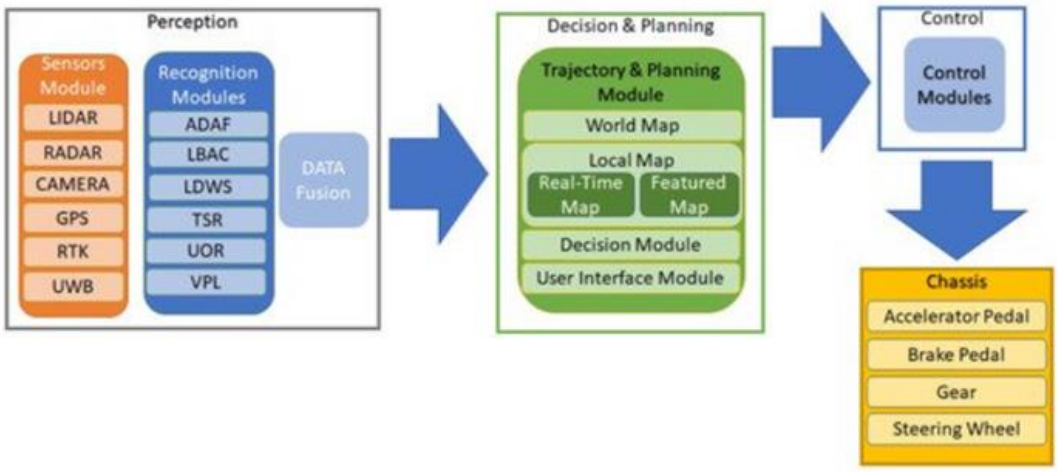

Figure 7 - System Architecture for AVs

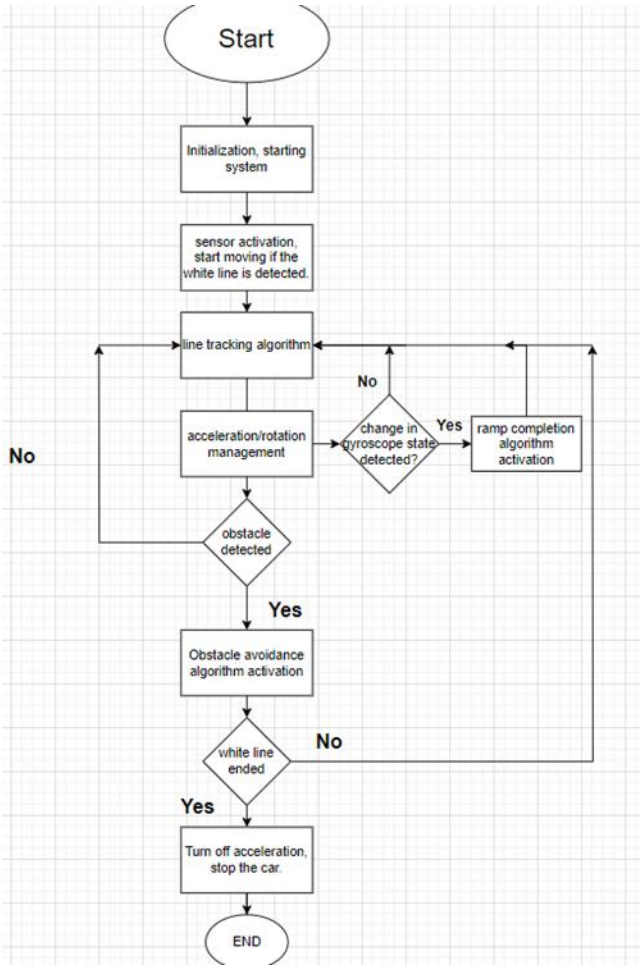

Figure 8 - Flow chart describing main algorithm of our program.

4.8 Integration concerns

Speed:

After analysing videos on Saxion's "Saxion - Autonomous RC Challenges" YouTube channel [3] our team concluded that we aim on passing the given track in under 45 seconds.

Accuracy:

- Obstacle: Our idea is that car will measure distance to the obstacle in front of it with help of ultrasonic sensors. In such case a predefined algorithm of obstacle avoidance will be activated.
- Line tracking: Program will continuously execute line tracking algorithm, looking for a white line using a set of Infrared sensors that will help to distinguish the line and follow it accurately. Position of the car relatively to white line will affect its acceleration so it can speed up during straight parts and slow down during turns.

5 Technical design overview

5.1 Overview of initial hardware design

The design includes the following main components:

- Object Detection System (Ultrasonic sensors): Detects obstacles in the vehicle's path.
- Processing Platform (Arduino UNO): Serves as the brain of the vehicle, processing input from sensors and controlling actuators.
- Ramp Detection System (Gyroscope/accelerometer): Detects inclines and declines on the path.
- Wheel Steering Mechanism (Rotating DC motor): Controls the steering of the vehicle.
- Line Detection System (IR sensors): Identifies and follows the line on the track.
- Power Supply (Li-ion Battery / NiMh accupack): Provides power to the system.

5.2 Interfaces

The revised design would incorporate these additional components as follows:

- *The Arduino UNO is centrally connected to all sensors and the motor driver, orchestrating the vehicle's operations.*
- *Ultrasonic sensors are connected directly to the Arduino, with signal conditioning if necessary.*
- *The Gyroscope is connected to the Arduino, potentially through an interface module for signal compatibility.*
- *The Rotating DC motor is connected via a Motor Driver, which is controlled by the Arduino.*
- *IR sensors for line detection are directly connected to the Arduino.*
- *A Voltage Converter is used where necessary to match the power supply voltage with component requirements.*
- *This setup ensures that all components can communicate and function together effectively, with the Arduino UNO acting as the central processing unit. Adjustments and additions of interface components like voltage converters, motor drivers, and resistors ensure that the electrical and signal requirements of each component are met, allowing for seamless integration and operation of the autonomous vehicle.*

5.3 Technical requirements

5.3.1 Component #1 (e.g. Li-ion Battery)

#	Requirement	Relation	Value	Unit
TR001	Maximum voltage level	≤	7.4	Volt
TR002	Number of cycles	>	1000	Cycles
TR003	Complete discharge time for single charge at maximum power	>	30	minutes

- If **TR001** ensures compatibility with the vehicle's electrical system, preventing overvoltage issues.
- **TR002** ensures the battery's longevity, reducing the need for frequent replacements.

- **TR003** guarantees that the vehicle can operate sufficiently long on a single charge, crucial for completing tasks without mid-operation recharging.

5.3.2 Component #2 (RC 540 motor)

The RC 540 motor, selected for propelling the vehicle, needs to match the electrical system's output and not exceed current limits to maintain efficiency and prevent overheating.

#	Requirement	Relation	Value	Unit
TR011	Nominal/operating voltage level	=	7.4	Volt
TR012	Maximum current level	≤	17	Amps

- **TR011** was chosen to match the motor with the system's power supply ensuring efficient operation.
- **TR012** limits the motor's current draw to prevent overheating and ensure the system's energy efficiency.

5.3.3 Software function #1 (e.g. RampControl)

#	Requirement
TR101	When ValueSensor > 300, then PowerMotorIncrease()
TR102	When ValueSensor < 0, then PowerMotorDecrease()

- **TR101** increases motor power when an upward incline is detected, ensuring the vehicle can climb effectively.
- **TR102** decreases power in a downward incline or flat terrain to conserve energy and control speed.

5.3.4 Software function #2 (e.g. ObstacleDetection)

ObstacleDetection software function enables the vehicle to navigate around obstacles by steering accordingly based on sensor input, crucial for avoiding collisions and ensuring smooth operation.

#	Requirement
TR111	When ValueSensor > 15, then SteeringRight()
TR112	When ValueSensor < 0, then SteeringLeft()
TR113	When ObstacleDetection=TRUE, then PowerMotorDecrease()

- **TR111** and **TR112** guide the vehicle to steer away from obstacles, ensuring it remains on its intended path.
- **TR113** initiates a slowdown or stop when an obstacle is detected too close, enhancing safety.

5.3.5 Mechanical Requirements for Vehicle Chassis

#	Requirement	Relation	Value	Unit
TR021	Length	=	380	mm
TR022	Width	=	255	mm
TR022	Height	=	155	mm
TR023	Weight	=	1410	g

- **TR021** and **TR022** ensure the vehicle's size is suitable for the track, providing enough surface area for component installation while enabling maneuverability.
- **TR023** limits the vehicle's height to ensure stability by maintaining a low center of gravity.

- **TR024** targets a lightweight design for the chassis to enhance efficiency and reduce power consumption.

5.3.6 Mechanical Requirements for Wheel and Steering System

The wheel and steering system directly influence the vehicle's ability to navigate the track accurately.

#	**Requirement**	**Relation**	**Value**	**Unit**
ID	Requirement	Relation	Value	Unit
TR031	Wheel Diameter	=	88	mm
TR032	Wheel Width	=	42	Mm (front)
TR033	Wheel Width	=	42	Mm (rear)
TR034	Steering Range	≥	30	Degrees
TR035	Material	-	Rubber	-

- **TR031, TR032** and **TR033** define the dimensions of the wheels to ensure they provide adequate traction and are capable of handling the track's terrain.
- **TR033** specifies the minimum steering range to ensure the vehicle can navigate turns effectively.
- **TR034** indicates the choice of material for the wheels, emphasizing traction and durability.

5.4 Detailed technical design

5.4.1 Mechanical

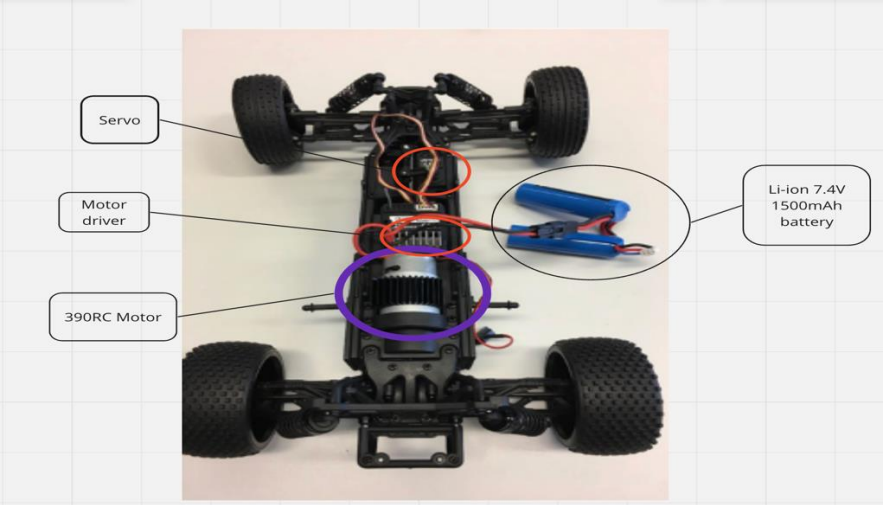

Figure 9 - Car components

All the basic car components are shown inside the car base (Fig. 9), excluding the Arduino and sensors.

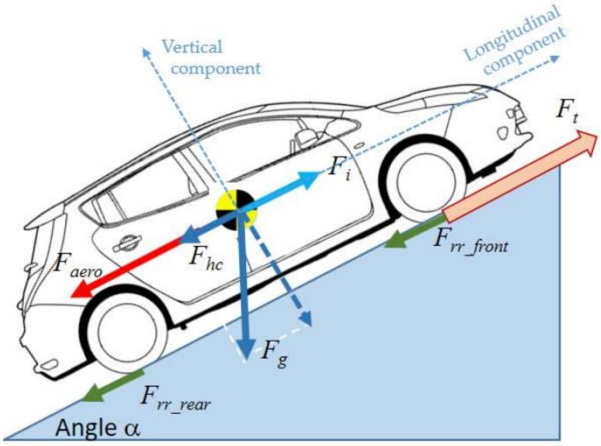

Figure 10 - Show the chassis of our car project which has rear wheel power drive.

5.4.2 Electrical

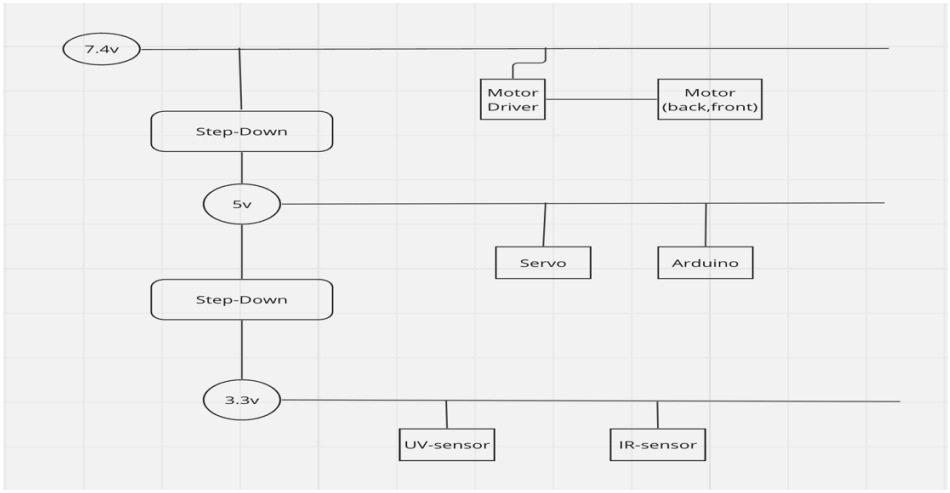

Figure 11 - One line diagram

The one-line diagram above describes how our energy is being distributed through all of the components, here you can see that raw power of 7.4 goes directly to the motor drove and main motor, after a step-down convertor the load becomes 5 volts which is distributed to the servo motor (is used for turnings) and to the Arduino itself, after the last step-down convertor which is inbuilt in Arduino power becomes 3.3 volts and is distributed to the sensor.

Battery: Li-ion 2s 1p 7.4v 1500mAh

Safe discharge: reducing the capacity at which a Li-ion battery (also known as lithium-ion) is regarded as fully discharged for protection purposes,

however, no matter how low it might be, lithium-ion batteries cannot be discharged to below two point five volts while Li-Po batteries can't go below three volts.

(Do not discharge below 3 volts or battery damage will occur!)

30 minutes of power can be obtained by discharging the battery at 80 % of its capacity without risking over discharge (while continuously pulling a current of 2.5 amps).

Voltage conversions:

- Charger: Li-Po 7.4V

- 7.4v Li-ion-> step-down converter (5v,3.3v etc)

- 7.4v Li-ion-> Motor Driver-> Motor(s)

- Step-down converter (5V)-> servo motor, Arduino.

- Down converter 3.3v-> Sensors

The electrical flow chart represented above shows to the reader how the power load is flowing to the components, starting with Battery which is on the top going to the Servo and Motor driver, after that is flowing through DC convertor to the Arduino and after that it is distributed to the sensors.

5.4.3 Power Consumption

Given following **energy sources**:
Reely NiMh accupack 8.2V 2400mAh 15C

Energy **consumers**:

- S60PH Sport Servo motor (average 1-2 Amps)
- Pololulu qtr-8rc InfraRed sensor (average 90-100 mA)
- HC-SR04 Ultrasonic sensor (average 2mA)
- Brushed DC motor 540 (average maximum consumption in our case: (2,5 Amps), average default consumption: (1.1 Amps).
- Arduino UNO (average 100mA)

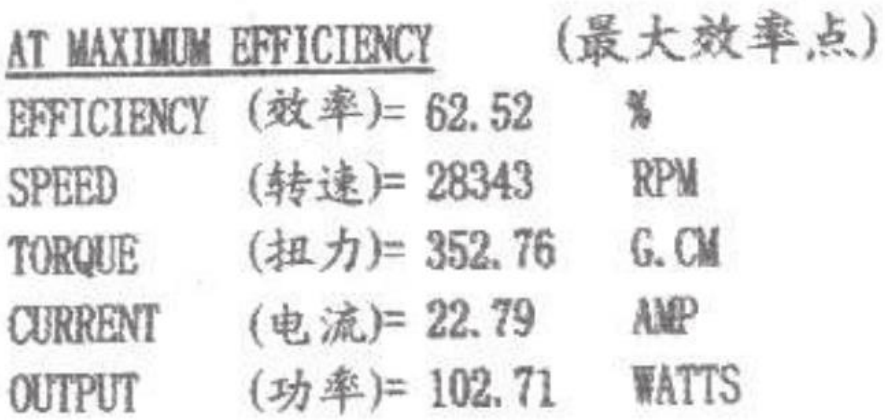

Figure 12 - Brushed Dc motor 540 power consumption at maximum efficiency datasheet screenshot

Average **maximum** consumption = 1.8 (worst case scenario consumption where car is expected to go half the time of usage at maximum speed that is used in program -> default(1.1A), ramp mode(2.5)) + 0.095A (IR sensor) + 0.002A (Ultrasonic sensor) + 2A (servo motor) + 0,1A (Arduino) = 4A

Using batter life formula we can calculate approximated time of work with maximum consumption:

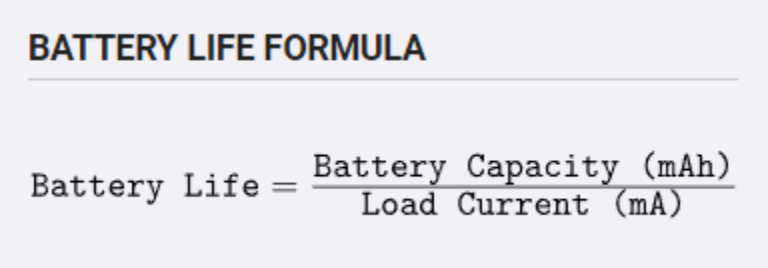

Figure 13 - Battery life formula

Accounting for battery discharge safety (at 15%) we say that our capacity is 2400 – 360(15%) = 2040

Battery life = 2040mAh / 4000 mA = 0.5 Hours = 30 **minutes**.

Theoretically speaking our car will be able to function 30 minutes in a row.

5.4.4 Electronic

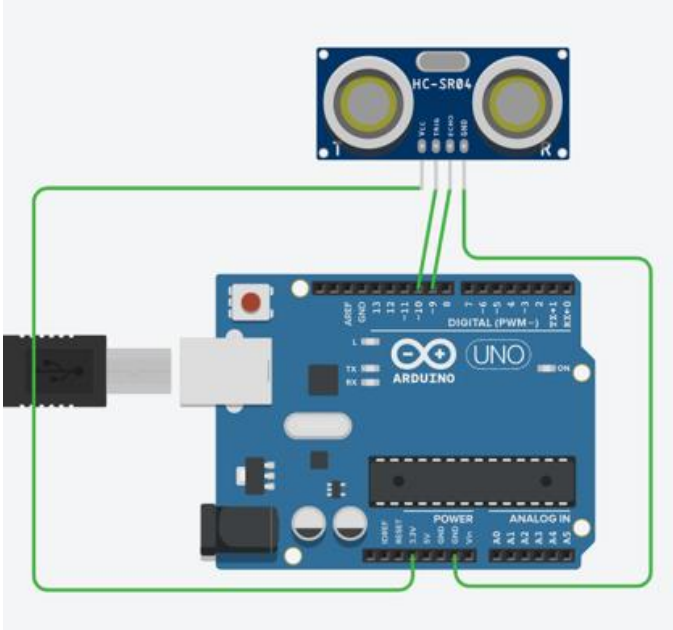

Figure 14 - Ultrasonic connection

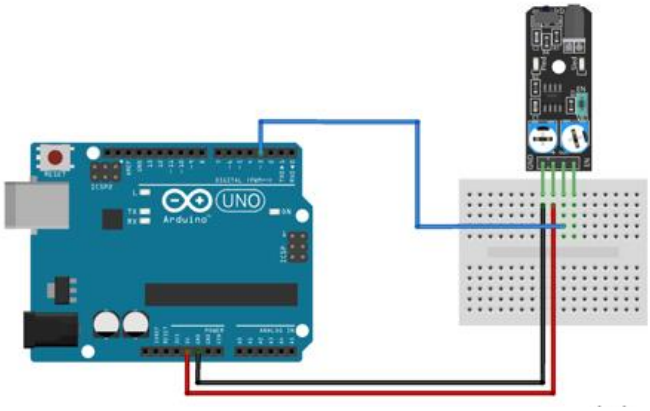

Figure 15 - Infrared connection

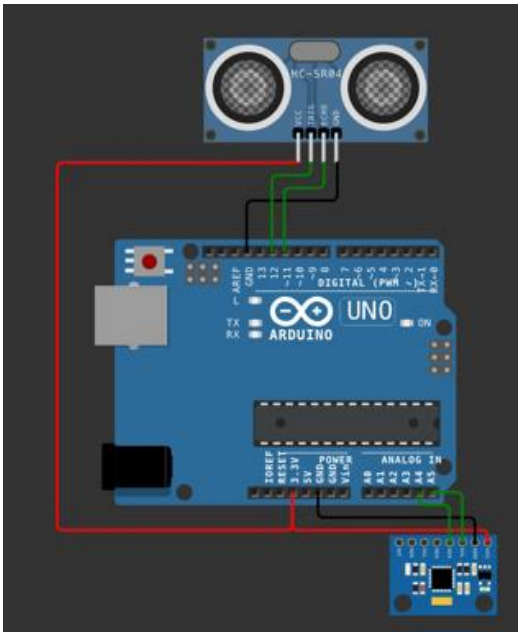

Figure 16 - Gyroscope connection

In these three pictures above which unfortunately can't be combined because of lack of the components that each CAD persist of, but all the connections will be mentioned in this description.

The Ultrasonic sensor HC-SR04 has 4 pins: ground, VCC beside that two Echo and Trigger also which are connecting to the PWM pins of Arduino.

The next sensor that will be used is Pololu QTR-8RC which is not represented here because it is not existing in the digital way, the difference of this one comparing it to the simple IR that I represented is that Pololu is an array that has 8 IR sensors build-in so it is easier to operate an it will be connected also to same ground and power as Ultrasonic and to the other 8 pins of Arduino.

The last sensor represented is MPU 6050 Gyroscope and Accelerometer is a Micro Electro-Mechanical Systems (**MEMS**) which consists of a 3-axis Accelerometer and 3-axis Gyroscope inside it. This helps us to measure acceleration, velocity, orientation, displacement and many other motions related parameter of

a system or object. Vcc Provides power for the module, can be +3V to +5V. Typically +5V is used, Ground Connected to Ground of system, Serial Clock (SCL) Used for providing clock pulse for I2C Communication, Serial Data (SDA) Used for transferring Data through I2C communication, Auxiliary Serial Data (XDA) Can be used to interface other I2C modules with MPU6050. It is optional, Auxiliary Serial Clock (XCL) Can be used to interface other I2C modules with MPU6050. It is optional, AD0 If more than one MPU6050 is used a single MCU, then this pin can be used to vary the address and Interrupt (INT) to indicate that data is available for MCU to read. From this bid diversity of pin, we will use only two of them beside the power and ground is SCL (serial clock) and SDA (serial data) both are connecting to the Analog pins of Arduino.

To sum up the information that we described above, after connecting all the components 13 pins of Arduino will be used.

6 Car body design[4]

6.1 Project Overview

Designing a car shield was an indeed interesting part of the project. As a team full of ACS students, we had never dealt with 3D modelling and overall design before.

6.2 Initial Design

The starting point was a simple and non-functional 3D model created in Tinker CAD (see pic 1.0). This model was of poor quality and could not be printed.

6.3 Improved Design

The second design was more promising compared to the previous one as it was made in Blender, which offers wider possibilities for 3D modelling. After consulting with FabLab, we redesigned our car to resemble a Tesla Cybertruck for better printing accuracy due to its strict geometrical forms (see pic 1.1).

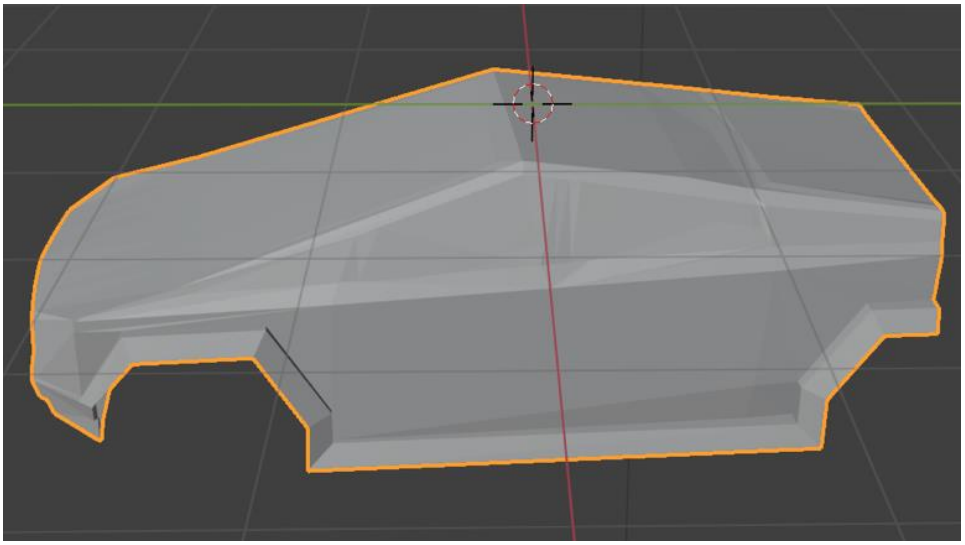

Figure 17 - CyberTruck car body design

6.4 Printing Challenges

Unfortunately, the 3D printer failed to print the model five times in a row. Additionally, the time required to print the model was prohibitive.

6.5 Alternative Approach

Due to these challenges, we decided to abandon the 3D printing idea. Instead, we opted to cut our car body on a laser cutter. This method was ten times faster and cheaper than 3D printing and allowed us to cut the body as many times as needed to perfect our design (see pic 6.2).

Figure 18 - Alternative design

7 Software testing

It is important to mention, that a total of three main functionalities are completed for this project and a lot of testing/variations were changed. Some of these changes are worth mentioning to clue in the logic of main program and explain some essential parts.

7.1 Line Tracking[5]

```
//general function for the line tracking that is continuously used in a main loop //to check for white line and update the position of the wheels.
void trackLine() {
 if (isFullyOnWhite()) {
  MOTOR_SPEED = STOP_SPEED;
  runMotor(MOTOR_SPEED);
  Serial.println("FULL STOP");
 }
 float servoangle = (float)position / 7000;
 servoangle = servoangle * 90;
 servoangle = 45 + servoangle;
 myservo.write(servoangle);
}
```

Figure Code 1.1 – track line function

Essentially the final version of this function calculates a “servoangle” variable that is a mapped value to a 90 angle range of motion of given servo motor that rotates wheels. To properly map this value we firstly use In-Built QTR8-RC library function readLineWhite() that takes an array to write current sensor value to and returns a value that is representing a position of white line relatively to the sensor (range: from 0 to number of sensors –1 * 1000. In our case from to 7000 where 0 means that line is on the very left part, and 7000 means that line is on very right part of QTR). This value is then divided by 1000 to be put in a range from 0 to 1 which is then easy to map to a working range of servo (90 degrees) by multiplying by 90. Finally a value of 45 is added because in the car’s setup servo that is turned to 90 degree angle means straight wheels, 45 – max right turn, 135 – max left turn.
This method provides really efficient line tracking that has little to no “extra” turns that make the car go unstable.

Worth mentioning that option of calculating two halves of sensor values and comparing their sum, difference to proceed with different states for different value ranges. Nonetheless, it was concluded that more generic and laconic algorithm has to be created.

```
void trackLine() {
  SumLeft =(sensorValues[0] + sensorValues[1] + sensorValues[2] + sensorValues[3]);
  SumRight = (sensorValues[4] + sensorValues[5] + sensorValues[6] + sensorValues[7]);
  int SumDifference = (SumLeft - SumRight);
  int totalSum = SumLeft + SumRight;
  else if(abs(SumDifference) < 500) dir = 0;
  else if(SumDifference < -500)dir = 1;
  else if(SumDifference > 500)dir = -1;

  switch (dir) {
    case 0:
      myservo.write(90);
      Serial.println("forward!");
      break;
    case 1:
      myservo.write(45);
      Serial.println("right!");
      break;
    case -1:
      myservo.write(135);
      Serial.println("left!");
      break;
```

Figure Code 1.2 - Alternative Trackline() function

7.1.1 IsFullyOnWhite()

This function provides the program with ability to differ a situation when the full white line is present under an IR which means that stop point has been reached. Each sensor value is being compared to predefined threshold to see if every value is in “white” range. Additionally, function was provided with a parallel processing duration check that guarantees that program considers white line really seen and stops program only when IR has been seeing this line for more than 25 milliseconds. This really helped to avoid any data faults or wrong readings that could accidentally stop the program.

7.1.2 *P.I.D Control System alternative [9]

Another Interesting method of detecting and following a line is P.I.D. controller. This method was taken in a deep consideration and was rigorously tested to realise the difference in various options for line tracking.

A PID (Proportional-Integral-Derivative) controller for a line-following robot helps the robot stay on a path by adjusting its steering based on three components:

- Proportional (P): This component corrects the robot's steering in proportion to the current error, which is the distance from the line. A larger error results in a larger correction.
  Formula: $P=K_p \times error$
- Integral (I): This component corrects the steering based on the cumulative sum of all past errors. It helps eliminate residual errors that the proportional component might miss.
  Formula: $I=K_i \times \sum error$
- Derivative (D): This component corrects the steering based on the rate at which the error is changing. It helps predict future errors and dampens the response to avoid overshooting. Formula: $D=K_d \times (d(error)/dt)$

How this works: Sensors are used (in our case infrared sensors) to detect its position relative to the line. Error is then calculated as the difference between the desired position (on the line) and the actual sensor readings. The PID controller processes the error through the P, I, and D components to compute a control signal. This control signal is sent to the robot's motors, adjusting their speed or direction to reduce the error and keep the robot on the line. By continuously adjusting the control signal based on the error, the PID controller ensures the robot follows the line accurately and smoothly.

To summarize : this is a good method that ensures high precision while building on its own mistakes, but it was decided that P.I.D. would be an overkill for this project purpose. Additionally it is hard to code and integrate a proper algorithm like this without previous experience.

7.2 Obstacle Detection[6]

Functionality of algorithm that evades an obstacle is rather straight forward. Due to limitations of chosen ultra-sound sensors and other reasons it is mostly predefined.

```
void evadeObstacle() {
  myservo.write(90); // even out wheels
  runMotor(0); //stop the motor to negate previous inertia
  delay(150);
  runMotor(-DEFAULT_SPEED); //go backwards
  Serial.println("back");
  delay(1000);
  runMotor(0); //stop the motor to negate previous inertia
  delay(150);
  myservo.write(45); //turn wheels to the right to start going around object
  runMotor(DEFAULT_SPEED);
  Serial.println("Right");
  delay(1500);
  myservo.write(135); //turn wheels to the left to even out the position of
          //the car relatively to the parallel where the object is present
```

```
  Serial.println("Left");
  delay(2000);
  myservo.write(90); //even out wheels and go backwards to balance position out     //to enter the line with
more advantageous angle
  runMotor(-DEFAULT_SPEED);
  Serial.println("back");
  delay(800);
  runMotor(0); //stop the motor to negate previous inertia
  delay(150);
  runMotor(DEFAULT_SPEED - 2); //start going forward but little bit slower
// to ensure finding line
  Serial.println("left to line");
  position = qtr.readLineWhite(sensorValues);
  myservo.write(100);
  while (isFullyOnBlack()) { //go to the left untill sensors see any white(line)
    position = qtr.readLineWhite(sensorValues); //update values
  }
//line is found
  runMotor(STOP_SPEED); // stop the motor to negate previous inertia
  delay(400);
  position = qtr.readLineWhite(sensorValues); //update the sensor values
  trackLine(); //we are in a function so trackline currently is not being called  //repeatedly inside main loop as
we are not here so update wheel angle.
  runMotor(DEFAULT_SPEED); // go forward, end object evasion.
}
```

Figure Code 1.3 – evadeObstacle() function

IsFullyOnBlack() works similarly to IsFullyOnWhite(), but does not wait for values to remain in same range for given time period because defining black color is being done with little to no errors.

7.2.1 Calculating Distance()

The thing that works as a main gatekeeper before execution of object avoidance is an if statement that calls CalcDistance() function and compares its values so they fit within certain range in order to proceed further.

```
float calcDistance(int echo, int trig) {
  digitalWrite(trig, LOW);
  delayMicroseconds(2);
  digitalWrite(trig, HIGH);//emit ultra sound wave
  delayMicroseconds(10);
  digitalWrite(trig, LOW);
  float duration = pulseIn(echo, HIGH); //receive the duration of a reflected ultra sound wave
  float distance = (duration * 0.0343) / 2; //The speed of sound is approximately 343 meters per second,
```

```
  //or 0.0343 centimeters per microsecond. Multiply by it to conver to cm.
  return distance; //Finally divide by 2 as it is the distance that sound wave took traveling both directions.
}
```

Figure Code 1.4 CalcDistance() function

7.2.2 Alternatives

An attempt of writing more generic code that could operate with any type of obstacle was made but did not succeed. Temporarily a second ultra-sound sensor was added to the side of REVO RC car in order to execute following **algorithm**:
1) when object is detected, go to the right as long as side sensor starts seeing any obstacle and then passes it.
2) balance out the position of car
3) go forward until side sensor stops seeing presence of possible obstacle that would have to be in parallel at this moment.
4) sensor stops seeing an obstacle which means it has been evaded
5) turn back to the line until it is spotted.

After lots of testing it was concluded that principle of usage of the HC SR04 do not meet needs of this purpose because they literally cannot spot tilted surfaces (which might be good for combined usage with ramp overcoming). The sound wave simply gets reflected and does not end up getting caught by the sensor itself to measure distance.

So, all in all this method would still highly rely on the form of an obstacle and it was not reasonable to finally consider this option.

7.3 Motor interaction

It is essential to note how the motor is being used within the program.

All throughout the code function runMotor(int speed) moves the car. Logic is really simple. There are two main pins that control the car (LPWM, RPWM) first one for driving forwards, second for driving backwards. Also, global values as DEFAULT_SPEED, RAMP_SPEED, STOP_SPEED are being used to pass in this function to ensure code readability, along with refactoring. It is easier to change value in 1 place instead of 10 places.

```
void runMotor(int speed) {
  if (speed >= 0) { //if speed >= 0 moving forward
    digitalWrite(RPWM_PIN, LOW); //turn off backward direction
    analogWrite(LPWM_PIN, speed); //set the speed to forward direction pin
  }
  else { //if speed < 0 moving backwards
    digitalWrite(LPWM_PIN, LOW);//turn off forward direction
    analogWrite(RPWM_PIN, -speed);//set the speed to backward direction pin
  }
  Serial.print(speed);
}
```

Figure Code 1.5 – RunMotor() function

7.3.1 Remark

Extremely important to mention that these pins cannot except a negative value, if they do – they set this pin value to its maximum. Even if you want to pass –1 to LPWM it will consider it as maximum value and cause a lot of trouble.

7.4 Ramp detection algorithm[10]

7.4.1 Background

The ramp detection algorithm was developed to enhance the vehicle's autonomous capabilities. Initially, the approach to this problem was uncertain because it was anticipated that the ultrasonic sensor, mounted at the front of the vehicle for obstacle detection, would mistakenly identify ramps as obstacles, thereby triggering the evadeObstacle() function, which is not the desired behavior. To address this, a preliminary solution involved deactivating the ultrasonic sensor after detecting the first obstacle. However, this approach was not generic and was quickly abandoned.

Surprisingly, the ultrasonic sensor did not detect the ramp as an obstacle due to the ramp's curved surface, which prevented the ultrasonic waves from reflecting back to the sensor. This observation led to the implementation of a ramp detection algorithm using the MPU6050 gyroscope, mounted on the vehicle's base frame for stability.

7.4.2 Angle calculation

```
// Function to calculate the angle based on MPU6050 data
void calcAngle() {
  ax = mpu.getAccelerationX();
  ax = map(ax, -17000, 17000, 0, 255);  // X axis data
}
```

Figure Code 1.6 – calcAngle() function

Using the MPU6050 library, the calcAngle() function was created and called continuously within the main loop, constantly updating the ax values. These values ranged from 30 to 150, depending on the incline angle of the vehicle. Analysis of these values led to the conclusion that the vehicle should operate within three ranges, each corresponding to a specific state:

1. Ramp detected – ax value exceeds 130.
2. Vehicle on flat surface – ax values between 95 and 130.
3. Vehicle moving downhill – ax value below 95.

Based on the detected state, the vehicle's speed was adjusted accordingly.

7.4.3 Ramp detection handling

Initially, the algorithm considered only two states: values above 130 and values below 130. This simplistic approach failed because the vehicle would accelerate up the ramp, and upon reaching a flat surface, it would exit the loop and resume line tracking (trackLine()), leading to high-speed deviations off the track. Thus, the algorithm was revised to incorporate three states to manage the vehicle's speed while descending the ramp. Important note is that trackLine() function is turned off while the car is on the ramp.

```
void handleRampDetection() {
  if (checkRamp()) {
```

```
int state = 0;
myservo.write(90);

while (state != 3) {
 if (state == 0) {
  runMotor(RAMP_SPEED);
  startTime = 0;
  while (state == 0) {
   calcAngle();
   Serial.print(state);
   Serial.print("\t");
   Serial.print(ax);
   Serial.println();
   if (checkForward()) {
    Serial.println("CHANGE TO STATE 1");
    state = 1;
   }
  }
 }

 if (state == 1) {
  startTime = 0;
  runMotor(DEFAULT_SPEED - 10);
  while (state == 1) {
   calcAngle();
   Serial.print(state);
   Serial.print("\t");
   Serial.print(ax);
   Serial.println();
   if (checkDown()) {
    state = 2;
   }
  }
 }

 if (state == 2) {
  runMotorBackwards();
  startTime = 0;
  while (state == 2) {
   calcAngle();
   Serial.print(state);
   Serial.print("\t");
   Serial.print(ax);
   Serial.println();
   if (checkForward()) {
    state = 3;
   }
  }
 }
```

```
    if (state == 3) {
      MOTOR_SPEED = DEFAULT_SPEED;
      runMotor(MOTOR_SPEED);
    }
   }
  }
}
```

Figure Code 1.7 – handleRampDetection() function

7.4.4 Debugging and refinement

To refine state detection, a state machine was initially tested within the loop, but this led to erratic behavior, such as detecting ramps on minor elevation changes. Increasing the detection threshold from 130 to 150 resolved some issues but required higher initial speeds, which introduced new problems. The successful solution involved tracking the duration the vehicle remained in a particular state. For instance, to initiate speed increase, the vehicle needed to stay on the ramp for 150ms before transitioning to state 1, indicating the vehicle was on the ramp and needed more speed.

```
// Function to check if the vehicle is moving down a ramp
bool checkDown() {
 unsigned long currentTime = millis();  // Get the current time in milliseconds

 if (ax < 95) {
  if (startTime == 0) {
   startTime = currentTime;  // Record the time when the threshold is first exceeded
  }
  if (currentTime - startTime >= 10) {
   return true;  // Execute the desired code if the condition is met (>10 ms)
  }
 } else {
  startTime = 0;  // Reset the start time if the variable drops below the threshold
 }
 return false;
}

// Function to check if the vehicle is moving forward on a ramp
bool checkForward() {
 unsigned long currentTime = millis();  // Get the current time in milliseconds

 if (ax > 95 && ax < 140) {
  if (startTime == 0) {
   startTime = currentTime;  // Record the time when the threshold is first exceeded
  }
  if (currentTime - startTime >= 35) {
   return true;  // Execute the desired code if the condition is met
  }
 } else {
  startTime = 0;  // Reset the start time if the variable drops below the threshold
 }
```

```
  return false;
}

// Function to check if the vehicle is on a ramp
bool checkRamp() {
  unsigned long currentTime = millis();  // Get the current time in milliseconds

  if (ax > 140) {
    if (startTime == 0) {
      startTime = currentTime;  // Record the time when the threshold is first exceeded
    }
    if (currentTime - startTime >= RampDuration) {
      return true;  // Execute the desired code if the condition is met (>150ms)
    }
  } else {
    startTime = 0;  // Reset the start time if the variable drops below the threshold
  }
  return false;
}
```

Figure Code 1.8 - State Check Functions for Ramp Detection

7.4.5 Handling line tracking after the ramp

Finally, ensuring the vehicle's return to the white line after descending the ramp involved adjusting the wheels to turn downward when moving downhill. Gravity still pushed the vehicle, but the backward wheel rotation reduced speed, giving the vehicle enough time to locate the line.

```
// Function to stop the motor
void runMotorBackwards() {
  digitalWrite(LPWM_PIN, LOW);
  analogWrite(RPWM_PIN, 20);
}
```

Figure Code 1.9 – RunMotorBackwards() function

This algorithm successfully integrated ramp detection and speed adjustment, allowing the autonomous vehicle to navigate ramps effectively and maintain its course.

8 Recommendations

I would like to emphasize that each member of our team did an excellent job, with everyone giving their best from start to finish. Every team member actively participated in discussions and brainstorming sessions, as well as in assembling and testing the machine.

Regarding our product, I would like to highlight areas for improvement. Firstly, instead of a motor driver, it would be better to use an ESC to read the RPM of the motor. This would allow us to maintain a consistent speed regardless of the battery charge. Additionally, using the ESC would enable us to lock the wheels for a smoother descent from the ramp.

Secondly, IR sensors are highly dependent on external lighting. Therefore, it would be beneficial to consider a wider range of coverage for the observed white line area, which would enhance accuracy and reliability under various lighting conditions.

We also encountered challenges with 3D printing, which prevented us from creating the initial body of the machine as intended. Thus, it is recommended that ASC students, designers, and electrical engineers should not be the only ones working on the project.

Despite these challenges, we significantly improved our skills in these areas and adhered to a well-structured plan. This allowed us to meet the client's objectives a week ahead of the deadline, giving us additional time to further enhance our product.

9 Conclusion

The REVO autonomous car project aimed to design, build, and optimize a self-guiding vehicle capable of navigating a predetermined path with precision and efficiency. The project set out to meet six critical objectives: riding autonomously, operating for at least 30 minutes, following a track line, stopping at the end of the track, avoiding obstacles, and locating the track line.

The outcome of the project indicates a significant degree of success in achieving these objectives:

Ride Autonomously: The project succeeded in developing a program that effectively guides the car using sensor feedback, as detailed in the functional design overview and software testing sections. The integration of Arduino and various sensors enabled autonomous navigation.

Work for at Least 30 Minutes: The vehicle met the performance standards for electrical features, ensuring it could operate for a minimum of 30 minutes. This was confirmed through calculations and measurements in the electrical system layout and power consumption analysis sections.

Follow the Track Line (20mm): The implementation of infrared sensors for line detection proved effective in guiding the car along the track. The software algorithms for line tracking were tested and refined to ensure accurate following of the track line, as documented in the software section.

Stop at the End of the Track: The car's ability to detect the end of the track and halt was successfully implemented using the IsFullyOnWhite() function, which provided reliable detection of the track's end, as described in the software testing section.

Avoid Obstacles: The integration of ultrasonic sensors allowed the car to detect and navigate around obstacles. The evadeObstacle() function demonstrated the vehicle's capability to avoid collisions and continue along its path, as shown in the software section.

Find the Track Line: Using a combination of sensors, the car was able to locate and follow the track line consistently. The use of the QTR-8RC infrared sensor array facilitated effective line tracking, as outlined in the electronic connections and software testing sections.

In conclusion, therefore, the main objectives of REVO are that it proved to be a working example of an automatic driving car capable of performing the intended tasks. The systematic approach adopted, comprehensive testing, and iterative advancements were all bound to make this project successful. Proof of documents and result

confirmation, therefore, will indicate that, within the specified objectives, properly workable projects can actually be achieved.

10 References

[1] "Battery energy consumption calculator." Accessed: Apr. 11, 2024. [Online]. Available: http://lculator.com/other/battery-life

[2] D. Lee, "Velocity-time graphs," Online Learning College. Accessed: Apr. 11, 2024. [Online]. Available: https://online-learning-college.com/knowledge-hub/gcses/gcse-maths-help/velocity-time-graphs/

[3] "System Architecture for Autonomous Vehicles." Accessed: Apr. 11, 2024. [Online]. Available: https://encyclopedia.pub/entry/8473

[4] "3D Model of car parts." Accessed: Apr. 11, 2024. [Online]. Available: http://www.mjxrc.net/goodshow/14210-14210.html

[5] Pololu Corporation, "QTR-8A and QTR-8RC Reflectance Sensor Array User's Guide." 2014. Accessed: Apr. 11, 2024. [Online]. Available: https://www.pololu.com/docs/pdf/0J12/QTR-8x.pdf

[6] Sparkfun, "Ultrasonic Ranging Module HC - SR04 datasheet." Sparkfun. Accessed: Apr. 11, 2024. [Online]. Available: https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf

[7] "Rc 540 motor datasheet." Accessed: Apr. 11, 2024. [Online]. Available: https://asset.conrad.com/media10/add/160267/c1/-/en/001385115DS01/adatlap-1385115-540-es-motor-reely-532114c.pdf

[8] "Programming | Arduino Documentation." Accessed: Apr. 11, 2024. [Online]. Available: https://docs.arduino.cc/programming/

[9] “Line Follower Robot (with PID controller),” projecthub.arduino.cc. https://projecthub.arduino.cc/anova9347/line-follower-robot-with-pid-controller-01813f

[10] InvenSense Inc., “MPU-6000 and MPU-6050 Product Specification Revision 3.4 MPU-6000/MPU-6050 Product Specification,” Aug. 2013.

Available: https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf