FINAL REPORT

Mazen Aboubakr

Soad Emad

Malak el Moazen

Abdallah Taman

Ibrahim Elshami

1. Hardware

1.1 Microcontroller circuit

Schematic:

The circuit was made by using the atmega328p-u microcontroller to mimic a circuit so close to the Arduino Uno on

1.ATmega328p and its connection:

1) PC6 **(pin 1)** I/O pin used as the **RESET** pin connected to:
- hardware reset, When the voltage on this pin drops to a low level (typically below a certain threshold), the microcontroller resets restarting its program execution.
- Pull-up resistance (R1) is connected between the RESET pin (PC6) and the +5V, This resistor keeps the pin at a high voltage level when no reset signal is applied, preventing accidental resets, and the electrolytic polarized capacitor connected with the reset button 7805 to minimize debouncing problems

2) PD0 **(pin 2)** is connected to the pin 0 in an 8-pin header pack
3) PD1 **(pin 3)** is connected to the pin 1 in an 8-pin header pack
4) PD2 **(pin 4)** is connected to the pin 2 in an 8-pin header pack
5) PD3 **(pin 5)** is connected to the pin 3 in an 8-pin header pack
6) PD4 **(pin 6)** is connected to the pin 4 in an 8-pin header pack
7) PD5 **(pin 11)** is connected to the pin 5 in an 8-pin header pack
8) PD6 **(pin 12)** is connected to the pin 6 in an 8-pin header pack
9) PD7 **(pin 13)** is connected to the pin 7 in an 8-pin header pack

- 2) to 9) all have the same use in our schematic which is a general-purpose digital input/output pin. It can be configured as an input or output through software, and its state can be read or set using the appropriate I/O registers.

10) VCC **(pin 7)** connected directly to the +5v output of the linear regulator 7805 to power the whole microcontroller circuit
11) GND **(pin 8,22)** connected directly to the common ground between the battery and regulator

12) PB6 **(pin 9)** is connected to
- One side from the crystal oscillator provides the clock for the microcontroller, PB6 serves as the input for the oscillator signal
- A ceramic capacitor(C4) to stabilize the crystal's oscillation frequency and form a resonant circuit with the crystal. This resonance ensures that the oscillation amplitude is consistent

13) PB7 **(pin 10) is connected to**

- The other side from the crystal oscillator provides the clock for the microcontroller, PB6 serves as the output for the oscillator signal
- A ceramic capacitor(C5) to stabilize the crystal's oscillation frequency and form a resonant circuit with the crystal. This resonance ensures that the oscillation amplitude is consistent

14) PB0 **(pin 14)** is connected to the pin 8 in a 10-pin header pack
15) PB1 **(pin 15)** is connected to the pin 9 in a 10-pin header pack
16) PB2 **(pin 16)** is connected to the pin 10 in a 10-pin header pack
17) PB3 **(pin 17)** is connected to the pin 11 in a 10-pin header pack
18) PB4 **(pin 18)** is connected to the pin 12 in a 10-pin header pack
19) PB5 **(pin 19)** is connected to
    - the pin 13 in a 10-pin header pack
    - a LED (D1) with its limiting resistance (R2) to test the functionality of the microcontroller in the circuit (BUILT-IN LED)

▪ 2) to 9) all have the same use in our schematic which is working as an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). It can be configured as an input or output through software, and its state can be read or set using the appropriate I/O registers

20) AVCC **(pin 20)** is sensitive to power fluctuations which is crucial for providing power to the analog-to-digital converter (ADC) and some of the analog circuitry
    - to the +5v output of the linear regulator 7805 ensures a stable supply.
    - a polarized electrolytic capacitor(C1) which is connected directly to GND which is essential for filtering out any noise or voltage fluctuations in the power supply

21) AREF **(pin21)** is used to provide a reference voltage for the analog-to-digital converter (ADC), the technique used in our project to power it is to connect it with an external +5 volt by using a jumper from the Arduino UNO

22) PC0 **(pin 23)** is connected to the pin A0 in a 6-pin header pack
23) PC1 **(pin 24)** is connected to the pin A1 in a 6-pin header pack
24) PC2 **(pin 25)** is connected to the pin A2 in a 6-pin header pack
25) PC3 **(pin 26)** is connected to the pin A3 in a 6-pin header pack
26) PC4 **(pin 27)** is connected to
    - the pin A4 in a 6-pin header pack
    - the pin SDA in a 10-pin header pack to serve as the SDA line for I^{2}C communication
27) PC5 **(pin 28)** is connected to
    - the pin A5 in a 6-pin header pack
    - the pin SCL in a 10-pin header pack which is used for clocking data on the I^{2}C bus. It works in conjunction with the SDA pin (PC4)

  - 22) to 27) all have the same use in our schematic which is a 7-bit bi-directional ANALOG I/O port with internal pull-up resistors. It can be configured as an input or output through software, and its state can be read or set using the appropriate I/O registers

2. linear regulator 7805:

- powering the whole circuit of the microcontroller and the other external components that will be attached
- protection for the microcontroller as it stables the high voltage coming from the power supply, battery, and adaptor to a +5 voltage which is the maximum safest option for atmega328-p

3. battery terminal: to attach the battery or the adaptor to it

4. pin headers: To interface and connect with external components

5. ceramic capacitors: which are placed close to GND or VCC to work as decoupling capacitors to:

- filter out high-frequency noise from the power supply, which could otherwise interfere with the proper operation of the circuit
- stabilize the voltage supply by compensating for sudden drops or spikes in power consumption
- reduce electromagnetic interference (EMI) in the circuit by providing a short path to ground for high-frequency signals

6. electrolytic polarized capacitors:

- C1: filtering out any noise or voltage fluctuations in the power supply
- C2: minimize debouncing problem coming from the push button

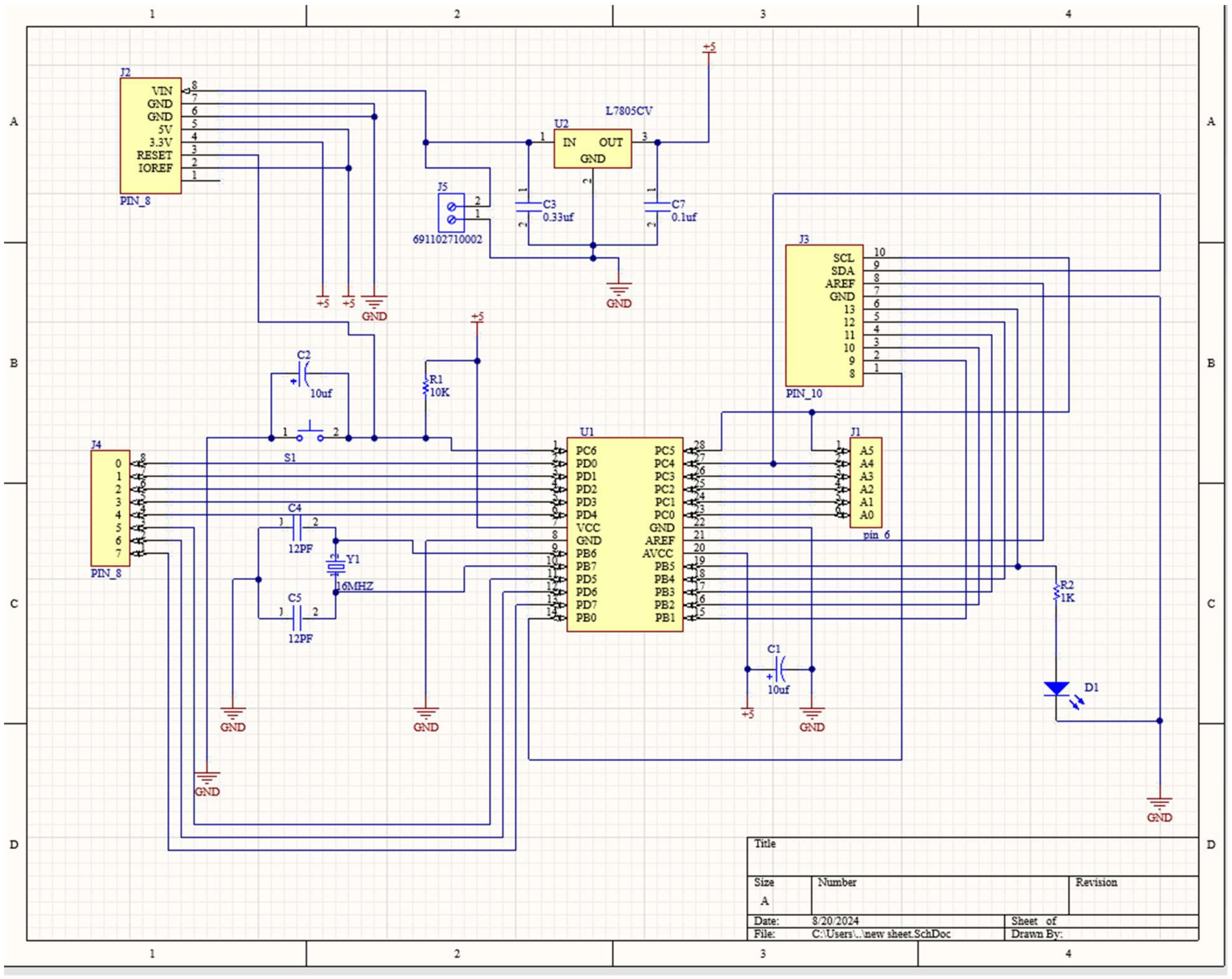
7. crystal oscillator:

- generates a stable clock signal that dictates the speed at which the microcontroller executes instructions. This clock frequency directly affects the timing of all operations, including loops, delays, and communication protocols.

8. led: to indicate that the circuit is working

9. resistance:

- R1: pullup resistance to keep the point at high voltage (not floating)
- R2: limiting current resistance to protect the diode



2. PCB layout and design:

The components were placed according to the logic, standards, rules, routing ability

- Battery terminal and regulator placed close together on the high left of the PCB board
- Pin-headers and other components were placed according to the shortest, most untangled path with the atmega328p

Number of layers: one layer (top)

Minimum clearance: 0.5mm

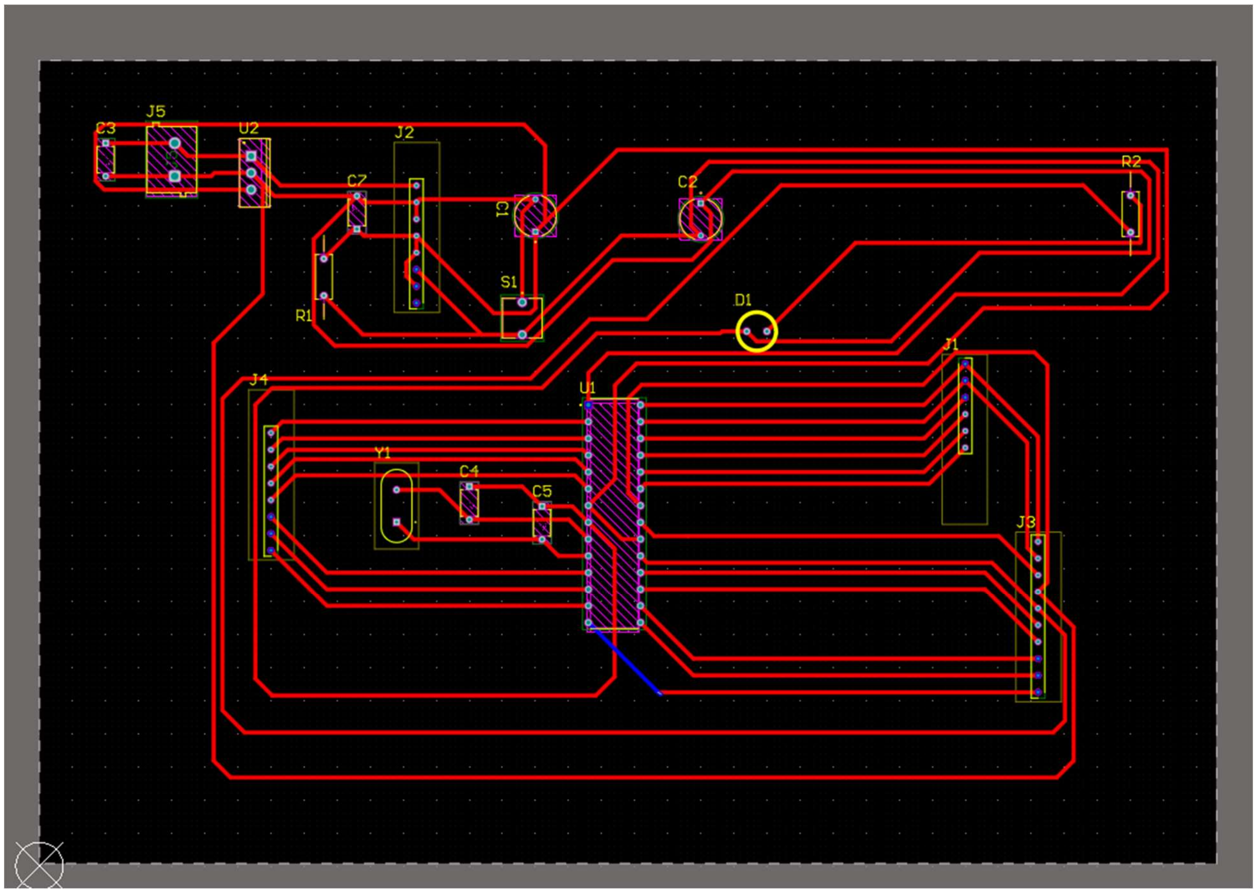
Average clearance: 0.5mm

Routing width (max, min, preferred): 0.8mm

Routing: was done automatically with manual checks and edits

Design rules: was checked and made sure all are logical

Dimensions: 12.2cm *17.8cm



3. symbols and footprint:

According to the datasheet and real-life measured components:

- Some were made manually and
- others were downloaded and imported from specific electronic websites (mouser, snapeda, etc), and then the required edits were made upon them

Indicators and sensors circuit

1.schematic:

1. voltage divider:

   were made internally in the circuit (not as a separate module) by using two resistances R4 and R5, the values for resistances allow a maximum voltage read of 28.5

   R4 is a 1k ohm resistance attached to:

   - Node common between measured ground and Arduino ground
   - The sensing point which is a pin (I-sens) that would be connected to the analog pin on the Arduino

   R5 is a 4.7k ohm resistance attached to:

   - Sensing point
   - A pin that will be connected to the measured voltage

2. LEDs (D1, D3, D4):

   Indicators for different aspects of the circuit

3. RGB LED(D1):

   Indicator for the speed, where each color pin is connected with a pin from a 10-pin header pack that will be connected to the pin-headers of the microcontroller, GND will be connected to the GND from the microcontroller circuit

4. Resistances (except the voltage divider ones):

   They are current limiting resistance to protect the indicator LEDs, their values were put according to the datasheet

5. Linear regulator 7805:

   Ensure that the voltage coming from the mc or going to it is a +5v

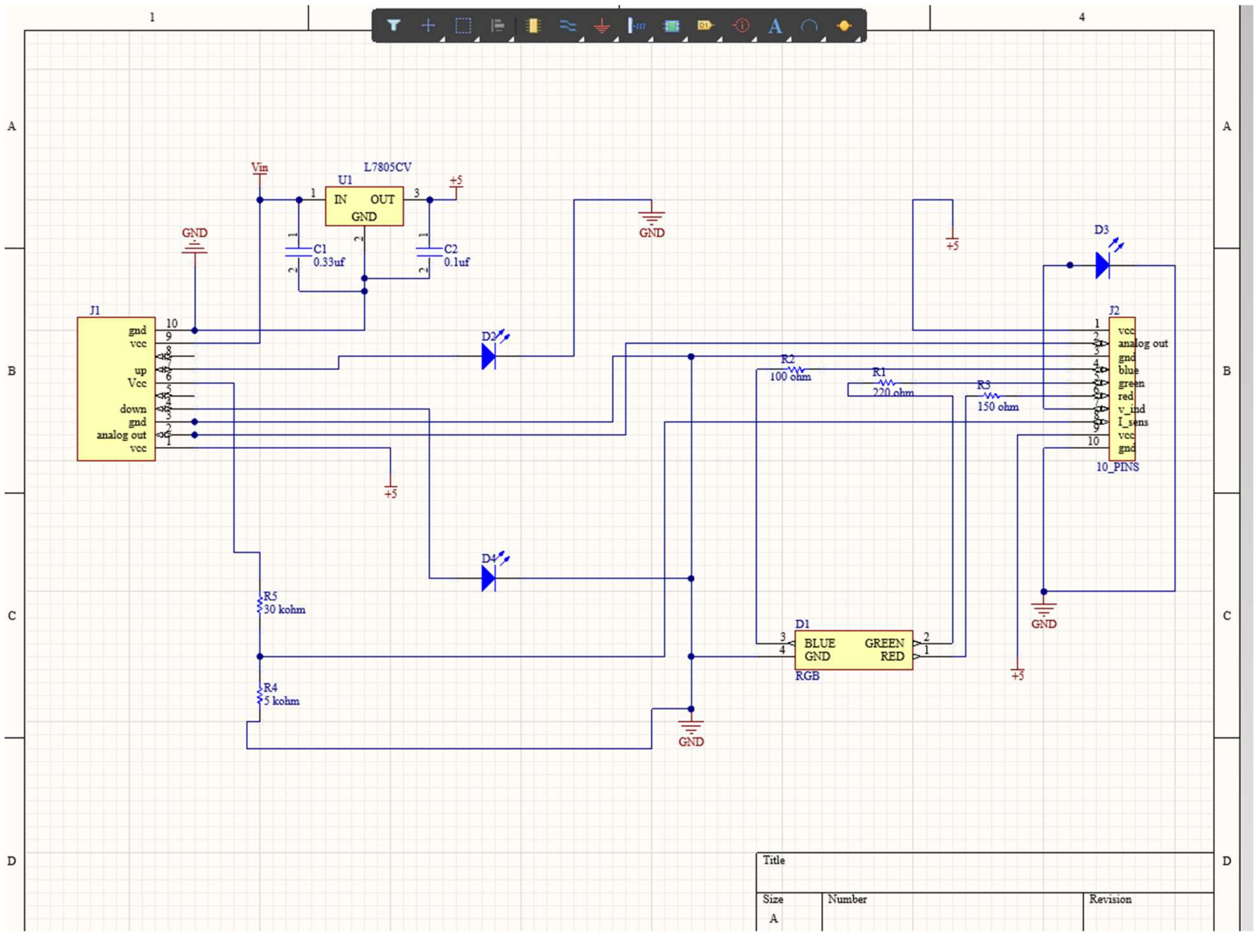
6. Ceramic capacitors (C1, C2):

   Decoupling capacitor for filtration

7. Pin-headers (J1, J2):

   2 sets of 10-pack pin headers are used for

   - Connection for interfacing with microcontroller pins
   - Additional ground and power pins for other external components



2. PCB layout and design:

The components were placed according to the logic, standards, rules, routing ability

- Battery terminal and regulator placed close together on the high left of the PCB board
- Pin-headers and other components were placed according to the shortest, most untangled paths

Number of layers: one layer (top)

Minimum clearance: 0.7mm

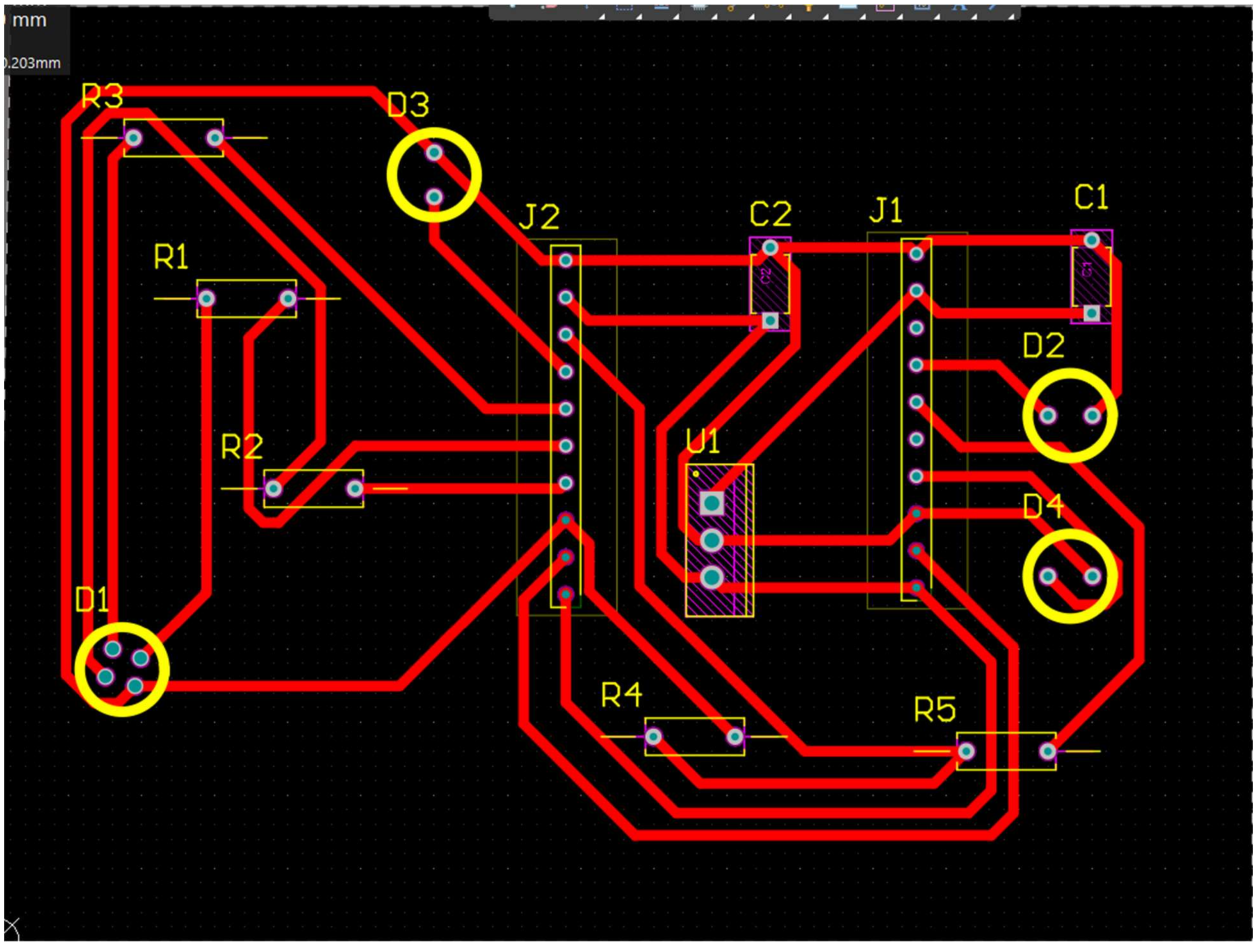
Average clearance: 0.7mm

Routing width (max, min, preferred): 0.8mm

Routing: was done automatically with manual checks and edits

Design rules: was checked and made sure all are logical

Dimensions: 8.5cm *6.5cm



3. symbols and footprint:

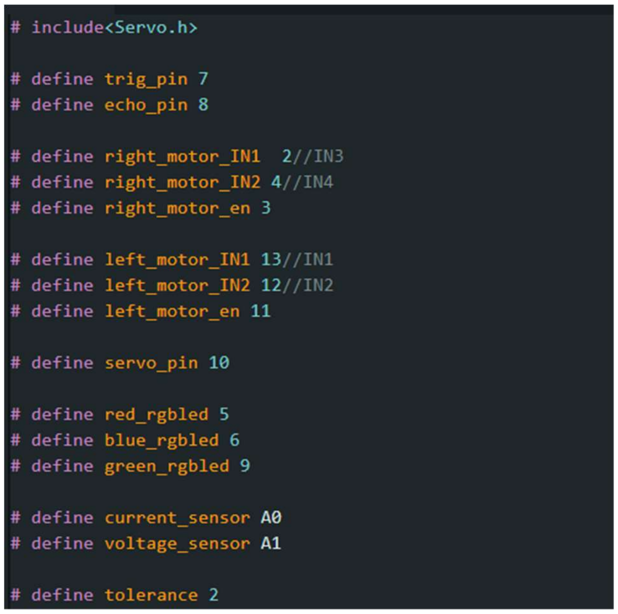
According to the datasheet and real-life measured components:

- Some were made manually
- others were downloaded and imported from specific electronic websites (mouser, snapeda, etc), and then the required edits were made upon them

2. Firmware

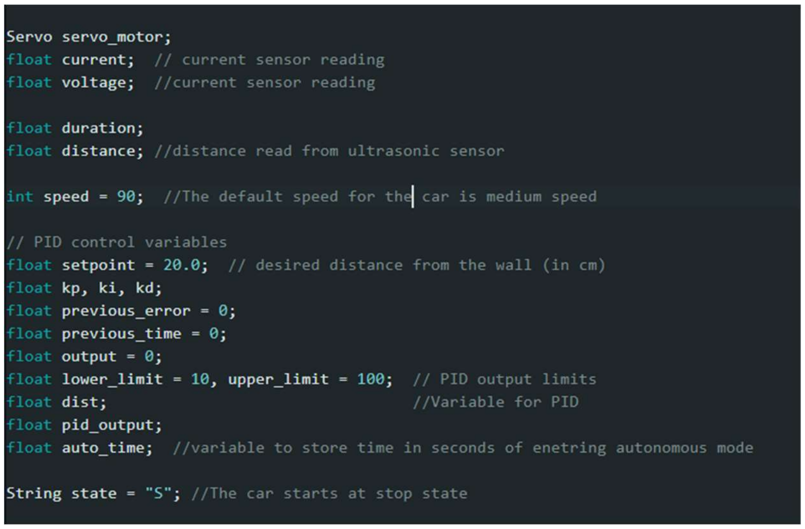
2.1 Arduino code

The code starts with declaring all constant values needed using #define to reduce usage of memory as possible and to hold the pin numbers of each component, the constants are distributed as following 3 constants for each motor 2 if them to specify the motor's direction and the third constant to specify its speed, 2 constants for the trigger and echo pin of the ultrasonic sensor ,3 constants for the RGB led one for each color and two constants for the current and voltage sensors.



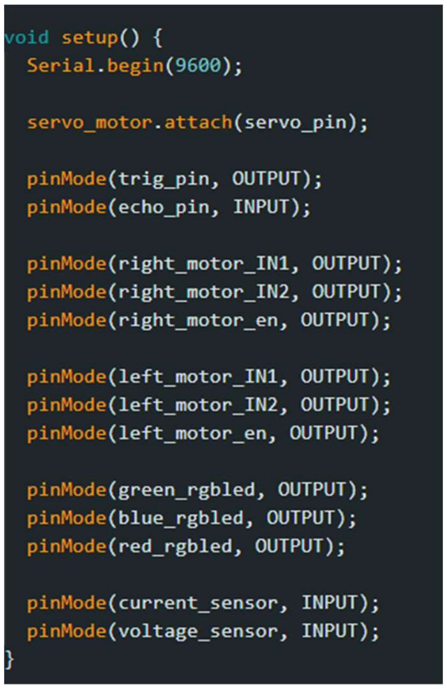
```
# include<Servo.h>

# define trig_pin 7
# define echo_pin 8

# define right_motor_IN1  2//IN3
# define right_motor_IN2 4//IN4
# define right_motor_en 3

# define left_motor_IN1 13//IN1
# define left_motor_IN2 12//IN2
# define left_motor_en 11

# define servo_pin 10

# define red_rgbled 5
# define blue_rgbled 6
# define green_rgbled 9

# define current_sensor A0
# define voltage_sensor A1

# define tolerance 2
```

The code then proceeds to declaring all variables with the needed datatypes.



```
Servo servo_motor;
float current;  // current sensor reading
float voltage;  //current sensor reading

float duration;
float distance; //distance read from ultrasonic sensor

int speed = 90;  //The default speed for the car is medium speed

// PID control variables
float setpoint = 20.0;  // desired distance from the wall (in cm)
float kp, ki, kd;
float previous_error = 0;
float previous_time = 0;
float output = 0;
float lower_limit = 10, upper_limit = 100;  // PID output limits
float dist;                                 //Variable for PID
float pid_output;
float auto_time;  //variable to store time in seconds of enetring autonomous mode

String state = "S"; //The car starts at stop state
```

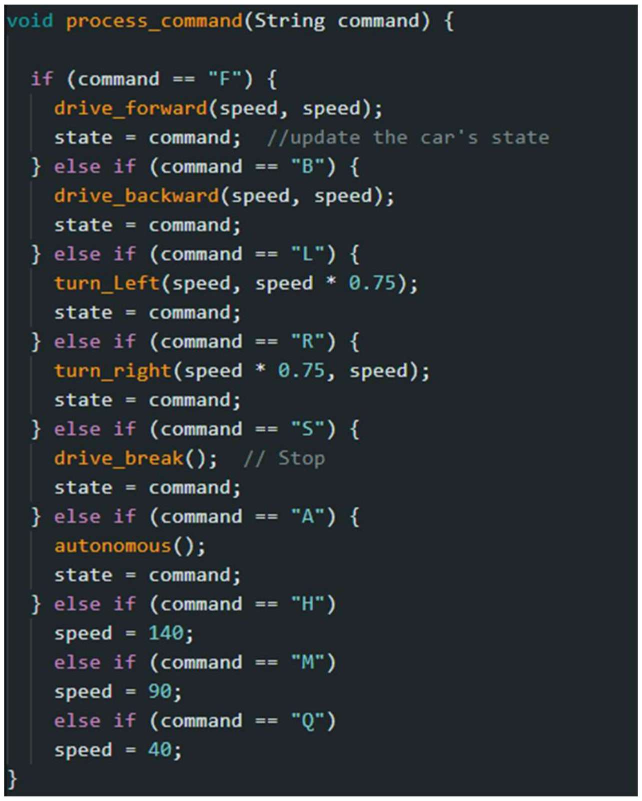
In the setup all commands that need to be made once are called as assigning INPUT/OUTPUT modes for each pin to serve its purpose and starting Serial connection to allow communication between microcontroller and the GUI user



```
void setup() {
  Serial.begin(9600);

  servo_motor.attach(servo_pin);

  pinMode(trig_pin, OUTPUT);
  pinMode(echo_pin, INPUT);

  pinMode(right_motor_IN1, OUTPUT);
  pinMode(right_motor_IN2, OUTPUT);
  pinMode(right_motor_en, OUTPUT);

  pinMode(left_motor_IN1, OUTPUT);
  pinMode(left_motor_IN2, OUTPUT);
  pinMode(left_motor_en, OUTPUT);

  pinMode(green_rgbled, OUTPUT);
  pinMode(blue_rgbled, OUTPUT);
  pinMode(red_rgbled, OUTPUT);

  pinMode(current_sensor, INPUT);
  pinMode(voltage_sensor, INPUT);
}
```

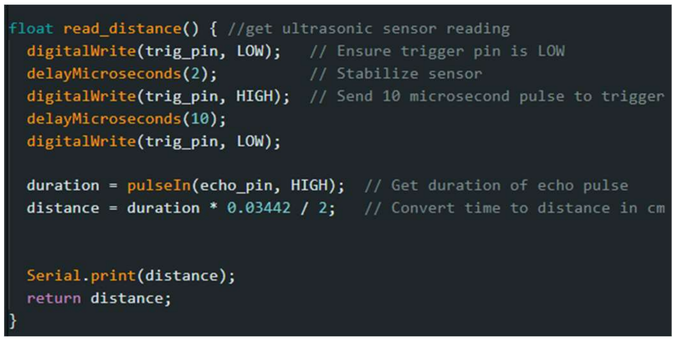
In order to understand the role of the loop function, we should first know other fundamental functions work.

The process_command function which is called each time any commands are given by the user, the process _command function uses a series of if-else statements to map the commands received to the corresponding functions to be called.

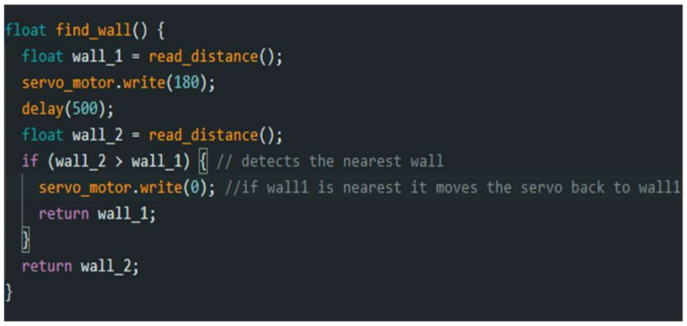


```
void process_command(String command) {

  if (command == "F") {
    drive_forward(speed, speed);
    state = command;  //update the car's state
  } else if (command == "B") {
    drive_backward(speed, speed);
    state = command;
  } else if (command == "L") {
    turn_Left(speed, speed * 0.75);
    state = command;
  } else if (command == "R") {
    turn_right(speed * 0.75, speed);
    state = command;
  } else if (command == "S") {
    drive_break();  // Stop
    state = command;
  } else if (command == "A") {
    autonomous();
    state = command;
  } else if (command == "H")
    speed = 140;
    else if (command == "M")
    speed = 90;
    else if (command == "Q")
    speed = 40;
}
```

The read distance function gets the ultrasonic reading which ranges from 2cm to 400 cm when called, this function is used later in find_wall function which is used to find the nearest wall and its distance in autonomus mode by using a servo to mount the ultrasonic sensor, the servo is set to be on the a side, then the distance to the wall is received then the servo moves the sensor to the other side and the distance to the second wall is stored, then the two reading are compared and the servo rotates to the nearest wall side and the car moves parallel to it



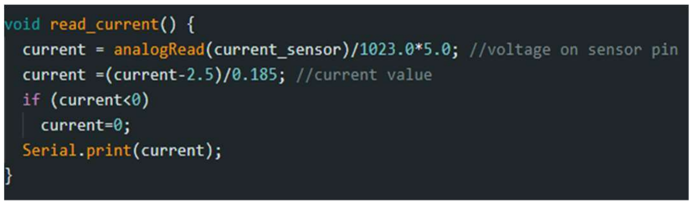
```
float read_distance() { //get ultrasonic sensor reading
  digitalWrite(trig_pin, LOW);   // Ensure trigger pin is LOW
  delayMicroseconds(2);          // Stabilize sensor
  digitalWrite(trig_pin, HIGH);  // Send 10 microsecond pulse to trigger
  delayMicroseconds(10);
  digitalWrite(trig_pin, LOW);

  duration = pulseIn(echo_pin, HIGH);  // Get duration of echo pulse
  distance = duration * 0.03442 / 2;   // Convert time to distance in cm


  Serial.print(distance);
  return distance;
}
```



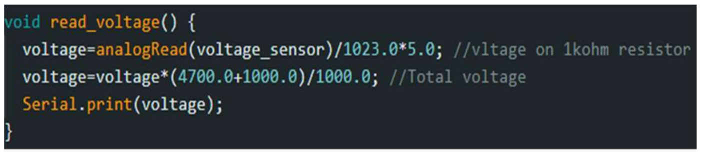
```
float find_wall() {
  float wall_1 = read_distance();
  servo_motor.write(180);
  delay(500);
  float wall_2 = read_distance();
  if (wall_2 > wall_1) { // detects the nearest wall
    servo_motor.write(0); //if wall1 is nearest it moves the servo back to wall1
    return wall_1;
  }
  return wall_2;
}
```

The read_current function reads the voltage on the current_sensor pin and applies it to a reversed equation to calculate the current reading. The equation: current= (voltage-2.5)/0.185 according to the ACS712 5A.

The read_voltage function reads the voltage on the voltage_sensor pin which is the voltage on a 1 kilo ohm which is connected in series to a 4.7 kilo ohm so by applying a reverse voltage divider we can get a voltage range of 0-28.5 volts, the voltage is found by applying the following equation Voltage = (1k ohm resistance voltage*(1k ohm+4.7k ohm)/1k ohm.

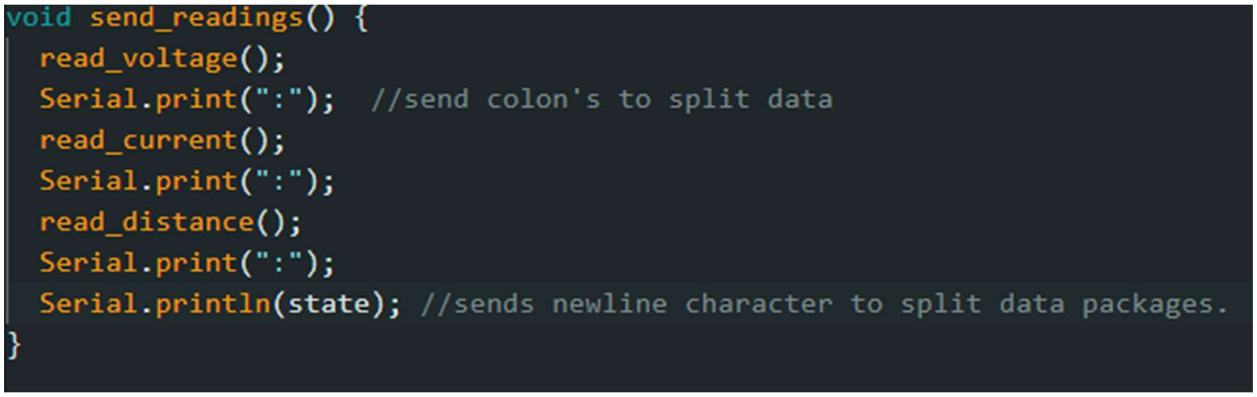


```
void read_current() {
  current = analogRead(current_sensor)/1023.0*5.0; //voltage on sensor pin
  current =(current-2.5)/0.185; //current value
  if (current<0)
    current=0;
  Serial.print(current);
}
```



```
void read_voltage() {
  voltage=analogRead(voltage_sensor)/1023.0*5.0; //vltage on 1kohm resistor
  voltage=voltage*(4700.0+1000.0)/1000.0; //Total voltage
  Serial.print(voltage);
}
```

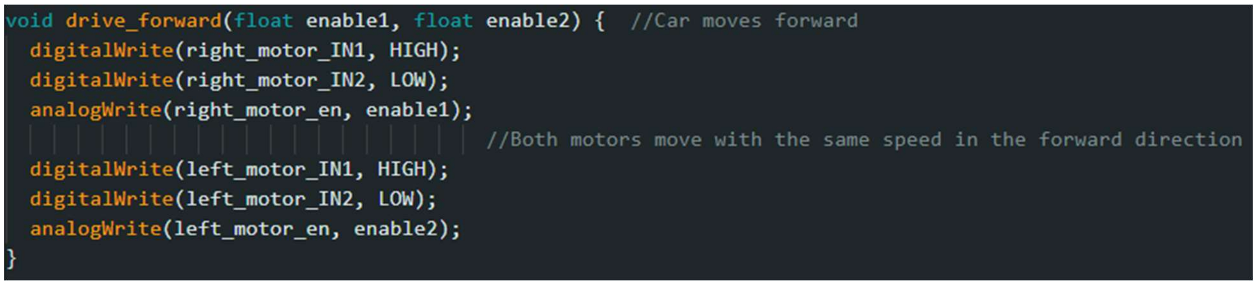
The send_readings function calls the 3 sensors functions ,read_voltage(), read_current(), read_distance(), sends their readings divided by a colon and then send the car state and prints a newline character '\n' to ensure data has a consistent format to be easily received on the user's end.



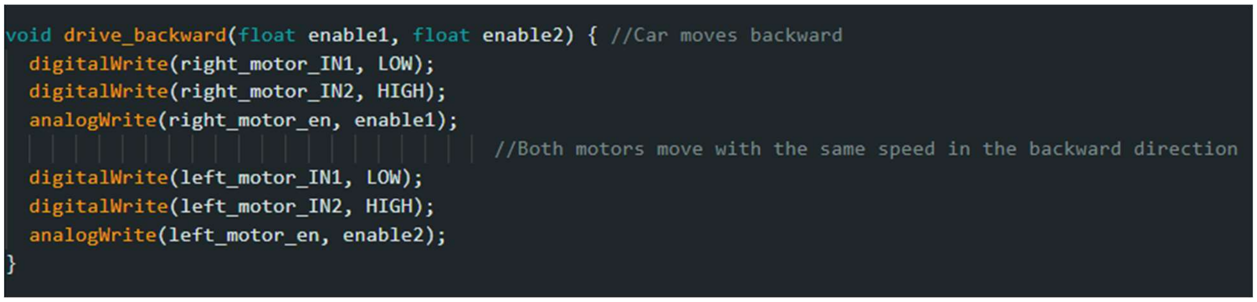
```
void send_readings() {
  read_voltage();
  Serial.print(":");  //send colon's to split data
  read_current();
  Serial.print(":");
  read_distance();
  Serial.print(":");
  Serial.println(state); //sends newline character to split data packages.
}
```

The drive_forward function moves the car in the forward direction by the speed set by the user either high , medium or low.

Similarly, the drive backward_function moves the car backward by the speed set by the user.

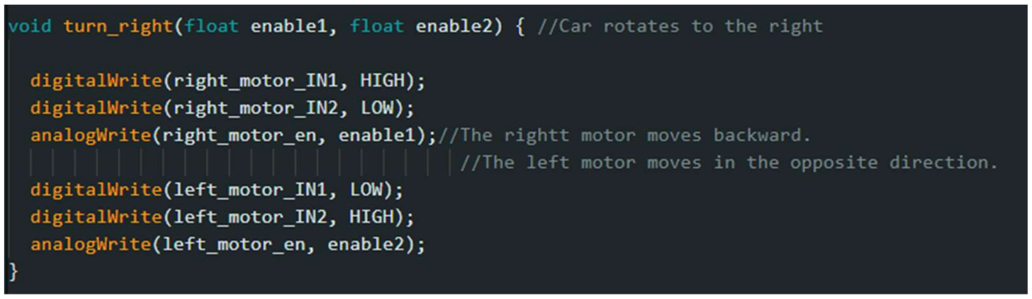


```
void drive_forward(float enable1, float enable2) {  //Car moves forward
  digitalWrite(right_motor_IN1, HIGH);
  digitalWrite(right_motor_IN2, LOW);
  analogWrite(right_motor_en, enable1);
                                        //Both motors move with the same speed in the forward direction
  digitalWrite(left_motor_IN1, HIGH);
  digitalWrite(left_motor_IN2, LOW);
  analogWrite(left_motor_en, enable2);
}
```

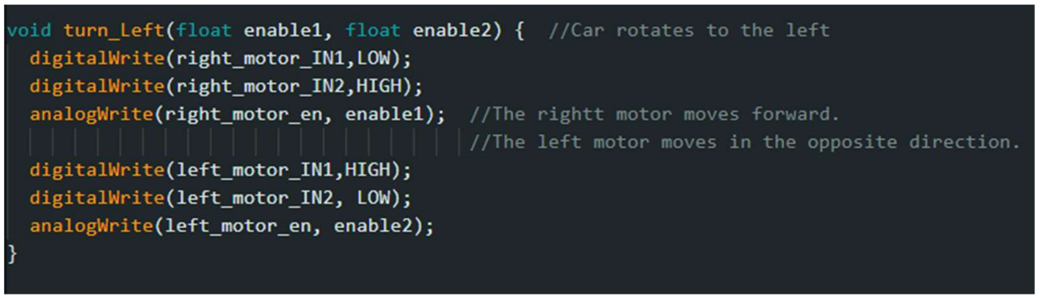


```
void drive_backward(float enable1, float enable2) { //Car moves backward
  digitalWrite(right_motor_IN1, LOW);
  digitalWrite(right_motor_IN2, HIGH);
  analogWrite(right_motor_en, enable1);
                                         //Both motors move with the same speed in the backward direction
  digitalWrite(left_motor_IN1, LOW);
  digitalWrite(left_motor_IN2, HIGH);
  analogWrite(left_motor_en, enable2);
}
```

The turn_right and turn_left functions operate by rotating the motors in 2 opposite direction to rotate the car clockwise or anti-clockwise.

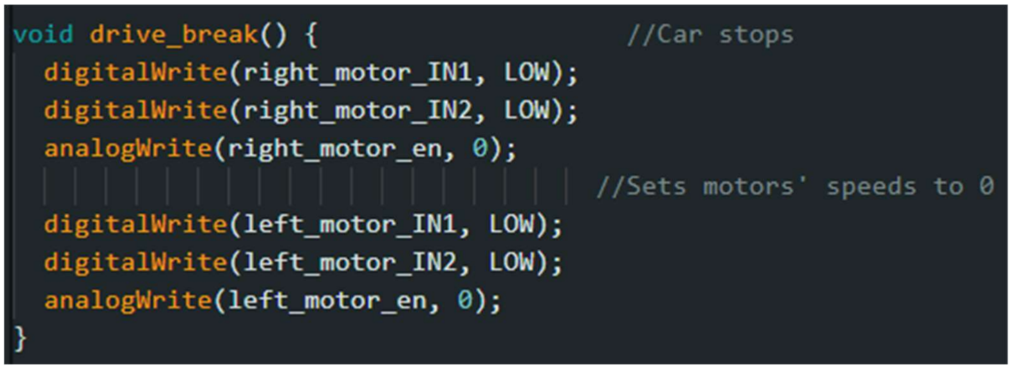


```
void turn_right(float enable1, float enable2) { //Car rotates to the right

  digitalWrite(right_motor_IN1, HIGH);
  digitalWrite(right_motor_IN2, LOW);
  analogWrite(right_motor_en, enable1);//The rightt motor moves backward.
                                       //The left motor moves in the opposite direction.
  digitalWrite(left_motor_IN1, LOW);
  digitalWrite(left_motor_IN2, HIGH);
  analogWrite(left_motor_en, enable2);
}
```



```
void turn_Left(float enable1, float enable2) {  //Car rotates to the left
  digitalWrite(right_motor_IN1,LOW);
  digitalWrite(right_motor_IN2,HIGH);
  analogWrite(right_motor_en, enable1);  //The rightt motor moves forward.
                                         //The left motor moves in the opposite direction.
  digitalWrite(left_motor_IN1,HIGH);
  digitalWrite(left_motor_IN2, LOW);
  analogWrite(left_motor_en, enable2);
}
```

The last motion related function is the drive_break function which stops rotation of the motors, it is also used to exit autonomous mode and enter manual mode.

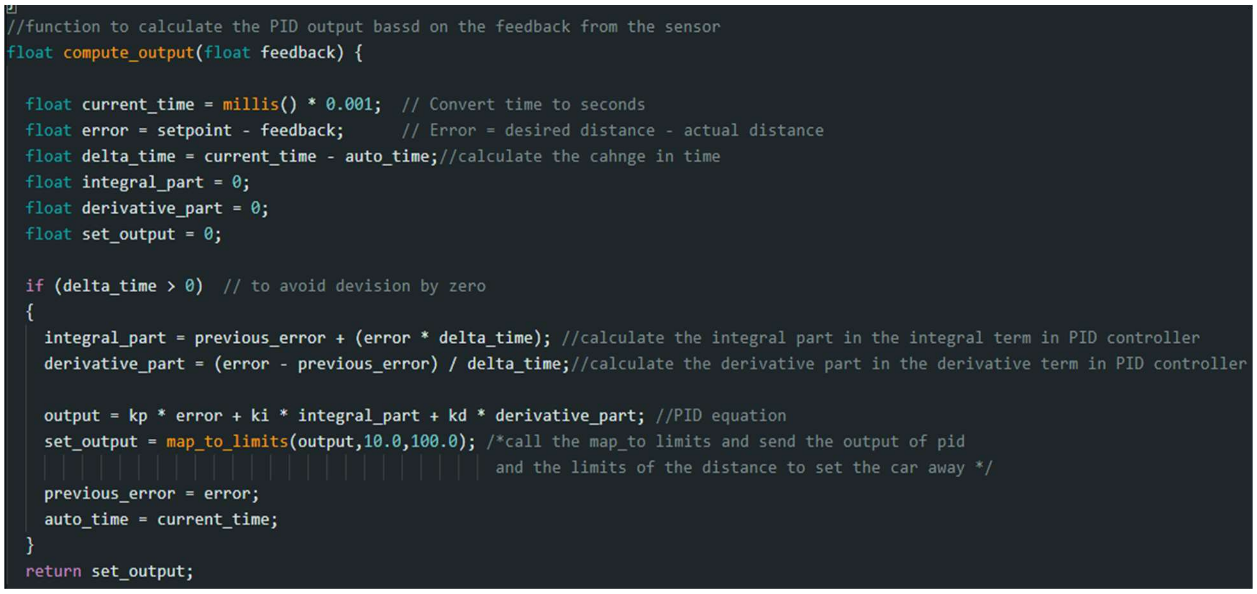


```
void drive_break() {                    //Car stops
  digitalWrite(right_motor_IN1, LOW);
  digitalWrite(right_motor_IN2, LOW);
  analogWrite(right_motor_en, 0);
                                      //Sets motors' speeds to 0
  digitalWrite(left_motor_IN1, LOW);
  digitalWrite(left_motor_IN2, LOW);
  analogWrite(left_motor_en, 0);
}
```

The compute_output function is for the calculation of the PID equation that takes the feedback from the ultrasonic sensor to calculate the error between the desired distance and the current distance from the wall then it calculate the PID equation that consist of three terms:

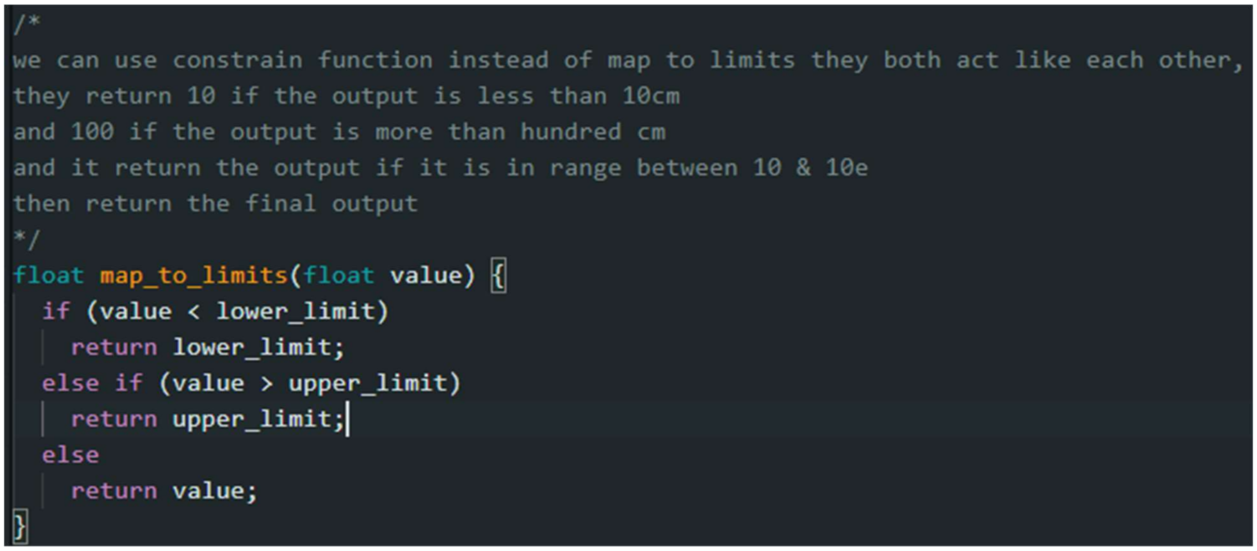
1. proportional term: the multiply of proportional gain (Kp)and the error
2. integral term: the multiply of integral gain(Ki) , error and the delta time.
3. derivative term: division of error to the change in time then multiply to the derivative gain (Kd).

After it calculates the output It calls the map_to_limits_function to set the limits as we explained in the documentation of the map_to_limits_function, then it returns the last output.



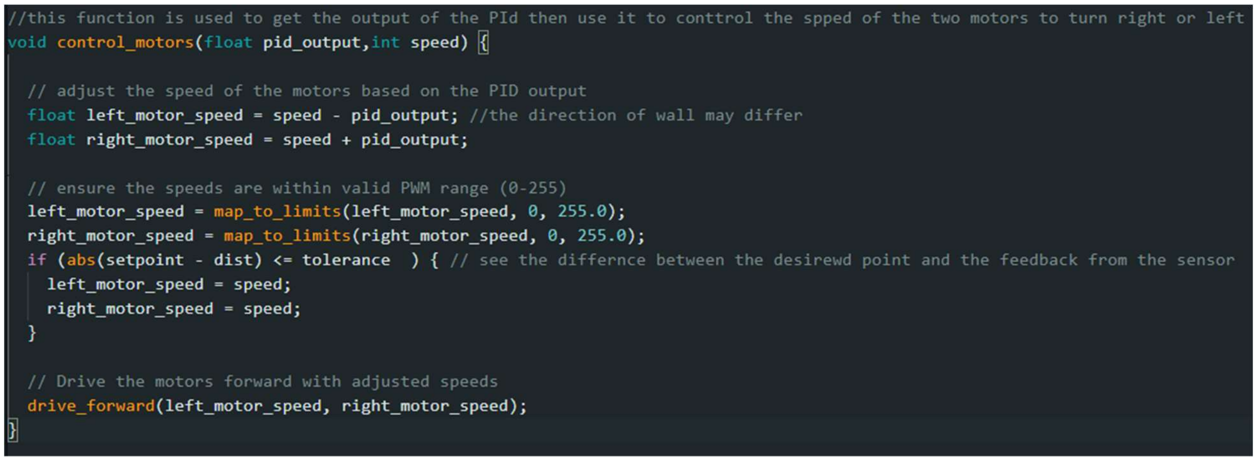
```
//function to calculate the PID output bassd on the feedback from the sensor
float compute_output(float feedback) {

  float current_time = millis() * 0.001;  // Convert time to seconds
  float error = setpoint - feedback;      // Error = desired distance - actual distance
  float delta_time = current_time - auto_time;//calculate the cahnge in time
  float integral_part = 0;
  float derivative_part = 0;
  float set_output = 0;

  if (delta_time > 0)  // to avoid devision by zero
  {
    integral_part = previous_error + (error * delta_time); //calculate the integral part in the integral term in PID controller
    derivative_part = (error - previous_error) / delta_time;//calculate the derivative part in the derivative term in PID controller

    output = kp * error + ki * integral_part + kd * derivative_part; //PID equation
    set_output = map_to_limits(output,10.0,100.0); /*call the map_to limits and send the output of pid
                                                    and the limits of the distance to set the car away */
    previous_error = error;
    auto_time = current_time;
  }
  return set_output;
```

The map_to_limits function is used to set limits to the output of the PID controller to ensure that it is in the range that we want and it takes three variables, the lower limit and the upper limit for the range and the value of the output of the PID



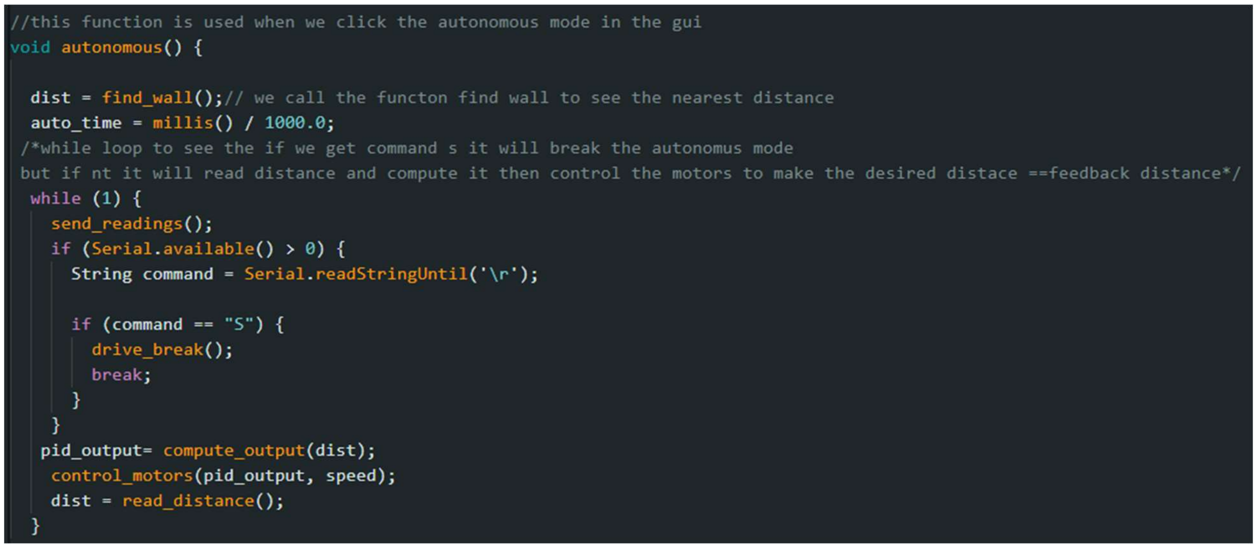
```
/*
we can use constrain function instead of map to limits they both act like each other,
they return 10 if the output is less than 10cm
and 100 if the output is more than hundred cm
and it return the output if it is in range between 10 & 10e
then return the final output
*/
float map_to_limits(float value) {
  if (value < lower_limit)
    return lower_limit;
  else if (value > upper_limit)
    return upper_limit;
  else
    return value;
}
```

The control_motors function is used to get the output of the PId then use it to control the sped of the two motors to turn right or left, to make the car move parallel to the wall from a fixed distance and the if condition in it is to detect if the difference between the desired distance and the current distance is less than the tolerance of the sensor that is set to 2 as we try to read the distance from the ultrasonic sensor in tinckercad and we found that after we adjust the speed of sound to be 0.3442 and we reached a small error in the read . Then it sees if the difference is less than the tolerance it will make the motor move forward and parallel to the wall If not it will make the car turn right or left to reach the desired distance.



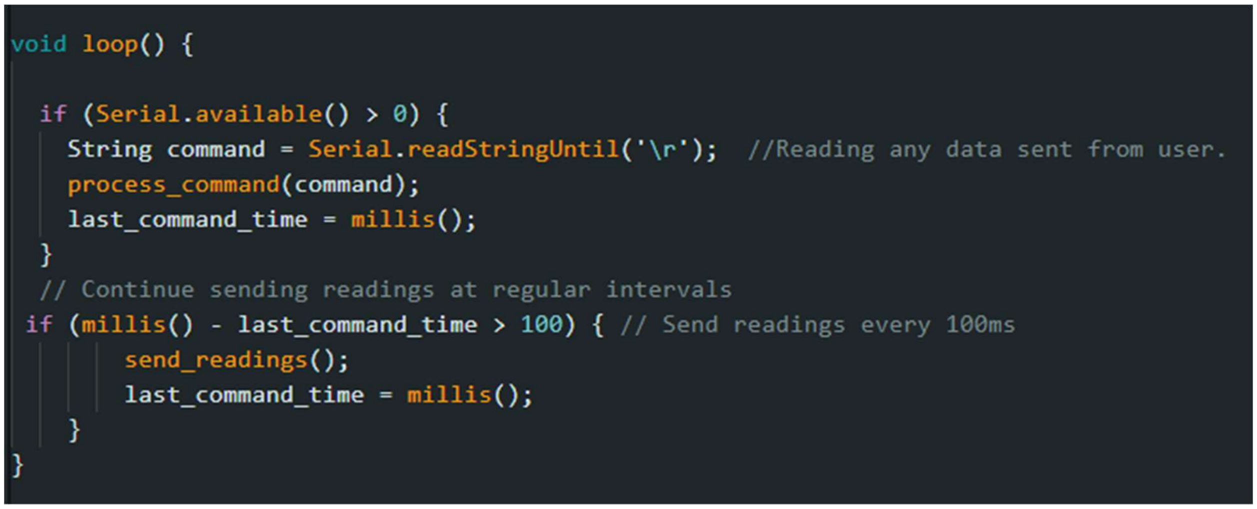
```
//this function is used to get the output of the PId then use it to conttrol the spped of the two motors to turn right or left
void control_motors(float pid_output,int speed) {

  // adjust the speed of the motors based on the PID output
  float left_motor_speed = speed - pid_output; //the direction of wall may differ
  float right_motor_speed = speed + pid_output;

  // ensure the speeds are within valid PWM range (0-255)
  left_motor_speed = map_to_limits(left_motor_speed, 0, 255.0);
  right_motor_speed = map_to_limits(right_motor_speed, 0, 255.0);
  if (abs(setpoint - dist) <= tolerance  ) { // see the differnce between the desirewd point and the feedback from the sensor
    left_motor_speed = speed;
    right_motor_speed = speed;
  }

  // Drive the motors forward with adjusted speeds
  drive_forward(left_motor_speed, right_motor_speed);
}
```

The autonomous function function is used when we click the autonomous mode in the GUI, it first sets the servo direction to the nearest wall using find_wall function and returns the distance between this wall and the car then it will see if it gets the Stop command it will break and exit the function if not it will compute the feedback and control the motors to move parallel to the wall.



```
//this function is used when we click the autonomous mode in the gui
void autonomous() {

  dist = find_wall();// we call the functon find wall to see the nearest distance
  auto_time = millis() / 1000.0;
 /*while loop to see the if we get command s it will break the autonomus mode
 but if nt it will read distance and compute it then control the motors to make the desired distace ==feedback distance*/
  while (1) {
    send_readings();
    if (Serial.available() > 0) {
      String command = Serial.readStringUntil('\r');

      if (command == "S") {
        drive_break();
        break;
      }
    }
   pid_output= compute_output(dist);
    control_motors(pid_output, speed);
    dist = read_distance();
  }
```

After understanding the fundamental functions we can understand the loop function role which is the key to switching the car’s modes and states were the user’s commands are read in it and processed using process_command function by calling the required functions and the loop function ends by send_readings function to ensure updated data are always available for the user.



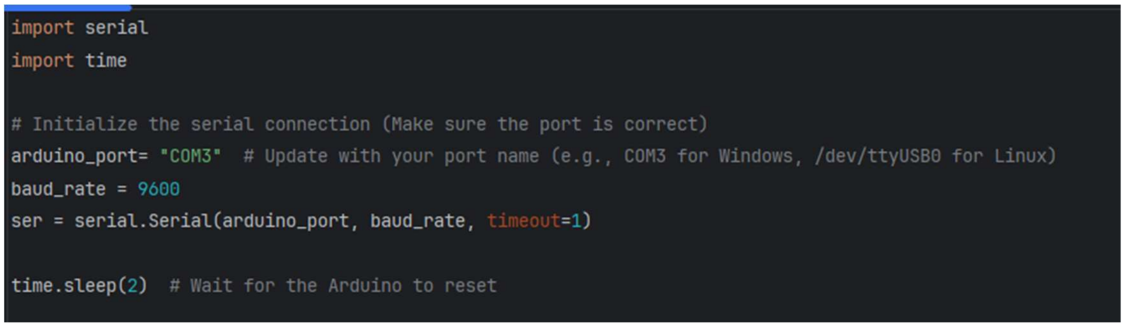
```
void loop() {

  if (Serial.available() > 0) {
    String command = Serial.readStringUntil('\r');  //Reading any data sent from user.
    process_command(command);
    last_command_time = millis();
  }
  // Continue sending readings at regular intervals
 if (millis() - last_command_time > 100) { // Send readings every 100ms
        send_readings();
        last_command_time = millis();
    }
}
```

2.1 Python code

The python uses Pyserial library to allow communication between the microcontroller and the GIU over Bluetooth.

The code begins with improting the Pyserial library and time library which is used to add delays in the code to allow initialization of connection.

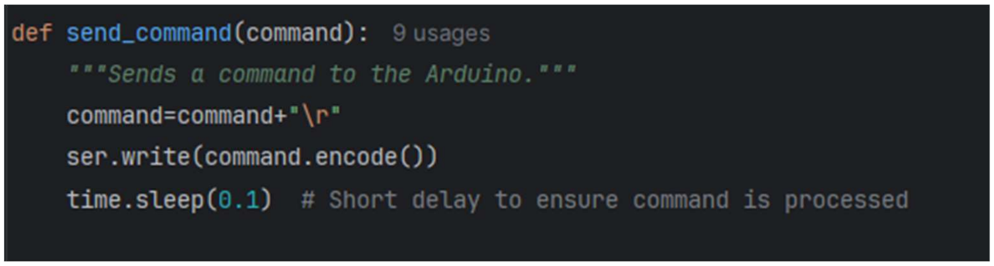
The first event that occurs is creating an instance of Serial class giving it the necessary arguments which are the COM port of the Bluetooth and the baudrate which is the number of bytes sent per second to be compatible with the Bluetooth module.



```python
import serial
import time

# Initialize the serial connection (Make sure the port is correct)
arduino_port= "COM3"  # Update with your port name (e.g., COM3 for Windows, /dev/ttyUSB0 for Linux)
baud_rate = 9600
ser = serial.Serial(arduino_port, baud_rate, timeout=1)

time.sleep(2)  # Wait for the Arduino to reset
```

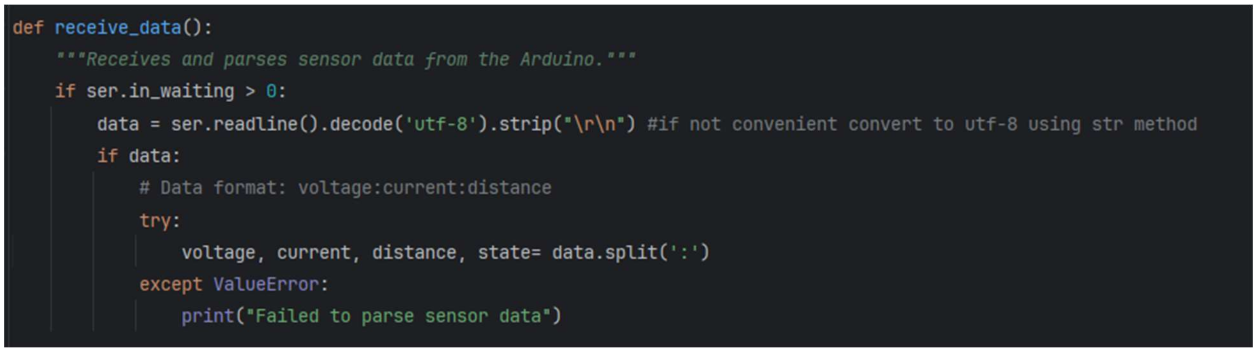
The code contains 2 main functions which are send_command and the receive_command.

The send_command function receives one argument which is the command letter and and adds a carriage return character which is used to prevent conflict of commands then decodes the command and transmits it



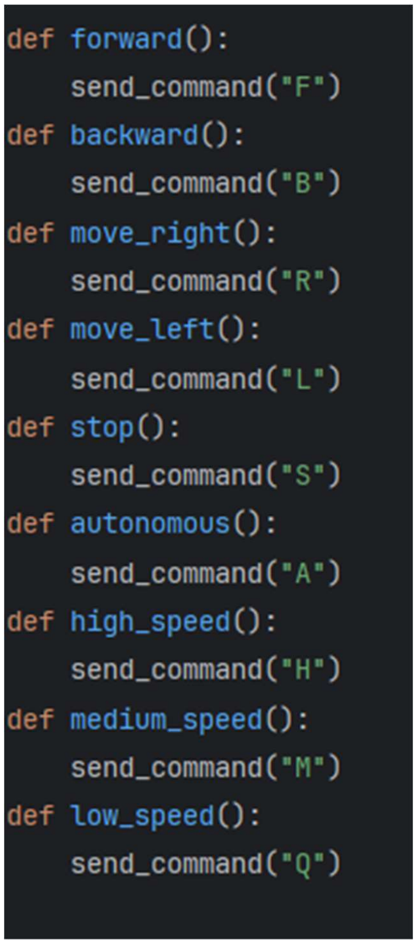
```python
def send_command(command):  9 usages
    """Sends a command to the Arduino."""
    command=command+"\r"
    ser.write(command.encode())
    time.sleep(0.1)  # Short delay to ensure command is processed
```

The receive_data function is used to check if any reading are sent from the mc and unpacks the data and adds each reading to its corresponding variable



```python
def receive_data():
    """Receives and parses sensor data from the Arduino."""
    if ser.in_waiting > 0:
        data = ser.readline().decode('utf-8').strip("\r\n") #if not convenient convert to utf-8 using str method
        if data:
            # Data format: voltage:current:distance
            try:
                voltage, current, distance, state= data.split(':')
            except ValueError:
                print("Failed to parse sensor data")
```

All other functions are used to provide the unique command character for the needed command


```
def forward():
    send_command("F")
def backward():
    send_command("B")
def move_right():
    send_command("R")
def move_left():
    send_command("L")
def stop():
    send_command("S")
def autonomous():
    send_command("A")
def high_speed():
    send_command("H")
def medium_speed():
    send_command("M")
def low_speed():
    send_command("Q")
```

It is decided not to use this python file (car_control.py) which is uploaded to the repository to avoid an error that prevents access to the mc COM port instead all function are written inside the main GUI file.

3.SOFTWARE

Computer vision

1-video stitching

The video stitching system integrates video processing and graphical user interface (GUI) components to perform real-time video stitching of left and right video feeds.

We use :

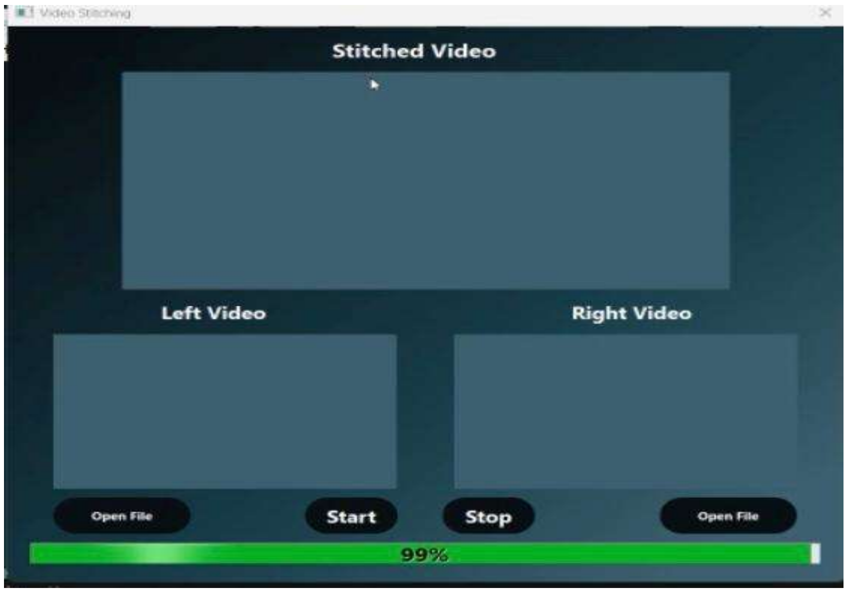
PyQt6 GUI: For user interaction, allowing users to select video files and control stitching operations.

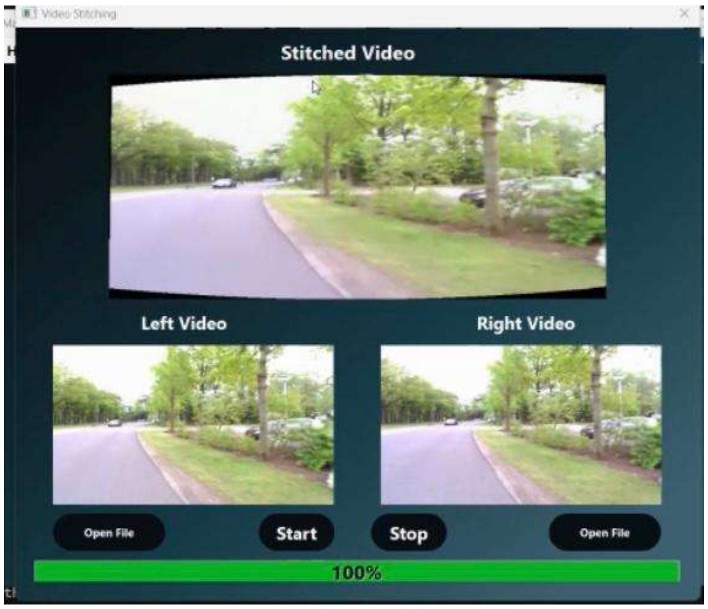
OpenCV: For video processing, including reading video files, stitching frames, and saving the output.

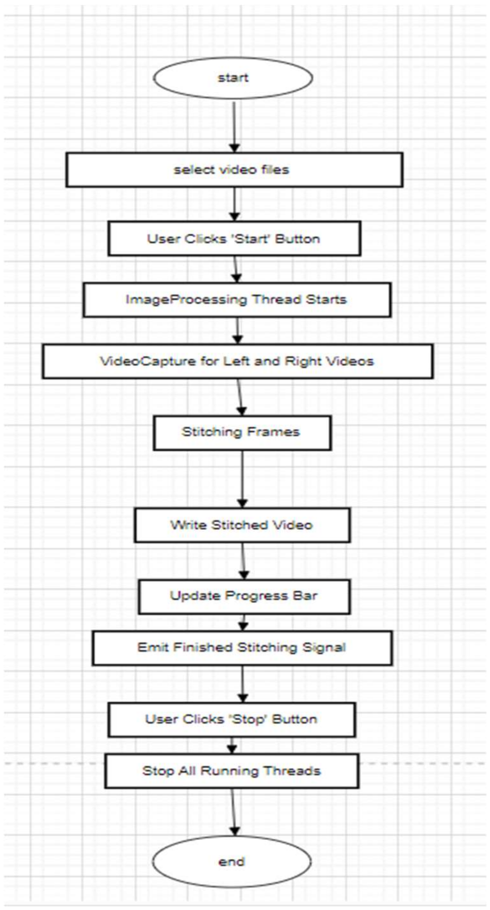
Multithreading: To handle video processing

Algorithm :

- **Input**: Two video files (left and right views).
- **Process**: Stitches frames from both videos into a single panoramic view.
- **Output**: A stitched video file




Video Stitching
Stitched Video
Left Video
Right Video
Open File
Start
Stop
Open File
100%


start
select video files
User Clicks 'Start' Button
ImageProcessing Thread Starts
VideoCapture for Left and Right Videos
Stitching Frames
Write Stitched Video
Update Progress Bar
Emit Finished Stitching Signal
User Clicks 'Stop' Button
Stop All Running Threads
end

2-stereo vision

Stereo vision is a technique for inferring the 3D structure of a scene from two images taken from slightly different viewpoints.

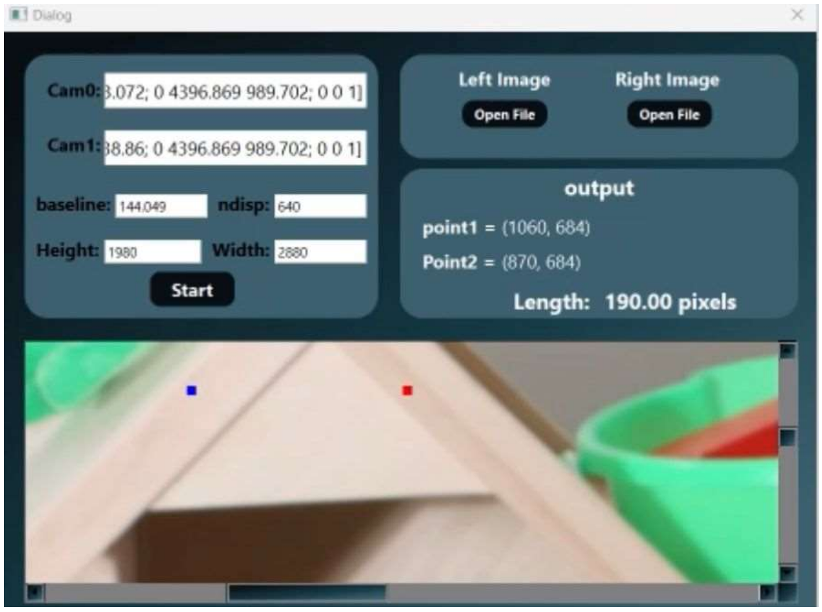
We Use:

the SIFT (Scale-Invariant Feature Transform) algorithm to detect keypoints and compute descriptors in both images.We Match features between the two images using the BFMatcher (Brute Force Matcher) with the L2 norm, Filter out outlier matches using the mask returned by the RANSAC process. Initialize the StereoSGBM (Semi-Global Block Matching) algorithm with appropriate parameters, including the number of disparities, block size, and uniqueness ratio.

Compute the disparity map using the rectified images. The disparity map represents the pixel difference between corresponding points in the two images.

Algorithm :

- **Input**: Two image files (left and right views), width ,height ,baseline and camera matrices
- **Process**: Plotting the epipolar lines on both images along with feature points, save the disparity as a gray scale and color image using heat map conversion and save the depth image as a gray scale and color image using heat map conversion.
- **Output**: in GUI the image will appear and you can know the distance between any 2 points



Cam0: 3.072; 0 4396.869 989.702; 0 0 1]

Cam1: 38.86; 0 4396.869 989.702; 0 0 1]

baseline: 144.049 ndisp: 640

Height: 1980 Width: 2880

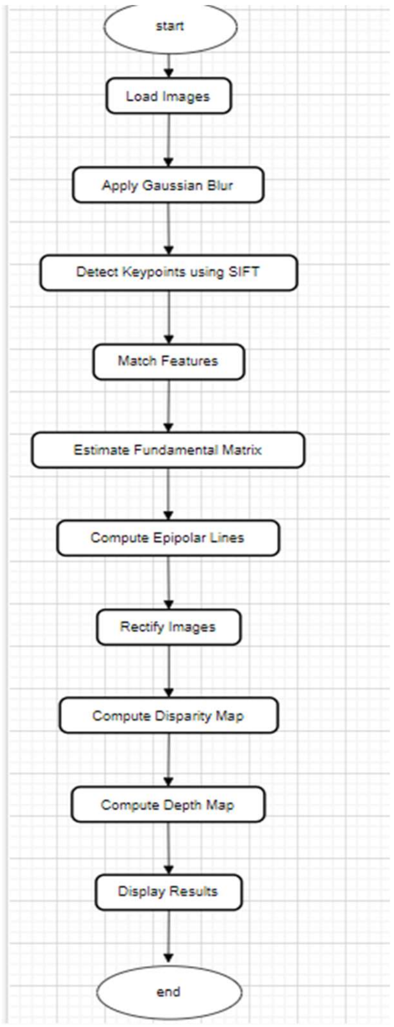
output

point1 = (1060, 684)

Point2 = (870, 684)

Length: 190.00 pixels

Flow chart:



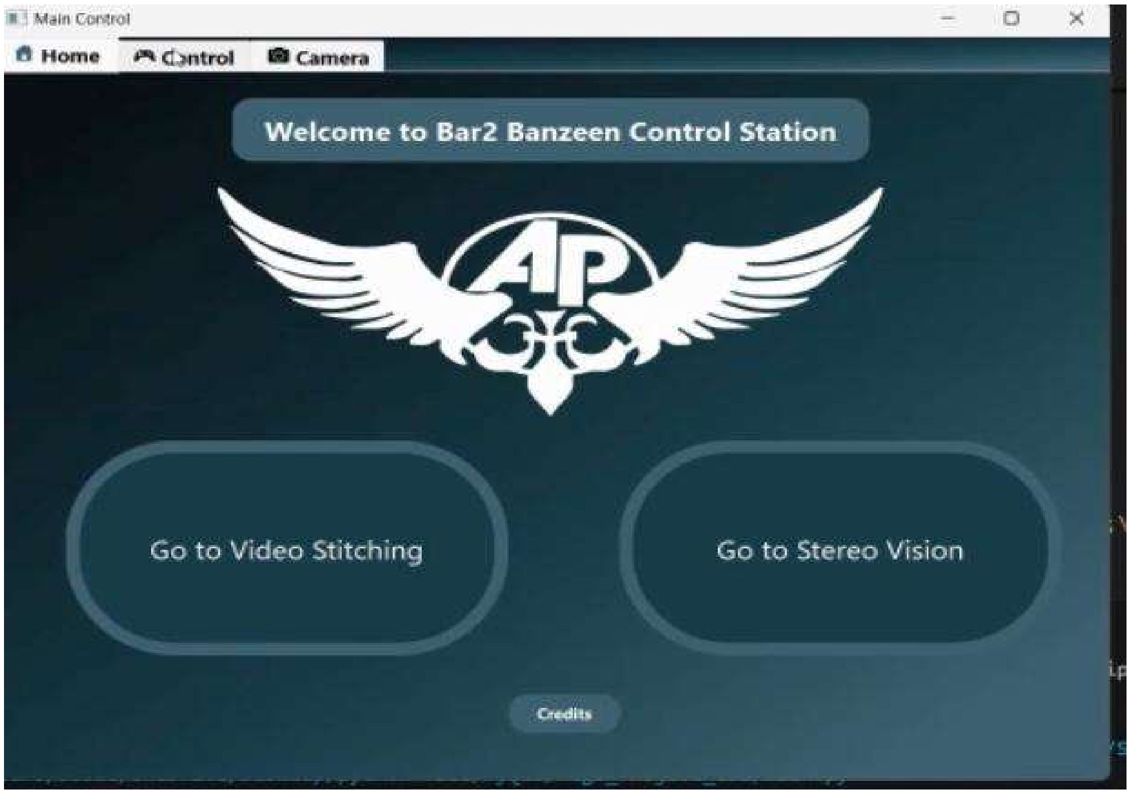
GUI:

Mainwindow:

We use tab widget in each window, The GUI features a central QTabWidget with three tabs: Home , Control and camera. Each tab contains styled buttons and labels arranged in frames with rounded corners. The color scheme involves shades of blue with white text.

Home Tab :

includes buttons for "Video Stitching" and "Stereo Vision," and displays the team logo.



Control tab :

1.Mode Selection

Autonomous and Manual Mode: allowing the user to choose between autonomous operation (where the system operates independently) and manual operation (where the user has direct control).

2. Speed Control

- **High, Medium, Low Speed Settings**: There are three buttons which likely control the speed of the system. The user can select one of these buttons to adjust the operating speed.

3. Controller Selection

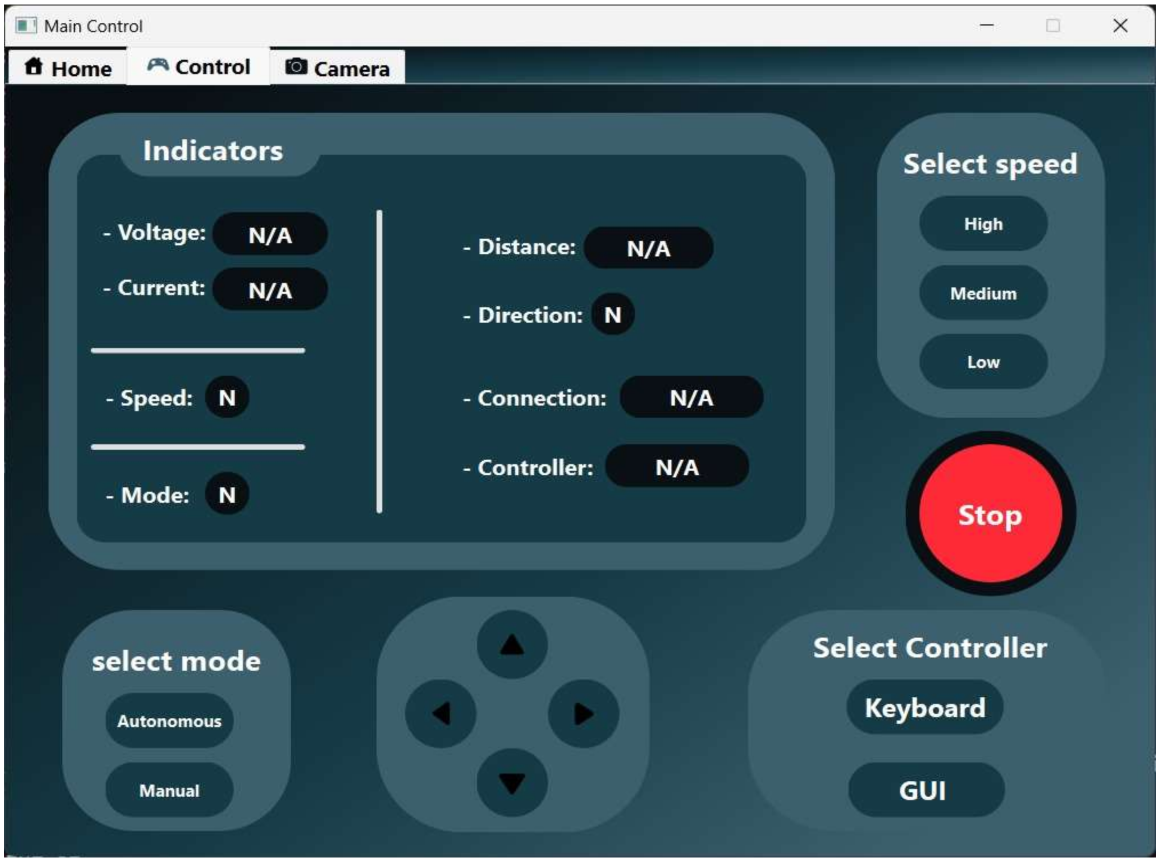
- **Keyboard and GUI Options**: There are two buttons labeled "Keyboard" and "GUI." These allow the user to choose the control method

4. Movement Controls

- **Directional Control Buttons**: There are four directional buttons arranged in a circular pattern, which are used to control the movement of the vehicle . These include buttons for moving forward, backward, left, and right.

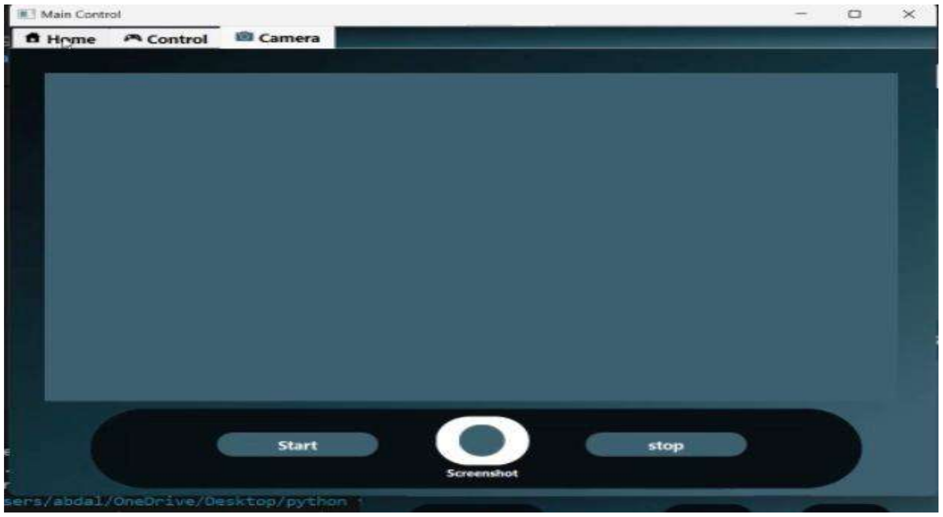
5. Additional Indicators and Information

- **Indicators Panel**: This section provides information about various parameters such as voltage, current, speed, mode, distance, direction, connection status, and the type of controller in use. It’s designed to keep the user informed about the status and performance of the system.
- **Stop Button**: A prominent red button labeled "Stop" is also present to stop the vehicle

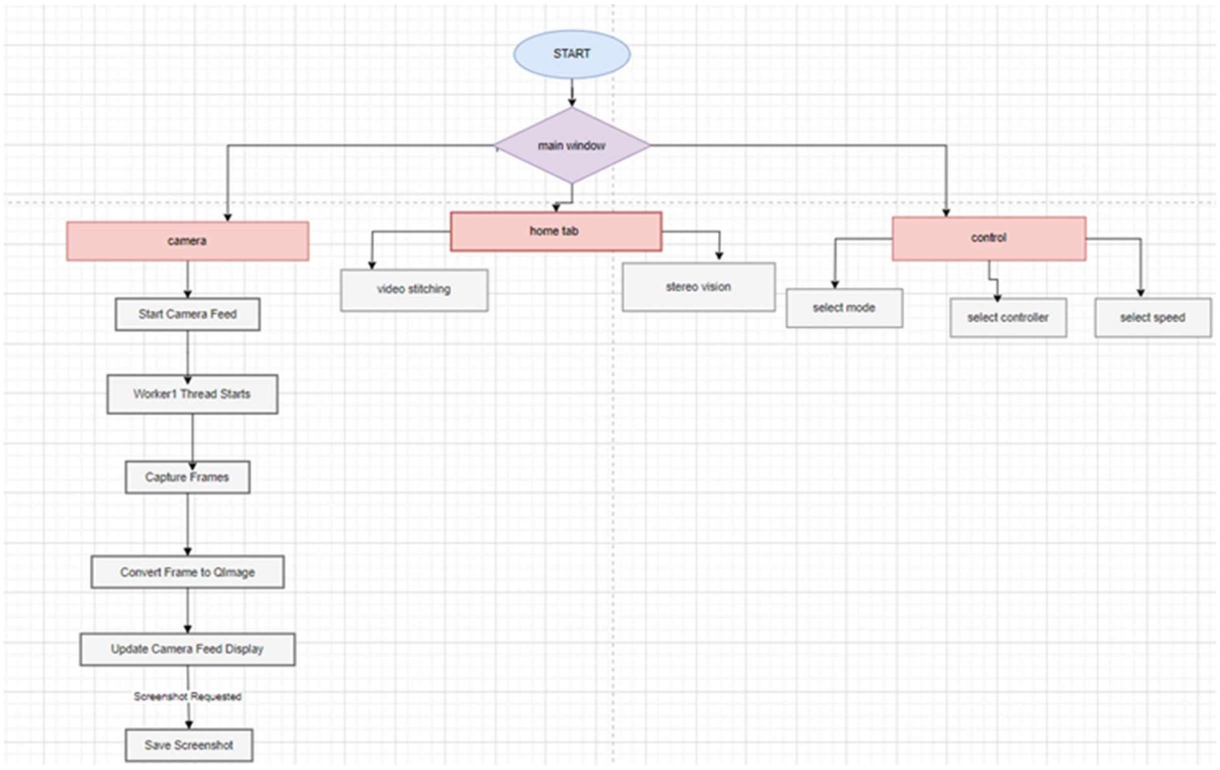


Camera Tab:

Displays the camera feed. Contains buttons for starting and stopping the camera feed and a screenshot button



Flow chart of the backended GUI:



We use thread for multi-core systems, threads can execute in parallel, which can lead to performance improvements.

The Worker1 class illustrates how to implement a thread safely using QThread.

QThread: This class is a subclass of QThread, which serves as a foundation for running concurrent tasks.

Clean Shutdown: The stop() method safely signals the thread to finish its operations before quitting. This is vital for resource management and preventing memory leaks.

Sub-window :

1)video stitching window :

- The buttons leftOpen_pushButton and rightOpen_pushButton allow the user to select video files from their system.When a file is selected, the path is stored in the instance variables self.left_path and self.right_path.
- When the user clicks the start_pushButton, the start_videoStitching() method is triggered. This method checks if both video paths are selected and initiates the video stitching process.

- ImageProcessing class runs the stitching algorithm in a separate thread to ensure the GUI remains responsive.
- The progress of the stitching is visualized using a progress bar (progressBar), updated dynamically through signals to avoid blocking the main thread.

Upon completion, the stitched video path is passed to the loop_videos() method, initiating playback.

- After stitching, the videos are displayed in their respective panels (leftVideo_label, rightVideo_label, and stitchedVideo_label).
- VideoLooper class is employed to loop each video in a separate thread, ensuring smooth playback.
- Each frame is processed and converted to a QImage before being displayed in the corresponding QLabel widget. This is done asynchronously, maintaining real-time playback and synchronization with the original video frame rate.
- The stop_pushButton is connected to the stop_videoStitching() method, which stop all active threads.This includes stopping the stitching process (if still running) and all video playback loops, effectively allowing the user to pause the operation at any time.

2)Stereo vision window:

- Buttons: Two buttons, leftOpen_stereo_pushButton and rightOpen_stereo_pushButton, are used for selecting the left and right stereo images.

Implementation: The getFileName_left and getFileName_right methods open a file dialog to allow users to select the images. The paths of the selected images are stored in self.left_path and self.right_path.

- Start Button: The start_stereo_pushButton triggers the stereo vision processing.

Implementation: When the start button is clicked, the start_stereo_vision method creates an instance of the StereoVision_processing class and starts the processing in a separate thread. This class handles tasks like:Loading images and applying Gaussian blur to reduce noise.

- Extracting and matching keypoints using SIFT and BFMatcher, Estimating the Fundamental matrix, decomposing the Essential matrix, and rectifying images.
- Emitting signals to update the GUI with the processed results, such as displaying epipolar lines, disparity maps, and depth maps.
- Dynamic Resizing: The size of the image display area (QLabel) is dynamically resized based on user input for width and height using the update_frame_and_label_size method.
- Mouse Interaction: The get_mouse_position method captures mouse click events on the image and calculates the corresponding pixel coordinates in the original image. It also fetches the Z-value (depth) from the depth map for the clicked point.

Users can select two points on the stereo image. The first point is stored in self.point1, and the second point is stored in self.point2. Once both points are selected, the calculate_distance   method calculates the Euclidean distance between the points in pixel units and displays it in the GUI.

- Plotting Results: Epipolar Lines, Disparity, and Depth Maps: After processing, the display_plots method displays the epipolar lines, disparity maps, and depth maps using Matplotlib.

Integration: The StereoVision_processing class emits the Plot_signal signal, which triggers the display_plots method to visualize these results in the GUI.

- Camera Parameters and Baseline Input:

The camera intrinsic matrices (K1 and K2), baseline distance, image width, height, and the number of disparities are set by the user through text fields in the GUI.

These parameters are essential for the stereo vision computations and are stored in their respective variables to be used during the processing**.**

Arduino Connection with GUI:

- The initialize_arduino method is responsible for establishing the serial connection, ensuring that the Arduino is ready for communication.
- A retry mechanism is implemented via check_arduino_connection, which periodically checks if the Arduino is still connected.
- The application provides feedback on the connection status through the update_connection_status method, changing the UI elements to indicate whether the Arduino is connected (green) or disconnected (red).
- The handle_arduino_disconnection method manages scenarios in which the Arduino becomes disconnected. It ensures the worker thread is stopped, and the connection is properly closed, preventing resource leaks.
- An Arduino worker thread (ArduinoWorker) is utilized to handle incoming data from the Arduino without blocking the main GUI thread. This keeps the interface responsive and allows for continuous monitoring without freezing.
- The send_command method abstracts the process of sending commands to the Arduino. It includes error handling to manage situations where command transmission may fail due to connectivity issues.
- The GUI allows users to switch between keyboard and GUI controls using the control_by_GUI method for flexibility in controlling the connected hardware.
- The received data from the Arduino (e.g., voltage, current, distance) is processed and reflected in the GUI in real time. This ensures that users have immediate access to sensor readings.