Autonomous Car for Pet

Members:

Shuhao Lu, Lingxiao Pan

Advisor:

Professor George Anwar

Problem Statement and Motivation

The inception of this project stems from the indolence displayed by my friend's tiny companion cat, whose lackadaisical attitude renders our attempts to engage it in play utterly futile. Consequently, we embarked on a venture to construct a remotely controllable apparatus with the dual purpose of entertaining our lethargic friend and dispensing delectable treats to our feline and canine comrades. However, we sought to elevate the challenge level for our animal friends in procuring said treats, hence the incorporation of onboard ultrasonic sensors tasked with visually detecting cats and dogs, enabling the vehicle to adeptly evade them. Alternatively, the vehicle can be adroitly maneuvered by the user through a Graphical User Interface (GUI) endowed with intuitive controls. In addition, the GUI shall furnish real-time readings of the ultrasonic distance, providing invaluable information to facilitate navigation. It goes without saying that the vehicle shall operate on a reliable power source, namely onboard batteries, ensuring uninterrupted excitement and amusement for all parties involved.

Design Overview

The vehicle consists of 4 motors, 2 motor drivers. The drive system consists of four identical 12V DC motors with encoders, each directly driving a 70mm wheel. The motor mounts are printed directly into the structure of the bottom chassis plate. Control for the motors is handled by twin L298N motor drivers with logic handled by an ESP32 microcontroller

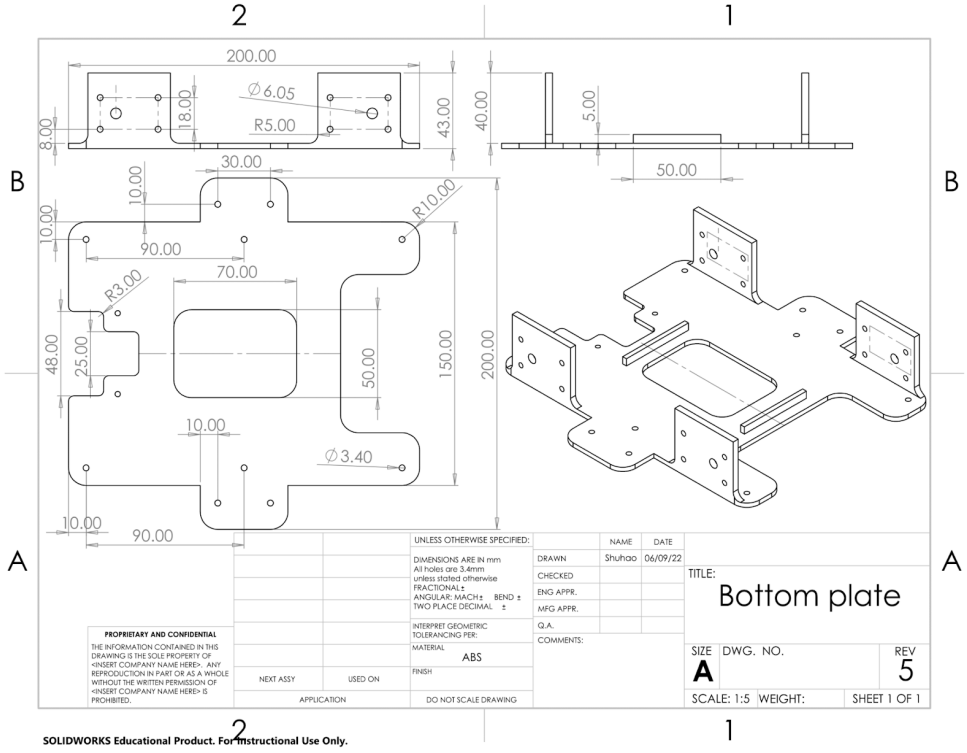


Figure 1: Engineering Drawing of Bottom Plate

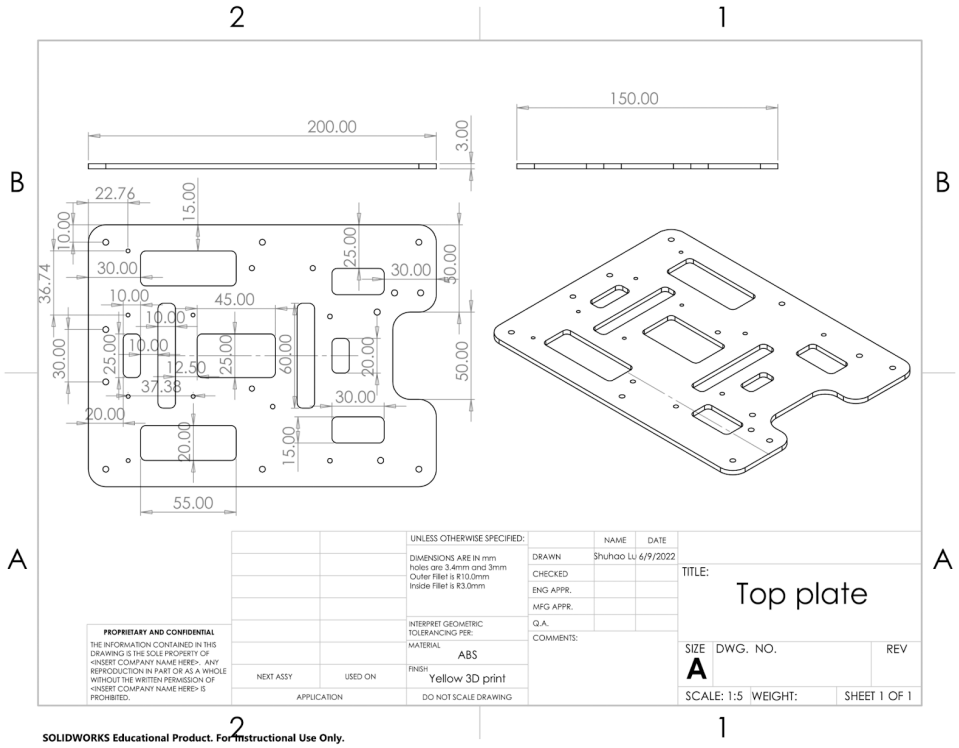


Figure 3: Engineering Drawing of Top Plate

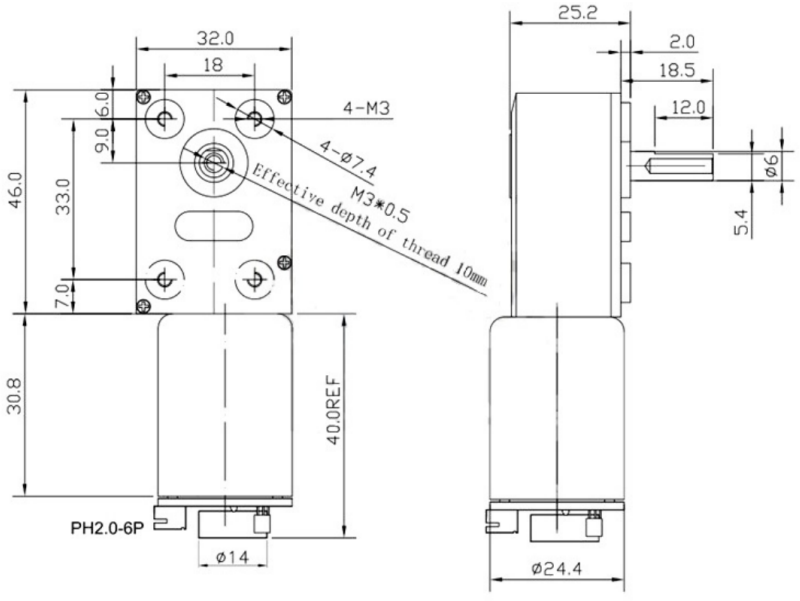


Figure 3: Engineering Drawing of Motor

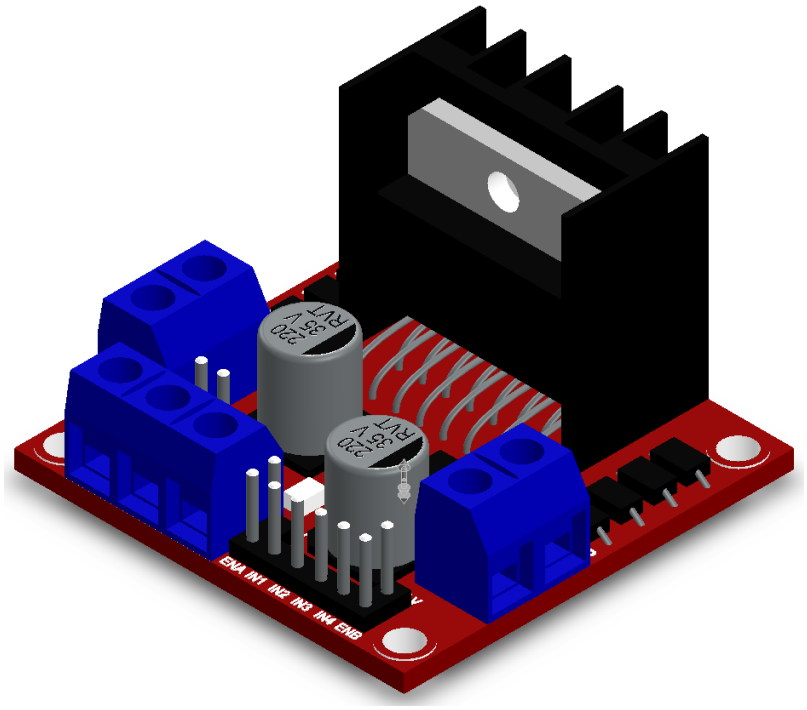


Figure 4: CAD Model of L298N Stepper Motor Driver

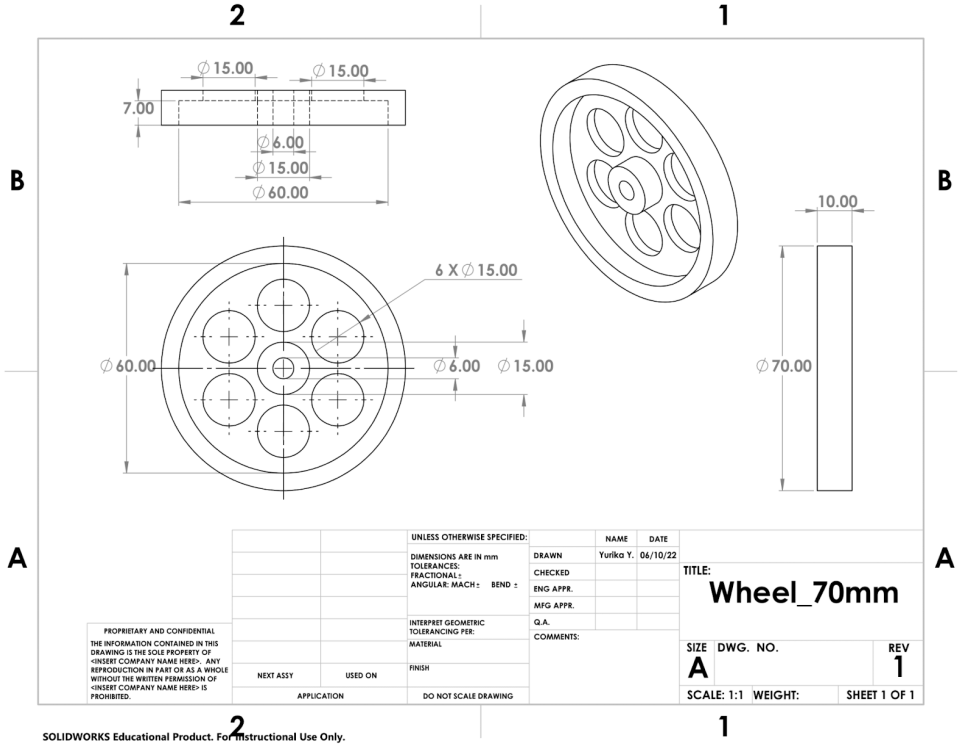


Figure 5: Engineering Drawing of 70 mm Wheel

Due to the complexity of the course, it was decided to use three ultrasonic sensors, each facing front, left, and right to obtain the distance information on the surroundings of the car.

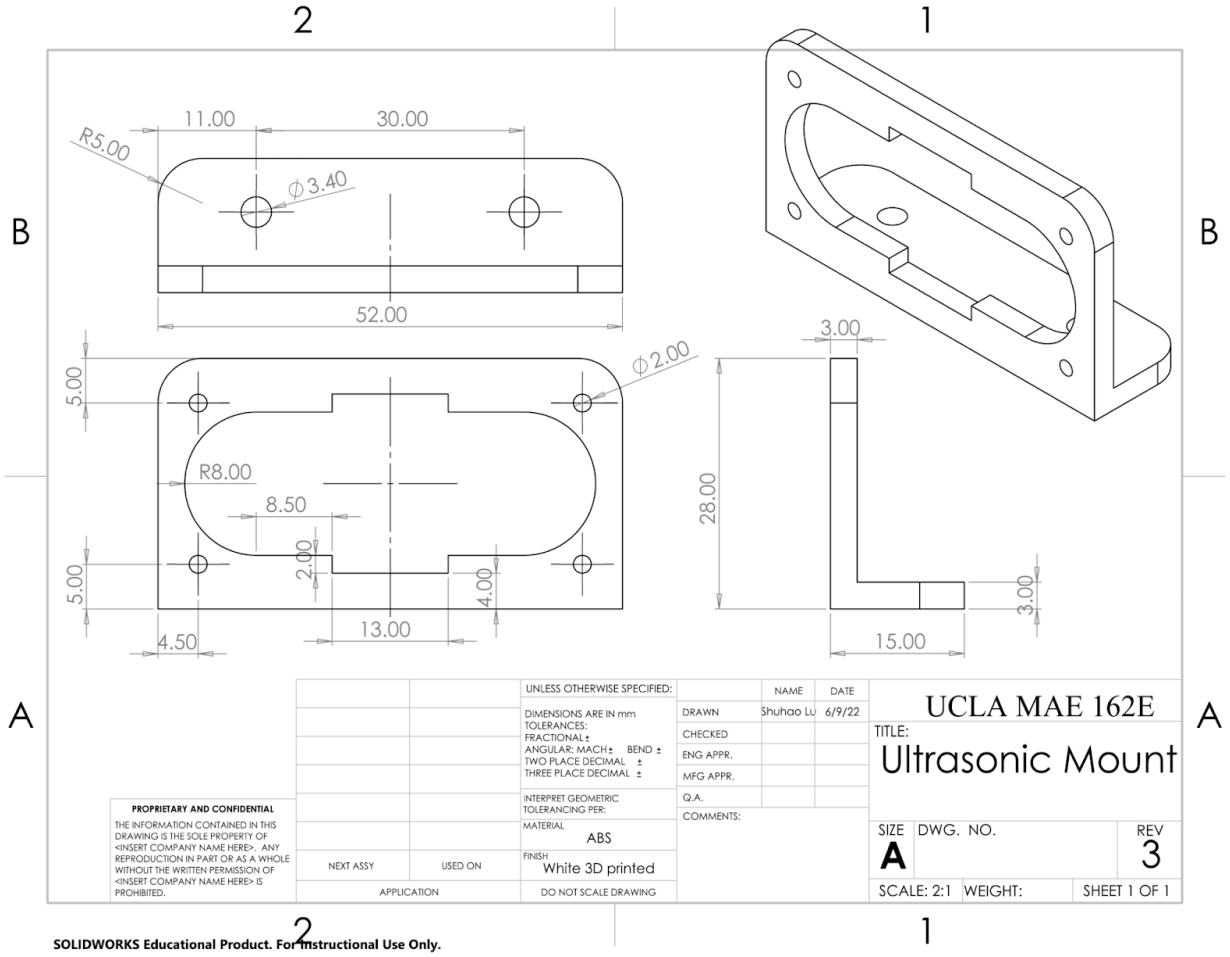


Figure 6: Engineering Drawing of Ultrasonic Sensor Mount

Based on the constraints from the high level design requirements laid out for this project, We obtained the following dimensions for the systems. Forces and Power calculations are also performed. Since the car is four wheel drive, The minimum coefficient is going to be 0.31, which is easy to achieve. The free body diagram of the vehicle driving up hill is shown on the next page

Table 1: Vehicle parameter and forces torque calculations

DESCRIPTION	SYMBOL	Variable Name	VALUE	UNITS
Vehicle/Hill Parameters:				
Mass	m	mass	2	kg
Wheelbase	L	L	0.146	m
CoM distance from Rear Wheel	Lc	Lc	0.0364	m
CoM Height from ground	hc	hc	0.0434	m
Force (weight)	Fmg	Fmg	19.6	N
Normal Wheel Forces and Tipping Angle:				
Incline Angle (hill)	%	theta	17	deg
Normal Force on front wheels	Nf	Nf	2.96961	N
Normal force on rear wheels	Nr	Nr	15.77396	N
CoM tipping height	hmax	hmax	0.119059	m
tipping angle	p	phi	17	deg
Minimum required friction coefficients:				
Minimum Friction for FWD	uf	mu_f	1.92971	
RWD	ur	mu_r	0.363288	
AWD	ua	mu_a	0.305731	
CoM distance for equal mu	Lc_equal	Lc_eq	0.086269	m
Minimum required Tractive Force for estimated mu				
Friction coeff. Mu_F = Mu_R	mu_W	mu_W	0.611461	
Rolling friction coeff.	mu_rol	mu_rol	0.003	
FWD tractive force	F_Tf	F_TF	1.815802	N
RWD tractive force	F_Tr	F_TR	9.645169	N
AWD tractive force	F_Tawd	F_TAWD	5.730485	N
Minimum required wheel torque based on tractive force				
Front wheel diameter	Dfw	di_fw	0.07	m
Rear wheel diameter	Drw	di_rw	0.07	m
Front wheel Torque	Tfw	t_fw	0.063553	N-m
Rear wheel torque	Trw	t_rw	0.337581	N-m
all wheel torque	Taw	t_aw	0.200567	N-m
Propulsion Power based on Tractive Force				
Drive system efficiency	n	eff	0.7	
RMS Velocity up ramp	v_rms	v_rms	0.2	m/s
FWD Power	P_FWD	P_FWD	0.254212	W
RWD Power	P_RWD	P_RWD	1.350324	W
AWD Power	P_AWD	P_AWD	0.802268	W

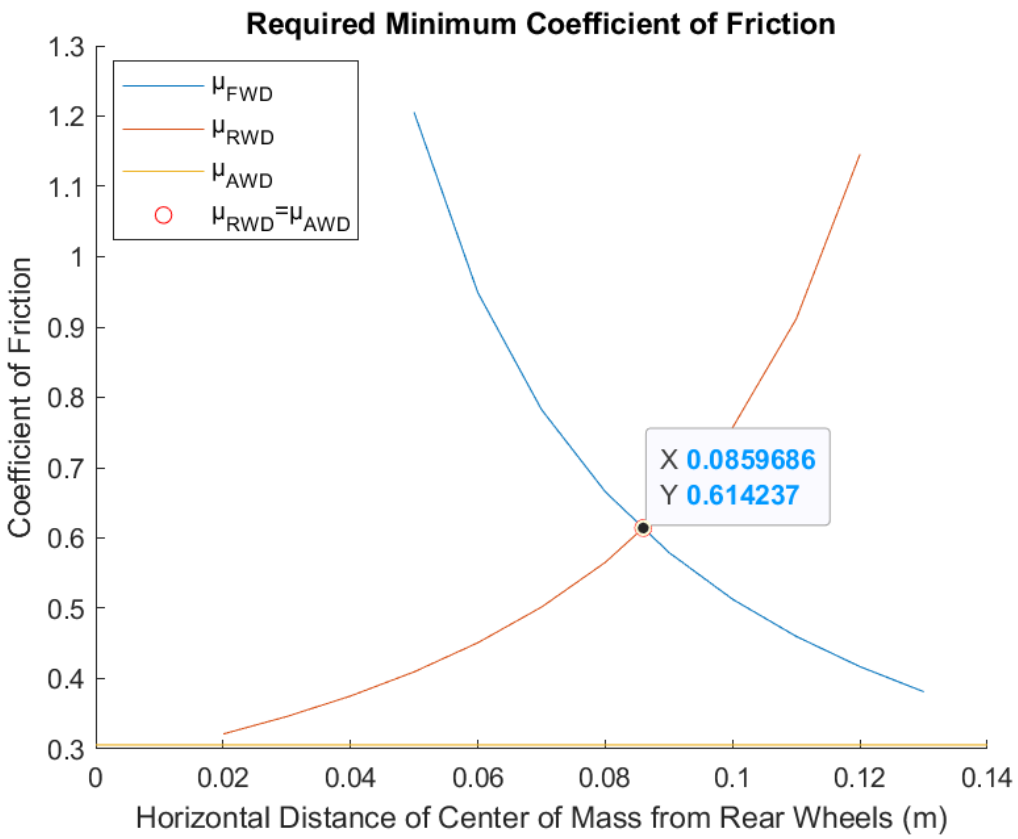


Figure 7: Minimum coefficient of friction with respect to center of mass

The required propulsion force of the transporter is calculated by the following equation:

$$F_{propulsion} = F_{inertia} + F_W + F_f + F_{rol} \quad (5)$$

The components of equation 3 are as follows:

$$F_{inertia} = ma \quad (6)$$

$$F_W = mg(\sin\theta) \quad (7)$$

$$F_f = \mu_{static} mg(\cos\theta) \quad (8)$$

$$F_{rol} = \mu_{rol} mg(sin\,\theta) \tag{9}$$

where m is mass, a is the calculated acceleration from Table 8, and g is gravity.

The required motor power is calculated by the following equation:

$$P_{prop} = F_{prop} \times V_{max} \tag{10}$$

Additionally, the required propulsion torque for the wheels for each path segment is calculated by the following equation:

$$T_{propulsion} = (F_{prop} \times \frac{wheel\ diameter}{2}) \frac{1}{\eta_{drive-system\ of\ efficiency}} \tag{11}$$

Motion Study

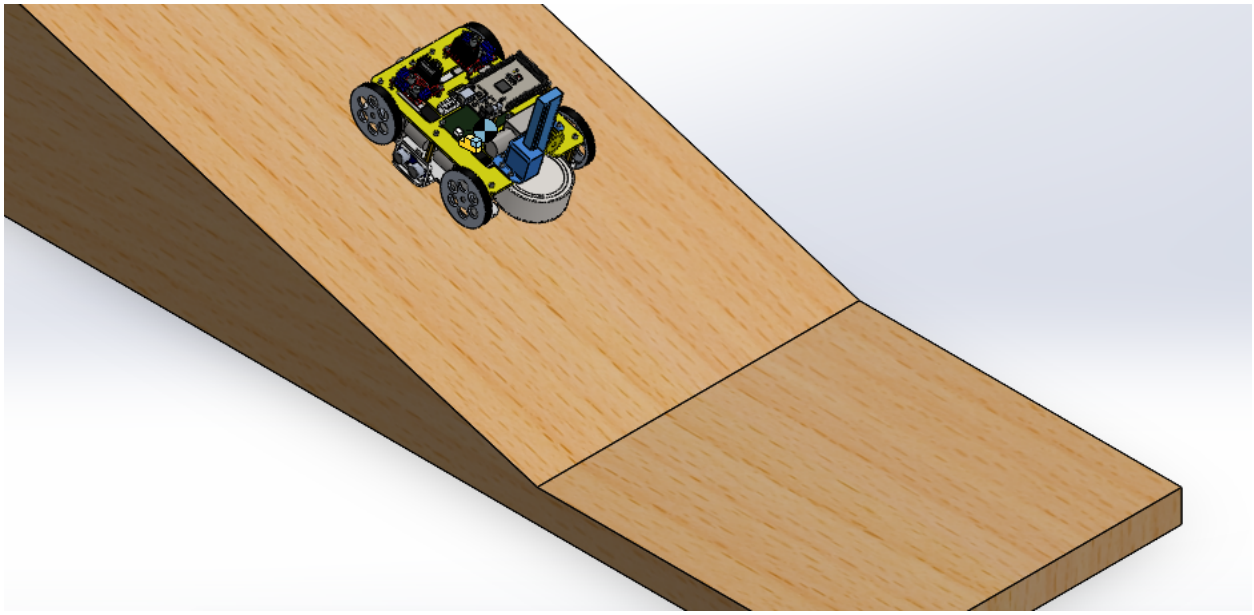


Figure 8: CAD model of vehicle on a 17 degree testing ramp.

To ensure that our motor choices would be adequate, a motion study was run on the vehicle using the parameters given (17 degree ramp, constant speed 40 RPM motor, coefficient of friction = 0.31). The torque output from one of the motors is as follows:

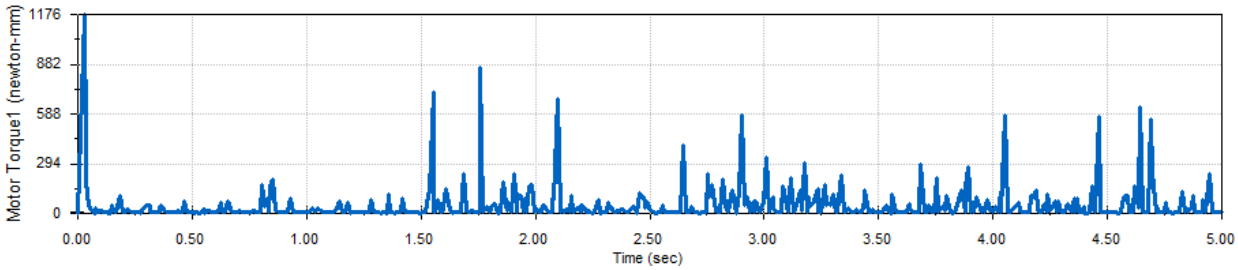


Figure 9: Torque plot of one motor in the 4WD vehicle traversing the ramp.

The required torque of the motor occasionally exceeds its calculated requirement during travel up the ramp and during startup. However, steady-state torque requirements of the motor generally below 0.3 N-m. This is reflective of the calculated motor torque requirements in section 4.1.1. During the actual run, the large spikes in torque would amount only to a slight slowdown in vehicle speed. There is sufficient friction in the wheels to prevent the vehicle from sliding down the hill, and since we made the choice to use worm drive gearboxes, gravity cannot roll the vehicle down the hill due to the wheels being unable to be back-driven.

Motor selection and detail

The motors that we selected for our car were chosen based on the output torque, the worm gear drives that they have which prevent them from being back-driven and their compact size. Unfortunately there are no performance curves for this motor.

Table 2: Parameters of worm gear drive motor

Gear ratio	1/150
No load current	60 mA
No load speed	40 rpm
Rated Torque	5.6 Kg*cm
Rated Current	0.6 A
Stall Current	1.3 A
Maximum Torque	24 Kg*cm

Ultrasonic Sensors

In this project, three HC-SR04 arduino ultrasonic sensors are used to measure the distance to obstacles from the front, left and right. The sensor is composed of two ultrasonic transducers. One outputs ultrasonic sound pulses and the other is a receiver which listens for reflected waves.

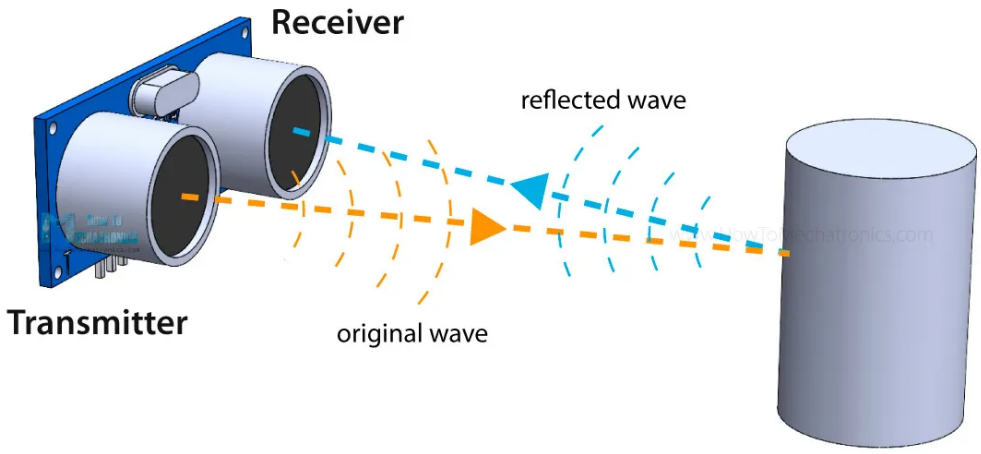


Figure 9: Ultrasonic sensor operating principle [3]

Table 3: Main specifications of HC-SR04

Operating Voltage	5V DC
Operating Current	15mA
Operating Frequency	40kHz
Min Range	2 cm
Max Range	400 cm
Accuracy	Relative error ≈ 5% of range
Measuring Angle	<15°
Dimension	45 ✕ 20 ✕ 15 mm

It should be noted however, that the accuracy of the ultrasonic sensor decreases greatly with distance. The relative error of the ultrasonic sensor is about 5% of range, meaning that at 1m the error is about 5cm. It can also be affected by other external factors such as temperature and humidity because the speed of sound changes in those conditions. As such it is best to use these sensors when the obstacles are close to it. Variation in the reading when the ultrasonic sensor is not moving should be expected as well.

Additionally when the ultrasonic sensor is extremely close to the obstacles, for example below 1 cm, the reading will stop reflecting the actual distance and instead display distance significantly larger than the actual distance. In the testing when placing the ultrasonic directly in front of the obstacles, the reading will be about 1m with random large variation. As such, it is best to not place the ultrasonic sensor directly on the edge of the vehicle, but instead, 1-2 cm away from the edge to prevent erroneous readings.

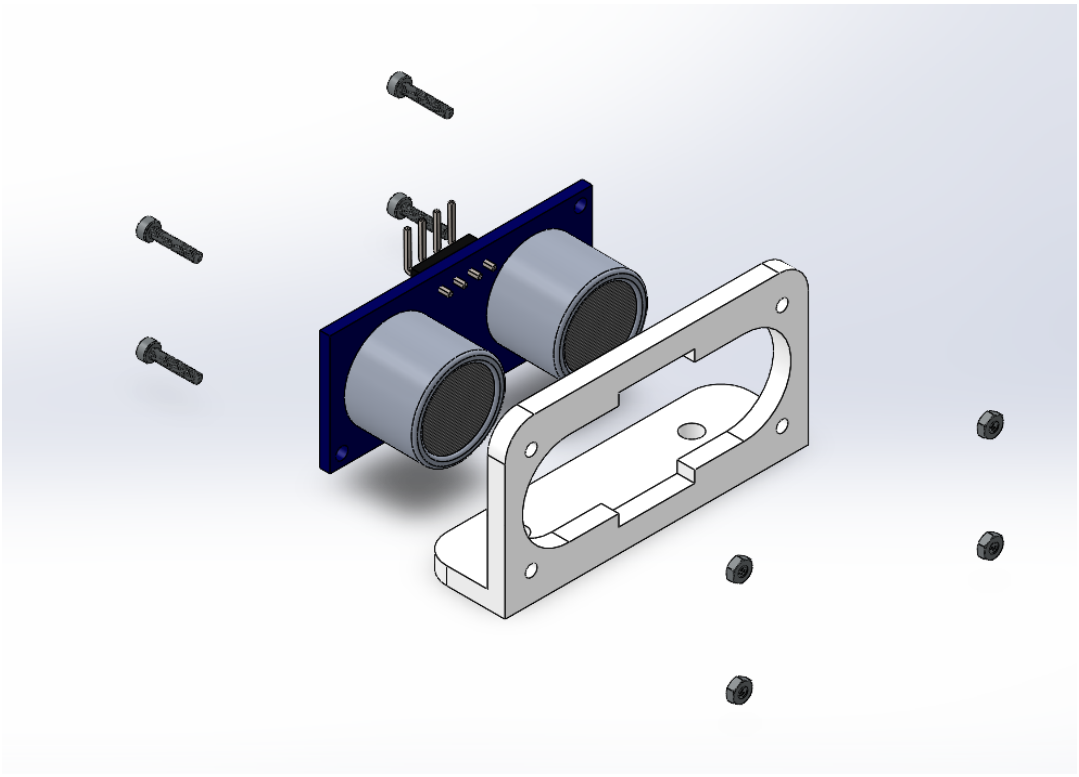


Figure 10: Exploded View of Ultrasound Sensor and Holder

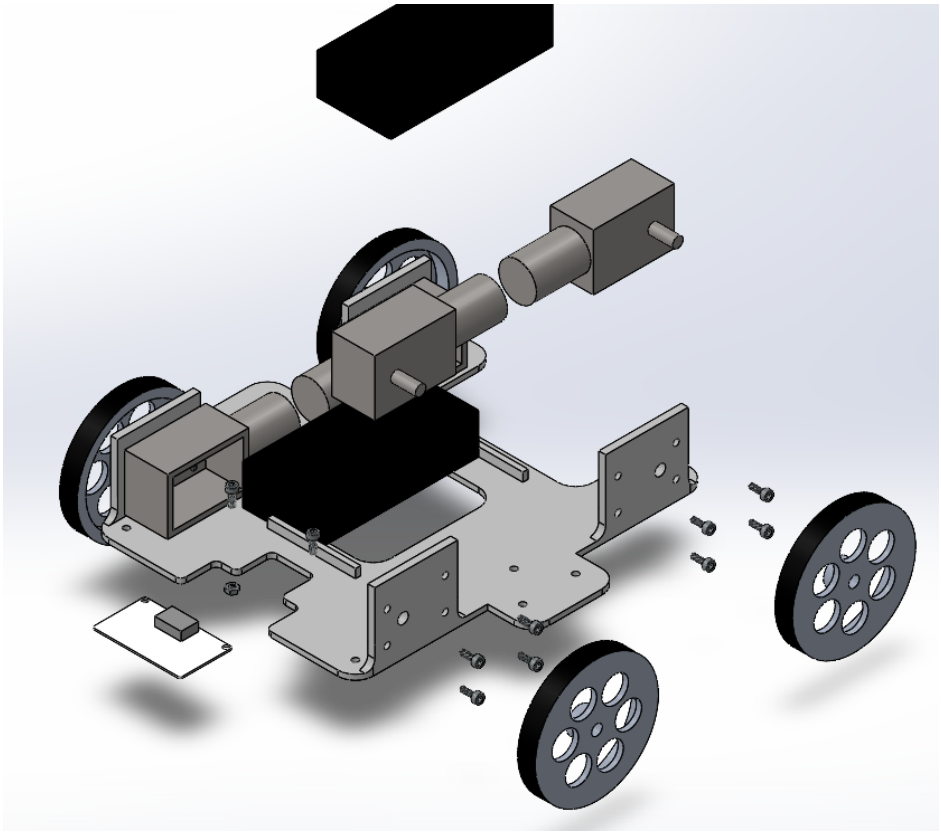


Figure 11: Exploded View of Lower Board

Programming and Navigation

1. Environment and integrated development environment(IDE)

The environment we use for the project development is micropython. And the IDE we choose for programming is Thonny, which is an excellent platform for micropython. Thonny not only has high performance for micropython but also is user friendly for python beginners. There's a huge amount of inner packages that could be used for embedded software development on micropython.

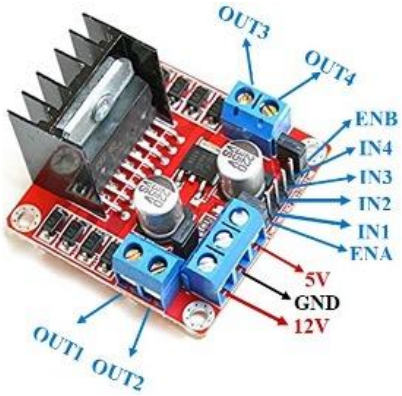
2. Basic python module we've written

After we have all the hardware and circuitry connected and tested. We have finished the following two basic python script files for our project. Wheelmotor.py, which is used for controlling four motors to do the actions like move forward, move backward, turn left, turn right and stop. And Ultrasound.py, which is used to track and receive the digital data return from three different ultrasound sensors.

a) Wheelmotor.py for controlling four wheels

The following image shows part of the wheel.py code. We connect the GPIO pins of the motor driver module L298N in the eps32 by inputting high and low levels respectively, so that the motor can drive the wheel to rotate, thus achieving the ability to move forward, backward, left and right.

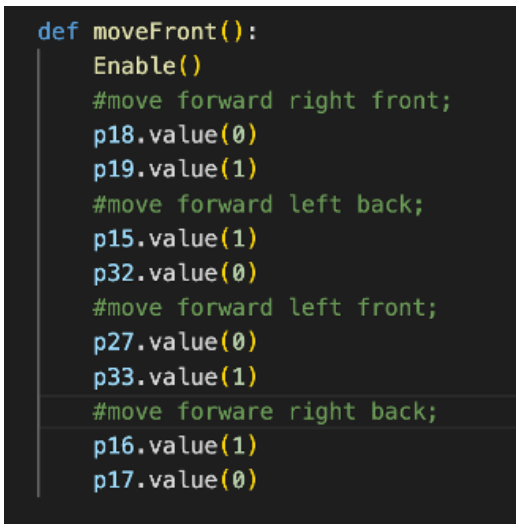
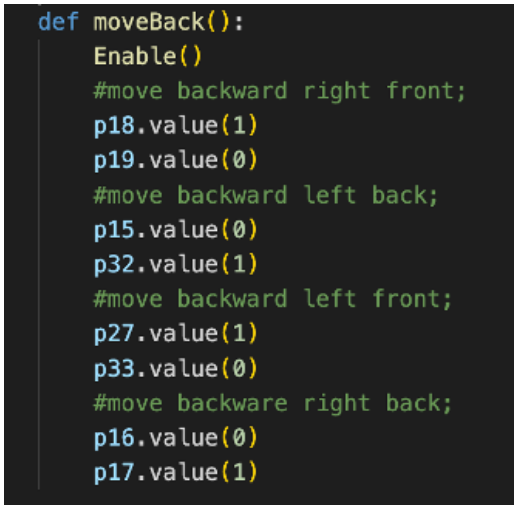
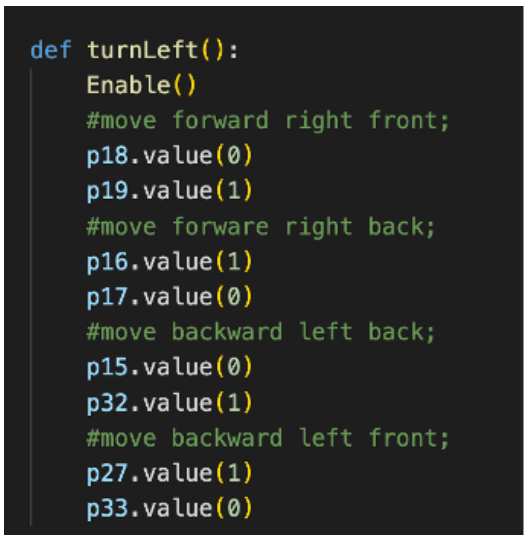
It should be noted that due to the characteristics of the L298N driver module, one L298N motor driver module can drive two motors to work. When connecting the wires, the eps32 microcontroller needs to assign two additional GPIO pins for ENA and ENB respectively for both motors.When the motor starts, it needs to input high level to ENA and ENB to ensure the motor can reach normal voltage.




```
def Enable():
    p13.value(1)
    p14.value(1)
    p5.value(1)
    p21.value(1)
```

Next, we've defined five python functions: moveFront(), moveBack(),turnLeft(), turnRight(),stop().That shows how we use motors to control cars to do these actions.

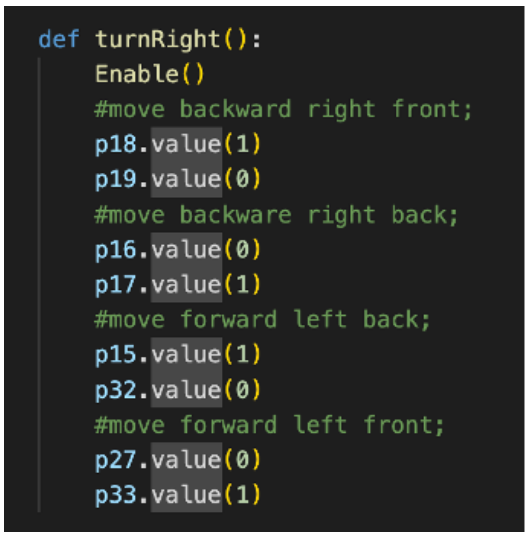
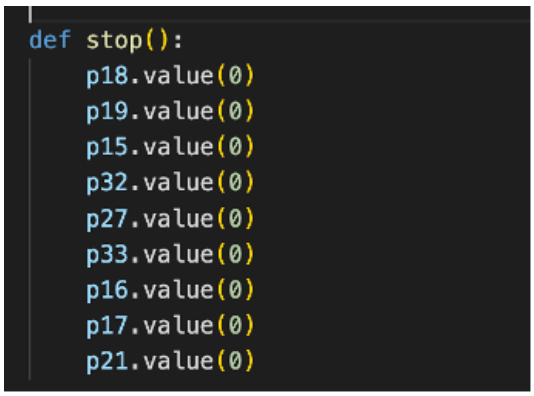
Table 4: Parameters of worm gear drive motor

moveFront()	
moveBack()	
turnLeft()	

```
def moveFront():
    Enable()
    #move forward right front;
    p18.value(0)
    p19.value(1)
    #move forward left back;
    p15.value(1)
    p32.value(0)
    #move forward left front;
    p27.value(0)
    p33.value(1)
    #move forware right back;
    p16.value(1)
    p17.value(0)
```

```
def moveBack():
    Enable()
    #move backward right front;
    p18.value(1)
    p19.value(0)
    #move backward left back;
    p15.value(0)
    p32.value(1)
    #move backward left front;
    p27.value(1)
    p33.value(0)
    #move backware right back;
    p16.value(0)
    p17.value(1)
```

```
def turnLeft():
    Enable()
    #move forward right front;
    p18.value(0)
    p19.value(1)
    #move forware right back;
    p16.value(1)
    p17.value(0)
    #move backward left back;
    p15.value(0)
    p32.value(1)
    #move backward left front;
    p27.value(1)
    p33.value(0)
```

turnRight()	
stop()	

```
def turnRight():
    Enable()
    #move backward right front;
    p18.value(1)
    p19.value(0)
    #move backware right back;
    p16.value(0)
    p17.value(1)
    #move forward left back;
    p15.value(1)
    p32.value(0)
    #move forward left front;
    p27.value(0)
    p33.value(1)
```

```
def stop():
    p18.value(0)
    p19.value(0)
    p15.value(0)
    p32.value(0)
    p27.value(0)
    p33.value(0)
    p16.value(0)
    p17.value(0)
    p21.value(0)
```

b) Ultrasound.py for sensor Object.

Next basic python script file would be Ultrasound.py, which is written for reading and receiving the signal data returned from three different sensors. In order to be able to program efficiently, it is important to manage the code modularly, so I encapsulated those highly reusable programs in a class. This way, for each different sensor, I only need to create a different sensor object and read the position data from the respective sensor by calling the function wrapped inside the class. And further processing of the data to achieve a variety of functions, such as the obstacle avoidance function we implement later in ultra_action.py file.

The following part shows the python code for Ultrasound.py:

```
class ORGHCSR04_ULTR:
    def __init__(self,trigNum,echoNum):
        self.trigNum = trigNum
        self.echoNum = echoNum
        self.trig = Pin(trigNum, Pin.OUT)
        self.echo = Pin(echoNum, Pin.IN)
        self.trig.off()
        self.echo.off()

    def start_scan(self):
#         while True:
            dist = self.start_hc()
#             self.operation(dist)
            utime.sleep_ms(200) # 这里根据需要设定SLEEP时间
            return dist

    def start_hc(self):
        self.trig.on()
        utime.sleep_us(10)
        self.trig.off()
        while self.echo.value() == 0 :
            pass
        start_us = utime.ticks_us()
        while self.echo.value() == 1 :
            pass
        end_us = utime.ticks_us()
        rang_us = utime.ticks_diff(end_us,start_us)/10000
        dist = rang_us*340/2
        return dist
```

The above picture is the Ultrasound class for ultrasound sensor. The class name is ORGHCSR04_ULTR.

The next line is the constructor for the class. The constructor indicates what parameter we need to pass in to crate the object. Here we take trigNum and echoNum as input for the constructor function. The number of trigNum and echoNum is the GPOI pins on esp32 that are connected to the trig and echo pins on the ultrasound sensor.

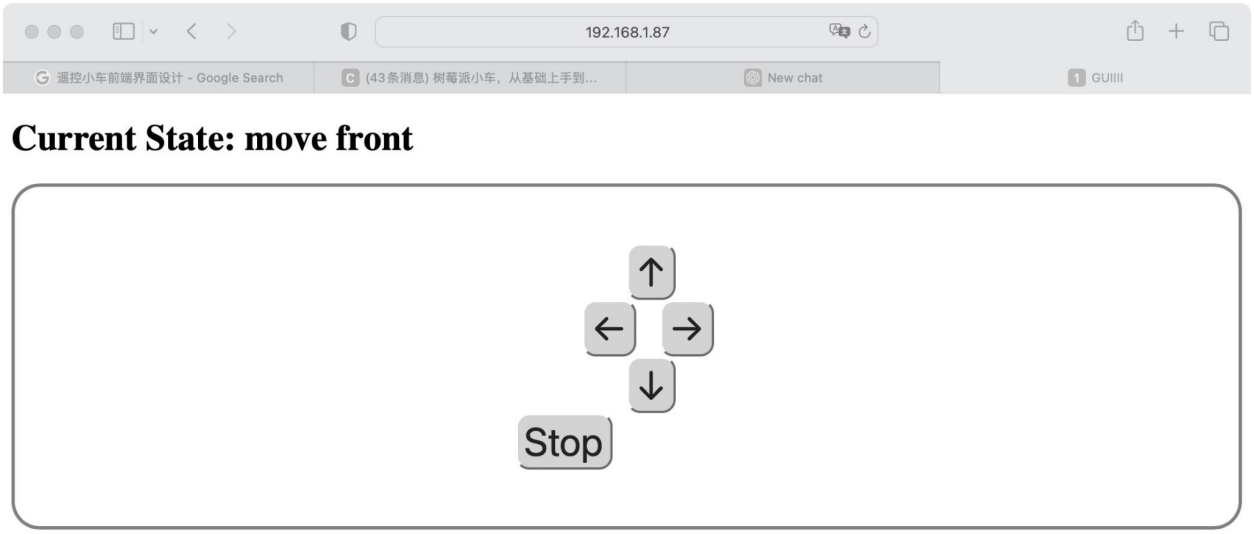
start_hc(self) is the function that once a user call, it would return the real-time distance for the user. The writing of this code involves the principle of distance measurement by Ultrasound sensor. In principle, ultrasound distance measurement is calculated by calculating the time difference between the echos transmitting and receiving signals, combined with the propagation speed of ultrasound in the air, to comprehensively calculate the current distance of the cart from the obstacle. As we can

see in this code. We initially set the echo value as high level to transmit the signal , after that we set the echo value as low level to receive the signal.

We also calculate the cumulated time needed for this process by calling the function utime.ticks_us() and calculate the distance based on the velocity.

3. GUI web page design

This part would introduce our GUI and web page we've designed for our project.



a) Webpage Html

In the appendix, code is provided by HTML part 1 and 2. The provided code represents a simple HTML webpage with a graphical user interface (GUI) for controlling movement in different directions. The webpage consists of the head section, style section, and body section.

Starting with the head section, it contains metadata and styling information for the web page. The meta tag specifies the character encoding for the page content, ensuring proper display of characters. The title tag sets the title of the webpage, which appears in the browser's title bar.

Moving on to the style section, it defines the CSS (Cascading Style Sheets) rules that control the appearance of the web page elements. The defined styles target different CSS classes and elements within the webpage. For instance, there are styles defined for

buttons, including font size, background color, and border radius. The :hover pseudo-class is utilized to apply a box-shadow effect when hovering over the buttons, enhancing the interactive experience.

In the body section, the webpage's visible content is included. The primary element is an h1 heading, which displays the text "Current State: {}". The curly braces {} serve as a placeholder that can be dynamically filled with a value using server-side programming. This allows for the dynamic update of the current state information on the webpage.

The GUI container is created using a div element with the class "border." This div acts as a container for the GUI elements and has a border, padding, and other styles applied to it. Inside this container, the GUI buttons for controlling movement are positioned.

There are four directional buttons: up (↑), left (←), right (→), and down (↓). Each button is enclosed in a tag, which represents a hyperlink. The buttons are styled using CSS classes and have specific styles applied to them, such as font size, background color, and a rounded border. When a user hovers over the buttons, a box-shadow effect is applied, creating a visual feedback mechanism.

The positioning of the buttons is achieved through the use of CSS classes and relative/absolute positioning. The buttons are divided into different sections using classes like "front," "middle," "left," "right," "back," and "stop." These classes are responsible for positioning the buttons accordingly on the webpage.

Each button is wrapped in a tag with an associated URL path. When a user clicks on a button, it will navigate to the specified URL path. The URLs "/Front," "/Left," "/Right," "/Back," and "/STOP" correspond to server-side endpoints that handle the movement commands. By clicking on a specific button, the user can indicate the desired direction of movement.

b) Wifi.py

```
async def connectWeb():
    # 创建 socket 对象
    s = socket.socket()
    s.bind(('0.0.0.0', '80'))
    # 监听 80 端口
    s.listen(1)
    State = "Initialization"
    while True:
        # socket 阻塞外部连接
        cl, addr = s.accept()
        # 拿到客户端 IP 地址
        print('Client connected from', addr)
        # 拦截客户端请求
        request = cl.recv(1024)
        if request.decode()[:100].find("Front") != -1:
            wheelmotor.moveFront()
            State = "move front"
            await asyncio.sleep(2.6)
            wheelmotor.stop()
        elif request.decode()[:100].find("Back") != -1:
            wheelmotor.moveBack()
            State = "move back"
            await asyncio.sleep(2.6)
            wheelmotor.stop()
        elif request.decode()[:100].find("Left") != -1:
            wheelmotor.turnLeft()
            State = "turn left"
            await asyncio.sleep(1.5)
            wheelmotor.stop()
        elif request.decode()[:100].find("Right") != -1:
            wheelmotor.turnRight()
            State = "turn right"
            await asyncio.sleep(1.5)
            wheelmotor.stop()
        elif request.decode()[:100].find("STOP") != -1:
            wheelmotor.stop()
            State = "Stop"
        response = html.format(State)
        cl.send(response)
        cl.close()
```

The provided code presents a Python function, `connectWeb()`, which serves as a web server and facilitates the control of a wheel motor based on incoming client requests. The function utilizes asynchronous programming, allowing for concurrent execution and non-blocking I/O operations, which is advantageous in handling multiple client requests concurrently.

To begin, the function creates a socket object using the `socket.socket()` function.

Sockets enable communication between different processes. The socket is then bound to the IP address '0.0.0.0' and port '80'. Binding associates the socket with a specific address and port, enabling it to listen for incoming connections on that address and port. The subsequent `s.listen(1)` call sets the socket to listen for incoming connections, with a maximum backlog of 1 connection.

The function enters a continuous loop using `while True`, which allows it to accept and handle incoming client connections. Upon a client's connection attempt, the function calls `s.accept()`, causing the execution to block until a connection is established. When a connection is established, the function returns a new socket object, `cl`, representing the connection, along with the client's IP address, `addr`.

Upon establishing a connection, the code proceeds to handle the client's request. It receives the client's request using `request = cl.recv(1024)`, which retrieves up to 1024 bytes of data from the client. The request's content is then examined to determine the desired movement command. This is achieved through string manipulation and the utilization of the `find()` method, enabling the identification of specific keywords such as "Front," "Back," "Left," "Right," or "STOP" within the request.

Based on the recognized command, the code initiates the appropriate action using the `wheelmotor` object. For example, if the command indicates "Front," the function `wheelmotor.moveFront()` is called to set the wheel motor in motion, moving it forward. The `State` variable is updated correspondingly to reflect the current action being performed. Additionally, the code introduces an asynchronous sleep period using `await asyncio.sleep()`, pausing the execution for a specified duration. This delay simulates a realistic time interval, allowing the motor to perform the movement for a designated period before coming to a stop. Once the movement is completed, the `wheelmotor.stop()` function is invoked to halt the motor.

Following the execution of the movement commands, a response string is generated using `html.format(State)`. This involves an HTML template mentioned above where the current state, stored within the `State` variable, is dynamically inserted into the HTML content.

The response is subsequently sent back to the client utilizing `cl.send(response)`. This action transmits the generated HTML response string back to the client, providing them with the relevant information regarding the current state.

Finally, the connection is terminated via `cl.close()`, effectively concluding the communication between the server and the client.

```
#编辑异步任务
def runTask():

    #create event loop
    loop = asyncio.get_event_loop()
    loop.create_task(connectWeb())
    loop.run_forever()

# runTask()
```

After the connect web function, The `runTask()` function sets up an event loop, schedules the `connectWeb()` function as a task within that loop, and then enters the loop to execute the tasks indefinitely using `run_forever()`. This structure enables the asynchronous execution of the `connectWeb()` function alongside other tasks, allowing for concurrent handling of client connections and movement control of the wheel motor.

4. Multitasking

a) Multitasking and real-time programming with sensors in ultra_Action.py

```python
#import necessary module
from ultraSound import ORGHCSR04_ULTR
import wheelmotor
#uasyncio asynchronous programming library
#allow to run multitasing in one thead with coroutine
import uasyncio
import time

#coroutine function:
async def read_ultr(ultr, delay):
    while True:
        #To make sure it's real time progamming:
        #I recorded the time stamps for the start and end of the mission
        start_time = time.ticks_us()
        dist = ultr.start_scan()
        print("%s: %0.2f CM" % (ultr.name, dist))
        if ultr.name == 'Front' and dist < 20:
            wheelmotor.moveBack()
            #uasyncio.sleep(1.5) is an asynchronous function
            #provided in the uasyncio library that suspends execution in a concurrent process,
            #allowing the event loop to continue executing other concurrent processes,
            #and then resumes execution of the concurrent process after a specified amount of time
            await uasyncio.sleep(1.5)
            wheelmotor.stop()
        if ultr.name == 'Left' and dist < 20:
            wheelmotor.turnRight()
            await uasyncio.sleep(1.5)
            wheelmotor.stop()
        if ultr.name == 'Right' and dist < 20:
            wheelmotor.turnLeft()
            await uasyncio.sleep(1.5)
            wheelmotor.stop()
        #elaspsed_time is the total time needed for run this function
        elapsed_time = time.ticks_diff(time.ticks_us(), start_time)
        #if elaspsed_time < delay : we wait for delay_elaspsed_time
        #so that coroutine would be execurte with in delay time
        if elapsed_time < delay:
            await uasyncio.sleep_ms(delay - elapsed_time)
        else:
            await uasyncio.sleep_ms(0)
```

The provided code demonstrates the usage of asynchronous programming and the uasyncio library to perform concurrent tasks and control a wheel motor based on readings from ultrasonic sensors.

The code begins by importing necessary modules, including `ultraSound` and `wheelmotor`, which contain the necessary functionalities for interacting with the ultrasonic sensors and controlling the wheel motor, respectively. The `uasyncio` module is also imported, enabling the use of asynchronous programming.

Next, the code defines a coroutine function called `read_ultr()`. Coroutines are special functions that allow for non-blocking, concurrent execution. This function takes two parameters: `ultr`, representing an ultrasonic sensor object, and `delay`, indicating the

delay time between successive readings.

Within the `read_ultr()` coroutine, an infinite loop is created using `while True`, ensuring continuous sensor readings. The `ultr.start_scan()` method is called to obtain the distance reading from the ultrasonic sensor, and the result is printed to the console.

Based on the name of the ultrasonic sensor (`ultr.name`) and the measured distance (`dist`), the code determines the appropriate action for the wheel motor. If the sensor is the front ultrasonic sensor (`ultr.name == 'Front'`) and the distance is less than 20 cm, the motor is instructed to move backward (`wheelmotor.moveBack()`) for a specified duration using `await uasyncio.sleep(1.5)`, and then stop (`wheelmotor.stop()`). Similar logic is applied for the left and right sensors, causing the motor to turn in the respective direction if the distance is below the threshold.

To ensure real-time behavior, the code measures the elapsed time for executing the function using `time.ticks_us()` before and after the sensor reading and decision-making process. If the elapsed time is less than the specified delay, the coroutine suspends execution for the remaining time using `await uasyncio.sleep_ms(delay - elapsed_time)`. This allows the coroutine to execute within the desired delay time. If the elapsed time exceeds the delay, the coroutine immediately continues to the next iteration without waiting.

```
#in main function, we generate thre Ultrasound object as three task
async def main():
    #front ultr
    Ultr1 = ORGHCSR04_ULTR(25,26,"Front")
    #left ultr
    Ultr2 = ORGHCSR04_ULTR(4,36,"Left")
    #right ultr
    Ultr3 = ORGHCSR04_ULTR(12,34,"Right")
    #Three corountine tasks with delay time
    #Ultr1 waiting time 50ms , Ultr2 waiting time 100ms , Ultr3 waiting time 150ms
    tasks = [
        read_ultr(Ultr1, 50),
        read_ultr(Ultr2, 100),
        read_ultr(Ultr3, 150),
    ]

    await uasyncio.gather(*tasks)

#create event loop
loop = uasyncio.get_event_loop()
loop.create_task(main())
loop.run_forever()

#run mian function
main()
```

The `main()` coroutine function is then defined, responsible for creating and coordinating the ultrasonic sensor tasks. Three ultrasonic sensor objects (`Ultr1`, `Ultr2`, `Ultr3`) are instantiated with their respective pin configurations and names. Three coroutines are created using the `read_ultr()` function, each associated with an ultrasonic sensor object and a specified delay time. These coroutines are stored in the `tasks` list.

The `main()` coroutine uses `uasyncio.gather()` to schedule the execution of all

the tasks concurrently. The `gather()` function takes the `tasks` list as input and returns a single awaitable object that represents the combined execution of all the tasks.

The event loop is obtained using `uasyncio.get_event_loop()`. The `main()` coroutine is then scheduled as a task within the event loop using `loop.create_task(main())`. Finally, the event loop runs indefinitely using `loop.run_forever()`, ensuring the continuous execution of the scheduled tasks.

The last line of code calls the `main()` function directly, which might be unnecessary since the function is already scheduled as a task within the event loop.

b) Multitasking between wifi.py and ultra_Action.py

```
import ultra_Action
import uasyncio as asyncio
import wifi


async def task1():
    # 异步任务1
    ultra_Action.run()
    await asyncio.sleep(1)
    print("Task 1 done")

async def task2():
    # 异步任务2
    wifi.runTask()
    await asyncio.sleep(2)
    print("Task 2 done")

async def main():
    # 主函数
    await asyncio.gather(task1(), task2())

asyncio.run(main())
```

In the final step, we want wifi.py could run simultaneously with three sensors in ultra_Action.py, to implement the multitasking between both python code.The best idea is using asyncio library to build two coroutines, each coroutine is a single task.The task1() function runs the ultra_Action library and then waits for 1 second before printing “Task 1 done”.The task2() function runs the wifi async library and then waits for 2 seconds before printing “Task 2 done”.The main() function is the entry point of the program. It uses asyncio.gather() to run both task1() and task2() concurrently2.

Future improvements

First and foremost, we would focus on refining the interaction between the cat and the vehicle. Understanding that the cat's engagement is key, we would explore the

incorporation of additional features or accessories that would captivate the cat's curiosity and stimulate its playfulness. This could involve attaching dangling toys with a motor attached, enticing the cat to actively participate in the play experience and fostering a stronger bond.

Furthermore, We would allocate time and effort to enhancing the user interface, the current setup only allows users to use a mouse to click on each button, or use fingers if they are on a mobile device, in the future, WASD keyboard controls or Xbox controllers can be incorporated, making the input much easier for the user. Additionally, the current setup has pre-planned motion, each command askes the vehicle to move only at specific time intervals, if a keyboard input is used, maybe the vehicle can move as long as keystrokes are pressed, this allows the user to precisely control the motion of the vehicle.

To further enrich the user experience, we would consider the incorporation of visual feedback. This will involve the integration of a camera to take photos and video of the cats in real time and stream the materials on Youtube for Prof. Anwar and classmates to see. However, this will be resource intensive and likely requires a second ESP32 to achieve the functionality.

Appendix

```
html = """<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;charset=UTF -8" />
    <title>GUIIII</title>
    <style>
      button {{
        font-size: 35px;
        background-color: lightgray;
        border-radius: 12px;
      }}
      button:hover {{
        box-shadow: 0 12px 16px 0 rgba(0, 0, 0, 0.24),
          0 17px 50px 0 rgba(0, 0, 0, 0.19);
      }}
      .border {{
        border-width: 10px;
        border-radius: 25px;
        padding: 50px;
        border: 3px solid gray;
        box-sizing: border-box;
      }}
      .front {{
        position: relative;
        left: 50%;
      }}
      .middle {{
        position: relative;
      }}
      .left {{
        position: relative;
        left: 47%;
      }}
      .right {{
        position: absolute;
        left: 53%;
      }}
      .back {{
        position: relative;
        left: 50%;
      }}
      .stop {{
        position: relative;
        left: 40%;
      }}
    </style>
  </head>
  <body>
```

HTML part 1

```
      }}
      .back {{
        position: relative;
        left: 50%;
      }}
      .stop {{
        position: relative;
        left: 40%;
      }}
    </style>
  </head>
  <body>
    <h1>Current State: {}</h1>
    <div class="border">
      <div class="front">
        <a href="/Front"><button>↑</button></a>
      </div>
      <div class="middle">
        <span class="left">
          <a href="/Left"><button>←</button></a>
        </span>
        <span class="right">
          <a href="/Right"><button>→</button></a>
        </span>
      </div>
      <div class="back">
        <a href="/Back"><button>↓</button></a>
      </div>
      <div class="stop">
        <a href="/STOP"><button>Stop</button></a>
      </div>
    </div>
  </body>
</html>
"""
```

HTML part 2