2E10 Final Report

Group X9

Contents

Introduction

The goal of the 2E10 project was to create a buggy capable of a certain level of autonomous driving. The overall module was split into 3 challenges:

- Bronze Challenge
- Silver Challenge
- Gold Challenge

The bronze challenge required us to a properly wire our buggy as well as use the ultrasonic sensors and the IR sensors to follow a line and stop at an appropriate distance from an obstacle. It also required us to connect our buggy to a control laptop over WiFi and to create a GUI in Processing that would receive data from the buggy and send commands to it. The silver challenge took this concept one step further and tasked us with implementing a PID controller to allow the buggy to follow an object at a safe distance. The gold challenge was more open ended and only required that we make use of the IMU, everything else was left to our imagination. However due to unfortunate circumstances, our group could not attempt the gold challenge. But bronze and silver were attempted and completed.

Self-Assessment

Throughout the entire semester we worked well together as a team. We had many strengths that helped us achieve our end goal, the main ones being:

- **Diversity of Skill**

  Everyone in the team came in with their own unique skills which greatly aid in the development of the buggy. The project manager Aaron Dinesh was skilled in both Arduino and Processing and so he wrote all the code for the bronze and silver challenges. Laura Noble having had previous experiencing creating videos, filmed and produced the weekly videos and helped considerably in the various reports and written assignments that we had to hand up through the semester. Oliver Smyth and Siddharth Lala were also instrumental in testing the code and troubleshooting any issues that arose.

- **Diversity of Ideas**

  Everyone also pitched in, giving ideas and solutions as to how we could tackle all the requirements of the various challenges. This meant that very early on we had a clear idea of everything that we needed to do and how we would implement our solutions. This also came in very handy when any issues would arise. Since everyone approached the problem in a different way, we were able to quickly diagnose and fix any issues that came up.

- **Organization**

  We were also highly organized from the start. Aaron took on the responsibility of creating a work breakdown structure, that outlined all the tasks that needed to be completed to have a successful bronze and silver demo. We also laid out contingency plans, in the unlikely event that one of our team members would fall ill. This would prove vital later in the semester.

However, no group is without faults, and we were no exception. Our 3 main weaknesses would be:

- **Communication**

  We had set up a WhatsApp group chat to allow us to communicate details of the project outside of the designated lab times. Even so, it was not used as much as it should have been. This meant that oft times we didn’t know if members of the group were going to turn up to the lab sessions and some didn’t know what work had been done over the past week.

- **Scheduling meetings**

  Trying to schedule meetings outside of the assigned lab slots was also an area we were deficient in. We make the mistake of thinking that the two lab slots would be enough for us to cover everything that needed to be said and to plan out the week ahead. However, with certain members not showing up to the lab sessions, it was clear that we needed to schedule more meetings throughout the week to make up for this.

- **Only one proficient coder**

  One other big weakness of the team was that Aaron was the only coder in the team, so all the coding and wiring tasks fell on him. While, he was fully proficient in the Arduino language and Processing, it would have greatly helped if there was another member of the team that knew how to code. This caused major issues when he fractured his leg and couldn’t attend college for 5 weeks of the semester. To work around this, an online meeting was set up where he could join the lab session and remotely code the buggy. He would email the code to us, and we would upload it to the buggy. This process was slow and wasn’t ideal, but it was the best we could do given the circumstances. It was for this reason we couldn’t progress to the gold challenge. However, if we had a second team member who was willing to write the code, this could have been avoided.

To Achieve Gold – 3 Changes We Would Make

While we were very happy with achieving silver, we all still wanted to go for gold. Though some circumstances were unavoidable, there are some changes we could make that would allow us to do the gold challenge, given 2 extra weeks.

1. **Schedule more meetings**

   It was evident from the semester gone by that more meetings were needed through the week, especially since some members did not turn up to the assigned lab slots. Having more sessions would mean that we would be able to catch them up with all the things they had missed from the previous lab session.

2. **Have more members help with the coding**

   While Aaron got all the coding for the bronze and silver challenges done. With the sheer amount of coding and bug-fixing required, he just managed to get the code done in time. If we were given 2 weeks to do the gold challenge, we would split up the coding tasks between the team members and learn the coding required to finish the tasks.

3. **Clean up the wiring**

   When wiring up the buggy, we were more concern about function over form. This meant that our wiring ended up being messier than it should have been. While it was good in the short term for getting the buggy running, it meant that if any issues arose with the wiring it was hard to trace the cause. We would also have issues with some wires disconnecting themselves during transportation. Due to the messy wiring, it meant that fixing required some time and took away from the time we could have spent coding the buggy. For gold we would make the wiring cleaner and easier to follow. We would also secure the wiring with some tape to avoid the issues of wires disconnecting themselves during transport.

The Ethical Concern

This project raises a number of ethical concerns. Not only with the issue of whether we should have self-driving cars or not, but also in code validation, hardware use and misuse and various other areas.

Code Validation

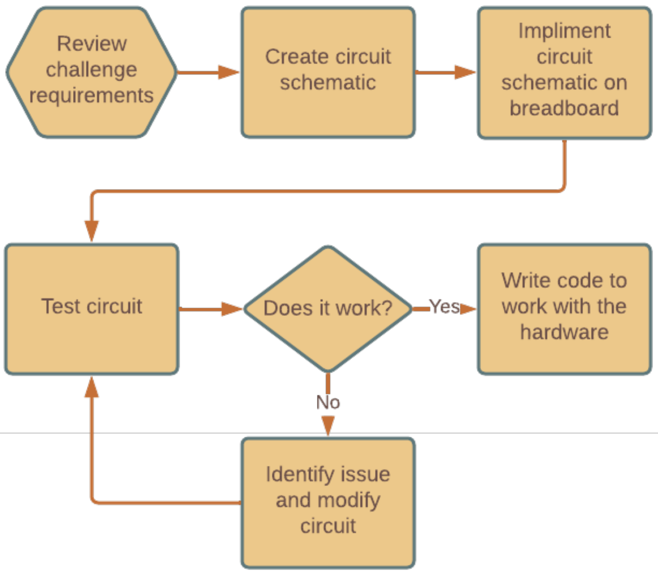
With our buggy, the code is relatively simple and could be understood well by an experienced coder. But with the rise of self-driving cars the code base required to make these cars function is massive. Take for example Tesla, one estimate says that a car from Tesla contains over 150 million lines of code (1). This is simply unfeasible for anyone to understand fully. Also, since they use libraries and code written by others, they are relying on them to keep their code up to date and free from any security issues. One famous example of this is the recent Log4J vulnerability. This was an issue in a piece of open-source software provided by the Apache Software Foundation that allows applications to log errors and record events. However due to an oversight on the part of the coders, any attacker could gain remote code execution and could steal plenty of sensitive information from applications such as usernames and passwords, among other things. Due to Log4J being an extremely popular library, it meant that a wide variety of applications fell prey to attacks from malicious parties (2). What if such a vulnerability were found in Tesla's software? Gaining remote code execution on a car that runs on software is many times more dangerous. While the chances of such an event happening are low, they are never zero. Tesla also make use of neural networks to make their self-driving car dream a reality. These neural networks are essentially black boxes, inputs come in and the car makes decision based on the output. But due to the black box nature of these networks, no one can know for sure how the data gets processed internally. Using a large training and validation set we can minimize the unpredictability of the car, but we can never make it zero. And even worse, when presented with a new situation that the network hasn't trained on, we cannot be sure what the car will do. Take for example a modification of the famous Trolly Problem. The problem is stated as such, "There is an old lady and a baby walking in the path of an on coming car. Should the car avoid the baby and kill the old lady, or should the car avoid the old lady and kill the baby?" While this might be a hypothetical scenario, similar situations may arise, and it is vital to know what the car might do in this situation.

Hardware Use and Misuse

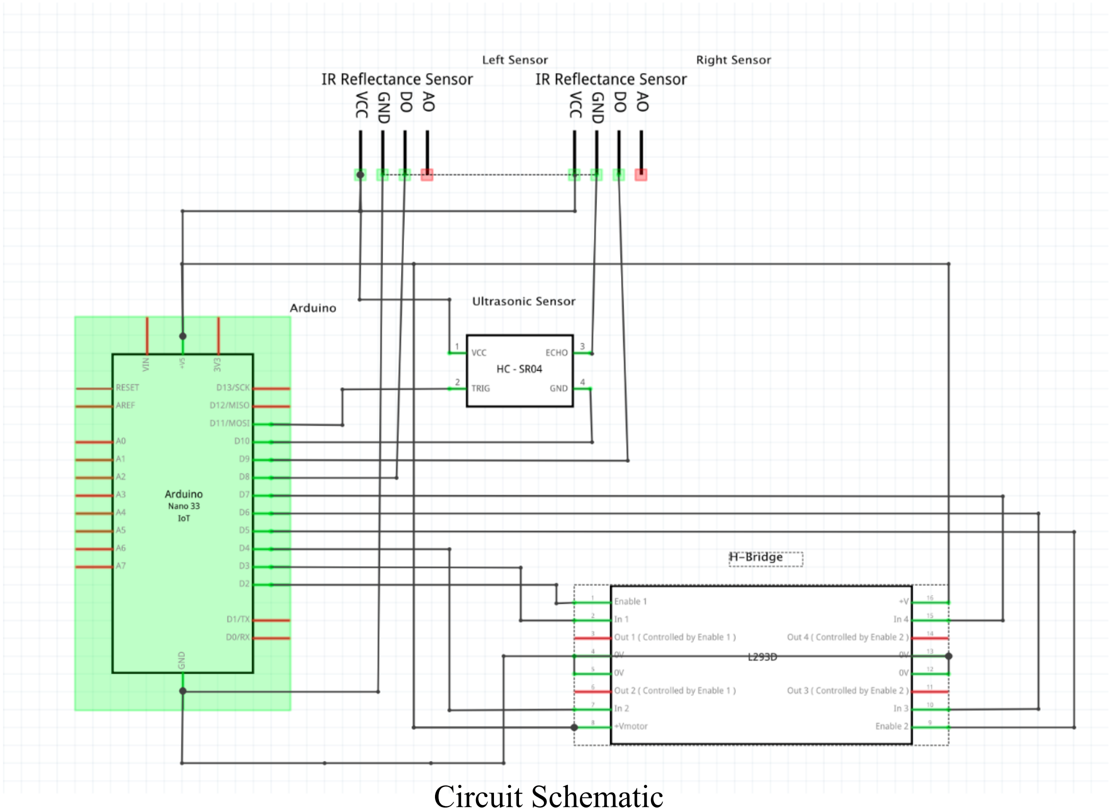
Self-driving cars also need an array of cameras and sensors to work properly and the output from these sensors get logged. The issue arises when this data is misused. The Netherlands Forensics Institute managed to reverse engineer the proprietary file storage and found that Tesla logged data such as steering angle, acceleration, autopilot use and much more (3). If a determined attacker was able to get access to such data by abusing a vulnerability in Tesla's software, it could allow them to track the car in real time using the cars inbuilt internet connection. This raises serious concerns about whether companies should be able to log things such as these and if customers should be given an option to opt out of such data logging.

Hardware Design

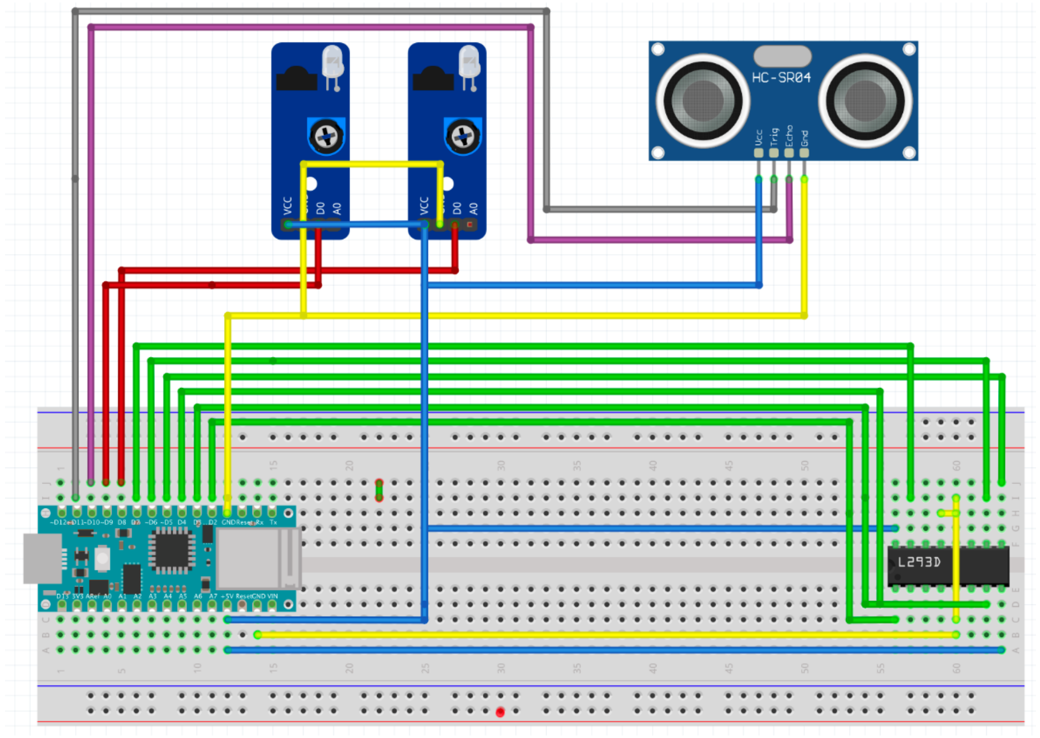
Wiring the buggy was the first of the many milestones we needed to accomplish on our way to the silver challenge. It certainly was no easy task since all we were given were a handful of electronic components and a breadboard for prototyping. Our hardware design flow can be seen from the flow chart below outlining our steps:


Review challenge requirements
Create circuit schematic
Impliment circuit schematic on breadboard
Test circuit
Does it work?
Yes
Write code to work with the hardware
No
Identify issue and modify circuit

Through this iterative design process, we came up with the following circuit diagram and schematic



Circuit Schematic



Fritzing Diagram

Motors and Sensors

This circuit is built around the Arduino Nano 33 IOT. This particular microcontroller was chosen because of its support for Wi-Fi and its onboard IMU. The Wi-Fi support was imperative for the bronze and silver challenges as it allowed us to send and receive data from the microcontroller. This will be explained in further detail in the software section of the report. To control the wheels, two standard DC motors that had a 5V operating voltage were used. However, none of the outputs on the Arduino were able to source enough current or voltage to be able to drive these motors. Instead, the L293D quadruple half H-bridge chip was used as an intermediary. Two of the four half H-bridges were combined to make one full H-bridge which would connect the motor to the battery. Then depending on which pins of the IC were driven high by the Arduino, the corresponding motor would be driven either forwards or backwards. The outputs follow the truth table listed below:

Pins On H-Bridge				
2	7	10	15	Output
L	L	L	L	Nothing Happens
H	L	L	H	Move Forwards / Differential Steering
L	H	H	L	Move Backwards / Differential Steering
H	L	H	L	Turn Clockwise
L	H	L	H	Turn Anticlockwise

The left and right enables pins (pin 1 and 9 on the h-bridge respectively) were used to turn on the left and right motors. By writing a PWM signal to these pins we could also vary the speed of each motor. By choosing different speeds for the left and right motors we could repurpose the move forward command as a differential steering command. While differential steering would provide a smoother driving experience, it required a lot of fine tuning to get right. It was decided that differential steering was not worth the added effort, so we opted to go for on the spot steering.

In order to sense the line, we made use of two TCRT5000 reflective optical sensors. Operating in 950nm wavelength it was particularly useful for detecting the white reflective tape that made up the line. This paired with the accompanying support PCB made it very easy to integrate into our design. The support PCB automatically stepped down the input 3.3V to the 1.5V operating voltage of the sensor. It also has a thresholding feature that turns on the digital output on the PCB once a certain threshold is reached. This threshold can be changed by turning the potentiometer on the PCB. It also has an analog output that can be used with the Arduino. Ultimately, we settled on using the digital output of sensor as it allowed for accurate line sensing without complicating the supporting code. The digital output from the sensor was fed into pin 9 and pin 8 of the Arduino for the left and right sensors respectively.

The last sensor we required for full functionality was the HC-SR04 ultrasonic sensor. Having an operating voltage range from 3V to 5.5V, it made it a perfect fit for the 3.3V logic on the Arduino. After connecting the VCC, trigger, echo, and ground pins to the Arduino, we were ready to write the code to interface with the ultrasonic. To start the distance measurement the trigger pin needed to be pulsed for $10\mu s$. This would trigger an ultrasonic pulse to be emitted from the module. Once this pulse has been reflected and received by the microphone on the module, the echo pin will go high for the duration of time it took the pulse to be emitted, reflected, and received. This module allowed us to measure distances from 2cm to 450cm with a resolution of 30mm (4).

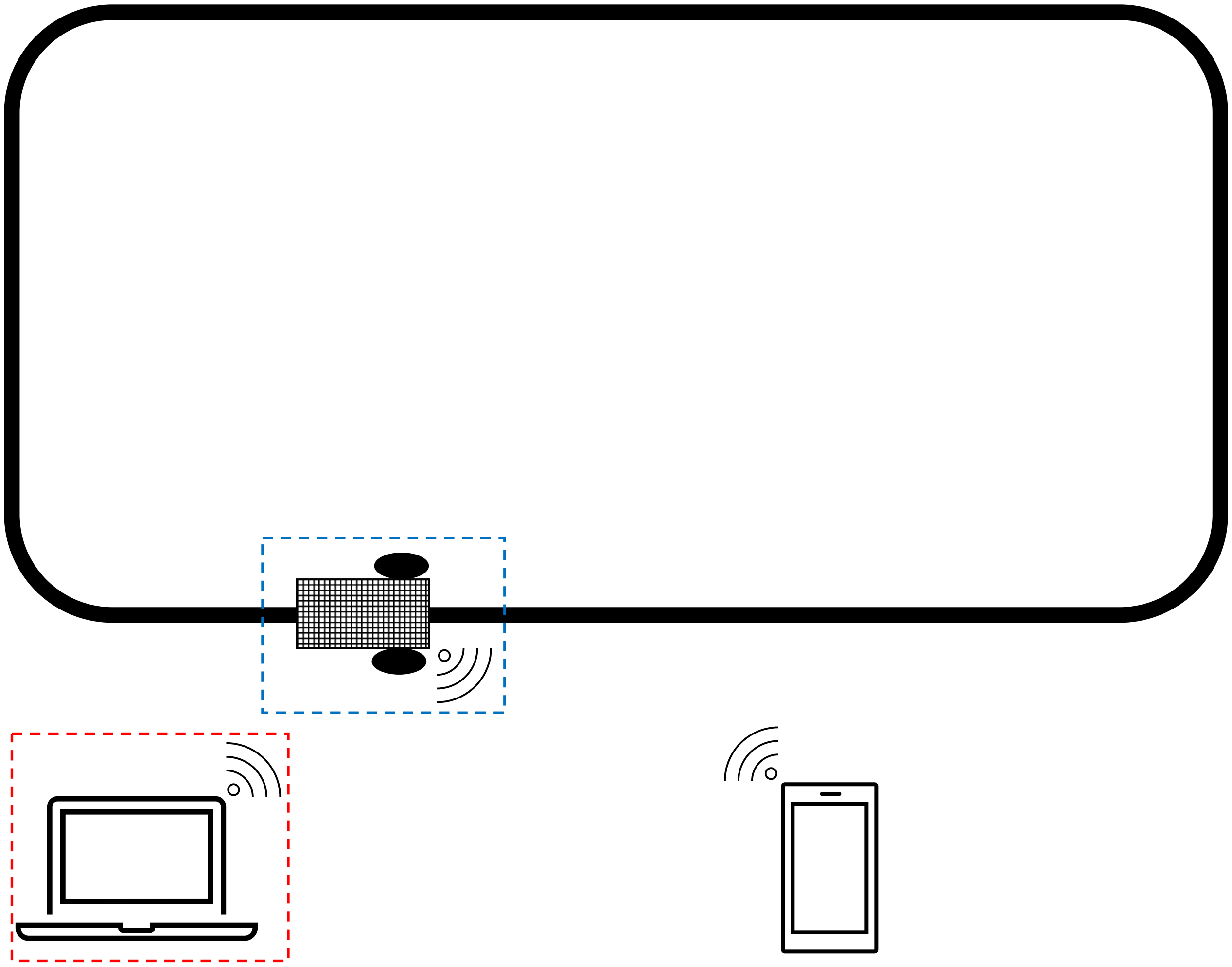
Hardware Challenges

The main challenge with getting the hardware running was understanding how the H-bridge works. As none of us had used a H-bridge before, we needed to investigate how it operates. However, after reading the datasheet for the IC we were able to develop an idea of how the H-bridge operated and how we could use its input pins (2, 7, 10, 15) to control the direction of the motors. Once we got over this hurdle, the rest of the hardware implementation was trivial, only needing a power, ground, and Arduino connections. We also had to make sure we kept the 5V and 3.3V lines separate. While this wouldn't be an issue for the ultrasonic sensor since it can work in any voltage range from 3V to 5.5V, it would prove problematic for the H-bridge and the IR sensors if these lines were combined. Sending 3.3V to the VCC pin of the H-bridge would lead to the motors not being able to run properly and sending 5V to the IR sensors could cause irreparable damage to its components.

Software Design

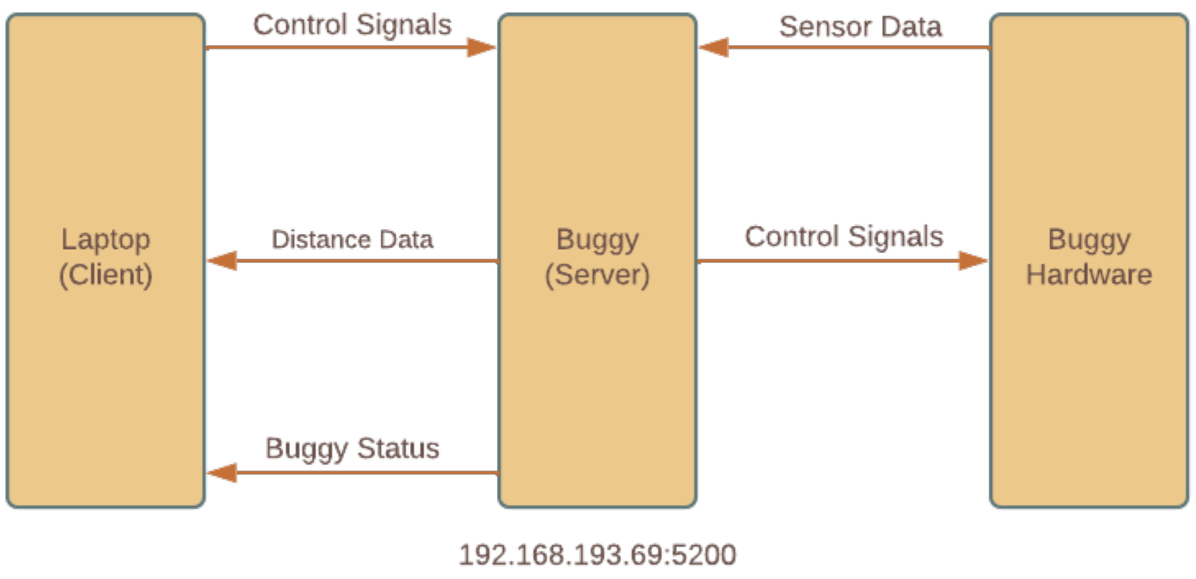
Domain and System Models

When starting the software design part of the project it was useful for us to sketch a domain and a system model. This helped us not only to visualize the flow of data in our system, but it also helped us to identify what the sever and the client would be.



Domain Model Diagram

The Domain model diagram lays out how our laptop (red) and buggy (blue) would interact in the real world. We decided on having the buggy be the central server while the laptop would be the client connecting to the server. We also decided to network everything together using a hotspot created from one of our phones. We opted for this over the network provided in the 2E10 labs as it meant that we could specify a static IP address for the buggy and be sure that it wasn't being used by any other device on the network. This also allowed us to hard code the IP in our processing script, which allowed us to guarantee a successful handshake every time we tried to connect to the buggy.



System Model Diagram

The system model diagram allows us to see the flow of data through the system. The laptop sends either a 'stop' or a 'go' to the buggy depending on user input. Once the buggy receives this control signal it acts accordingly by either starting or stopping the driving control algorithm. Independent of the laptop control signals, the buggy also gathers data from the various sensors and relays this data back to the laptop where it is displayed on the Processing GUI. As well as sensor data the buggy also transmits its status back to the laptop, indicating whether it has stopped due to the stop button being pressed or is there is an obstacle in the way. We didn't make use of any custom data structures in our code as we felt that everything we wanted to achieve could be reached using the inbuilt data structures such as ints, strings, and chars.

Control Code (Arduino)

The control code on the Arduino is a relatively simple algorithm. It first begins, in setup by defining a couple of input and output pins for the sensors and H-bridge. After that it moves onto creating a Wi-Fi server object and connects to a Wi-Fi network of our choosing. This was accomplished through the use of the WiFiNINA library. Next, we move into the loop function. This function run forever, essentially acting like a while true loop. It is in here where the main driving algorithm gets run. Upon receiving a connected client, we check to see if it has successfully connected and has sent data. If both these are true, we read the data that was sent and assign it to the char char variable command. The client in this case is a laptop running the processing script which will be explained further on in the report. Then, every 5 milliseconds we trigger the ultrasonic sensors to measure the distance. Our user defined function measureDistance interfaces with the ultrasonic sensor through the use of the NewPing library. This library was specifically created with the efficient use of the HC-SR04 sensor in mind. Once we have the distance measurement, we pass it to the computePID function, which returns the PID error (PID_err) between the distance measured and our desired distance (setpoint_distance). Then if the command received from the client is stop (denoted by the character s), we stop the buggy. Otherwise, we call drive with PID_err. The drive function first checks if the distance measured is less than setpoint_distance, if this returns true then we stop the buggy. Otherwise, we check to see if the IR sensors have detected the line and follow this truth table:

IR Sensors		
Left	Right	Action Taken
1	0	Turn Left
0	1	Turn Right

If the previous conditions have all failed, then it must mean we are centered on the line and can move forward with PWM speed PID_err. Below the full pseudocode for the Arduino is laid out, with commands in blue, function calls in purple and variables in green.

```
IMPORT NewPing
IMPORT WiFiNINA
IMPORT String

BEGIN setup
        Start the serial port at 9600 baud
        Set the input and output pins
        Set the IP address to be an IP of your choosing
        Connect to the Wi-Fi and begin the server on a port of your choosing
        Print the IP address to serial console
END

BEGIN loop
        CALL server.available RETURNING a connected client

        IF the client is connected THEN
                IF the client has sent data to be read THEN
                        Read the data that was sent
                        SET command to be that data
                        Print command to serial console
                END
        END


        IF 5 milliseconds have passed THEN
                CALL measureDistance RETURNING the distance from the ultrasonic sensor
                SET distance to the value returned by measureDistance
                CALL computePID RETURNING the error calculated using the PID algorithm
                SET PID_err to the value returned by computePID
        END

        IF command is equal to 's' THEN
                CALL stop which stops the buggy
        ELSE
                CALL drive with PID_err and distance
        END

END

BEGIN drive
        Read in value from left IR sensor and SET left_read to be this value
        Read in value from right IR sensor and SET right_read to be this value

        IF distance is less than the setpoint_distance THEN
                CALL stop
        ELSE
                Go forward with PWM speed PID_err
        END

        IF left_read is equal to 1 and right_read is equal to 0 THEN
                Turn left
        ELSE IF left_read is equal to 0 and right_read is equal to 1 THEN
                Turn right
        END

END
```

Control Code (Processing)

The processing code runs on the client PC and its job is to send commands to the Arduino and to receive the distance and a status update from the Arduino. The GUI made use of the ControlP5 library, and the networking was done through the Processing.net library. These two libraries made coding the GUI and the networking extremely easy as much of the low-level coding was abstracted away from us. This allowed us to rapidly prototype various interface designs until we finally converged on the one used in our silver demonstration. It consisted of two ControlP5 buttons, labeled “Stop” and “Go” which would trigger a stop or go command to be sent to the Arduino when clicked. We also wanted to display the data sent back from the Arduino, however the inbuilt “Textfield” didn’t quite work the way we hoped it would and didn’t display our data properly. Instead, we would sometimes get no data being displayed or some data from the previous transmission being displayed over the new data that was sent. For these reasons we opted to make use of two invisible buttons which had their caption label being updated each time a new transmission came in. Below a full pseudocode of our processing sketch can be found, with a similar coloring scheme to the one used in the Arduino pseudocode.

```
IMPORT ControlP5
IMPORT Processing.net

BEGIN setup
        SET screen size to fullscreen (1920 by 1080)
        CREATE new gui object from ControlP5
        CREATE client object from Processing.net
        Connect to client with its IP address and port
        ADD start button to write 'g' to client and then print "Start" to console output
        ADD stop button to write 's' to client and then print "Stop" to console output
        ADD button for distance measurement output
        ADD button for other text output
        LOAD image for background
        SET background to be the loaded image
END


BEGIN draw
        WHILE there is still data to be read from the client
                Read the data
        END

        SET the caption label of the distance output button to be the distance measurement sent
        from Arduino.

        IF the data is equal to 'o' THEN
                SET the caption label of the other text output button to be "Obstacle detected"
        ELSE
                SET the caption label of the other text output button to be "Nothing Found"
        END

        Clear any data that might have been sent before draw runs again

END
```

References

1. r2evans. How many lines of code are in a Tesla? 2022 [updated 06/04/22. Available from: https://code-features.com/q/18847387.

2. Torres-Arias S. What is Log4j? A cybersecurity expert explains the latest internet vulnerability, how bad it is and what's at stake. The Conversation. 2021.

3. DaSilva S. You Can Now Directly Read Data Logs From Tesla Vehicles 2021 [Available from: https://jalopnik.com/you-can-now-directly-read-data-logs-from-tesla-vehicles-1847910795.

4. 4tronix. HC-SR04+ Low-Voltage UltraSonic Distance Sensor [cited 2022 18/04/22]. Available from: https://shop.4tronix.co.uk/products/hc-sr04-low-voltage-ultrasonic-distance-sensor.