**Firmware for esp32 Server using User Datagram Protocol:**

**Imports and Configurations:**

Import the necessary libraries for setting up the configurations/ip addresses, etc.

#include <WiFi.h>

#include <AsyncUDP.h>

Then the ip addresses of the esp32, camera, default gateway, etc are defined. Make sure to note the ip addresses and the UDP port as it is very important to establish further connections.

const char\* ssid = "aiLite\_0128";

const char\* password = "12345678";

const int udpPort = 8888; // UDP port

// Modified network configuration to work with camera

IPAddress local\_IP(192, 168, 82, 10);

IPAddress gateway(192, 168, 82, 1);

IPAddress subnet(255, 255, 255, 0);

The motors are controlled using defined pins and PWM frequencies. The bot starts with a default speed of 200 (out of 255). You can easily change the speed and direction by sending commands from a connected device. For instance: 'f' moves forward, 'b' moves backward, 'l' turns left, 's' stops, and you can specify speeds like 'f:150' (forward at 150), 'b:230' (backward at 230), or 'l:150' (left at 150).

// Motor pin definitions

#define motorpinLF1 14

#define motorpinLF2 16

#define motorpinLB1 32

#define motorpinLB2 33

#define motorpinRF1 19

#define motorpinRF2 17

#define motorpinRB1 25

#define motorpinRB2 26

// PWM properties

const int freq = 2400; // PWM frequency

const int resolution = 8; // 8-bit resolution (0-255)

// PWM channels for each motor pin

const int pwmChannelLF1 = 0;

const int pwmChannelLF2 = 1;

const int pwmChannelLB1 = 2;

const int pwmChannelLB2 = 3;

const int pwmChannelRF1 = 4;

const int pwmChannelRF2 = 5;

const int pwmChannelRB1 = 6;

const int pwmChannelRB2 = 7;

// Default motor speed (0-255) - Set to 200

int motorSpeed = 200;

Then in the setup function the pins, the access points and the other things are configured.

void setup() {

Serial.begin(115200);

// Configure AP with DHCP range that won't conflict with static IPs

WiFi.softAP(ssid, password);

WiFi.softAPConfig(local\_IP, gateway, subnet);

Serial.println("WiFi Access Point started.");

Serial.print("ESP32 IP: ");

Serial.println(WiFi.softAPIP());

// Configure motor enable pin

pinMode(motorEnablePin, OUTPUT);

digitalWrite(motorEnablePin, HIGH);

// Configure PWM for all motor pins

ledcSetup(pwmChannelLF1, freq, resolution);

ledcSetup(pwmChannelLF2, freq, resolution);

ledcSetup(pwmChannelLB1, freq, resolution);

ledcSetup(pwmChannelLB2, freq, resolution);

ledcSetup(pwmChannelRF1, freq, resolution);

ledcSetup(pwmChannelRF2, freq, resolution);

ledcSetup(pwmChannelRB1, freq, resolution);

ledcSetup(pwmChannelRB2, freq, resolution);

// Attach PWM channels to GPIO pins

ledcAttachPin(motorpinLF1, pwmChannelLF1);

ledcAttachPin(motorpinLF2, pwmChannelLF2);

ledcAttachPin(motorpinLB1, pwmChannelLB1);

ledcAttachPin(motorpinLB2, pwmChannelLB2);

ledcAttachPin(motorpinRF1, pwmChannelRF1);

ledcAttachPin(motorpinRF2, pwmChannelRF2);

ledcAttachPin(motorpinRB1, pwmChannelRB1);

ledcAttachPin(motorpinRB2, pwmChannelRB2);

The ESP32 listens for UDP commands sent to port 8888. The udp.onPacket() callback function processes received commands, also checks if speed is included along the command.

if (udp.listen(udpPort)) {

Serial.println("UDP server started.");

udp.onPacket([](AsyncUDPPacket packet) {

// Read the received data

String packetData = "";

for (int i = 0; i < packet.length(); i++) {

packetData += (char)packet.data()[i];

}

// Improved command parsing to better support speed control

// Format: <command>:<speed>

// Examples: "f:150" (forward at speed 150), "b" (backward at default speed)

int colonPos = packetData.indexOf(':');

char command = packetData[0];

// Check if speed setting is included

if (colonPos > 0 && colonPos < packetData.length() - 1) {

// Extract speed value after the colon

String speedStr = packetData.substring(colonPos + 1);

int newSpeed = speedStr.toInt();

// Validate and set the speed

if (newSpeed >= 0 && newSpeed <= 255) {

motorSpeed = newSpeed;

Serial.printf("Motor speed set to: %d\n", motorSpeed);

}

}

Then the appropriate command is executed and the response is sent to the UDP port. Then the motor functions are defined after that.

Serial.printf("Received command: %c, Speed: %d\n", command, motorSpeed);

switch (command) {

case 'f': Forward(); break;

case 'b': Backward(); break;

case 'l': Left(); break;

case 'r': Right(); break;

case 's': Stop(); break;

default: Serial.println("Invalid Command");

}

udp.writeTo((const uint8\_t\*)"Command executed", 16, packet.remoteIP(), packet.remotePort());

});

}

}

void loop() {

// Nothing needed in loop, everything runs in the UDP callback

}

void Forward() {

Serial.println("Moving Forward");

ledcWrite(pwmChannelLF1, motorSpeed);

ledcWrite(pwmChannelLF2, 0);

ledcWrite(pwmChannelLB1, motorSpeed);

ledcWrite(pwmChannelLB2, 0);

ledcWrite(pwmChannelRF1, motorSpeed);

ledcWrite(pwmChannelRF2, 0);

ledcWrite(pwmChannelRB1, motorSpeed);

ledcWrite(pwmChannelRB2, 0);

}

void Backward() {

Serial.println("Moving Backward");

ledcWrite(pwmChannelLF1, 0);

ledcWrite(pwmChannelLF2, motorSpeed);

ledcWrite(pwmChannelLB1, 0);

ledcWrite(pwmChannelLB2, motorSpeed);

ledcWrite(pwmChannelRF1, 0);

ledcWrite(pwmChannelRF2, motorSpeed);

ledcWrite(pwmChannelRB1, 0);

ledcWrite(pwmChannelRB2, motorSpeed);

}

void Left() {

Serial.println("Turning Left");

ledcWrite(pwmChannelLF1, 0);

ledcWrite(pwmChannelLF2, motorSpeed);

ledcWrite(pwmChannelLB1, 0);

ledcWrite(pwmChannelLB2, motorSpeed);

ledcWrite(pwmChannelRF1, motorSpeed);

ledcWrite(pwmChannelRF2, 0);

ledcWrite(pwmChannelRB1, motorSpeed);

ledcWrite(pwmChannelRB2, 0);

}

void Right() {

Serial.println("Turning Right");

ledcWrite(pwmChannelLF1, motorSpeed);

ledcWrite(pwmChannelLF2, 0);

ledcWrite(pwmChannelLB1, motorSpeed);

ledcWrite(pwmChannelLB2, 0);

ledcWrite(pwmChannelRF1, 0);

ledcWrite(pwmChannelRF2, motorSpeed);

ledcWrite(pwmChannelRB1, 0);

ledcWrite(pwmChannelRB2, motorSpeed);

}

void Stop() {

Serial.println("Stopping");

ledcWrite(pwmChannelLF1, 0);

ledcWrite(pwmChannelLF2, 0);

ledcWrite(pwmChannelLB1, 0);

ledcWrite(pwmChannelLB2, 0);

ledcWrite(pwmChannelRF1, 0);

ledcWrite(pwmChannelRF2, 0);

ledcWrite(pwmChannelRB1, 0);

ledcWrite(pwmChannelRB2, 0);

}

**Firmware for esp32 Camera using HTTP Protocol:**

As mentioned before, the required libraries are included at the beginning to handle WiFi connectivity, camera interfacing, image processing, and HTTP server functionalities:

#include "esp\_camera.h"

#include <WiFi.h>

#include "esp\_timer.h"

#include "img\_converters.h"

#include "Arduino.h"

#include "fb\_gfx.h"

#include "soc/soc.h"

#include "soc/rtc\_cntl\_reg.h"

#include "esp\_http\_server.h"

The WiFi credentials are set to match the robot’s network. The ESP32-CAM is assigned a static IP to ensure consistent network communication

// WiFi credentials - same as robot

const char\* ssid = "aiLite\_0128";

const char\* password = "12345678";

// Static IP for camera

IPAddress local\_IP(192, 168, 82, 14);

IPAddress gateway(192, 168, 82, 1);

IPAddress subnet(255, 255, 255, 0);

The camera’s hardware pins are defined based on the AI Thinker ESP32-CAM module

// Camera pins for AI Thinker ESP32-CAM

#define PWDN\_GPIO\_NUM 32

#define RESET\_GPIO\_NUM -1

#define XCLK\_GPIO\_NUM 0

#define SIOD\_GPIO\_NUM 26

#define SIOC\_GPIO\_NUM 27

#define Y9\_GPIO\_NUM 35

#define Y8\_GPIO\_NUM 34

#define Y7\_GPIO\_NUM 39

#define Y6\_GPIO\_NUM 36

#define Y5\_GPIO\_NUM 21

#define Y4\_GPIO\_NUM 19

#define Y3\_GPIO\_NUM 18

#define Y2\_GPIO\_NUM 5

#define VSYNC\_GPIO\_NUM 25

#define HREF\_GPIO\_NUM 23

#define PCLK\_GPIO\_NUM 22

The HTTP streaming server is implemented to serve the video stream. The response type is set to multipart/x-mixed-replace to allow continuous frame updates. The stream\_handler function captures frames, compresses them if necessary, and sends them over HTTP

// HTTP streaming implementation

#define PART\_BOUNDARY "123456789000000000000987654321"

static const char\* \_STREAM\_CONTENT\_TYPE = "multipart/x-mixed-replace;boundary=" PART\_BOUNDARY;

static const char\* \_STREAM\_BOUNDARY = "\r\n--" PART\_BOUNDARY "\r\n";

static const char\* \_STREAM\_PART = "Content-Type: image/jpeg\r\nContent-Length: %u\r\n\r\n";

httpd\_handle\_t stream\_httpd = NULL;

static esp\_err\_t stream\_handler(httpd\_req\_t \*req) {

camera\_fb\_t \* fb = NULL;

esp\_err\_t res = ESP\_OK;

size\_t \_jpg\_buf\_len = 0;

uint8\_t \* \_jpg\_buf = NULL;

char \* part\_buf[64];

res = httpd\_resp\_set\_type(req, \_STREAM\_CONTENT\_TYPE);

if(res != ESP\_OK) {

return res;

}

while(true) {

fb = esp\_camera\_fb\_get();

if (!fb) {

Serial.println("Camera capture failed");

res = ESP\_FAIL;

} else {

if(fb->format != PIXFORMAT\_JPEG) {

bool jpeg\_converted = frame2jpg(fb, 80, &\_jpg\_buf, &\_jpg\_buf\_len);

esp\_camera\_fb\_return(fb);

fb = NULL;

if(!jpeg\_converted) {

Serial.println("JPEG compression failed");

res = ESP\_FAIL;

}

} else {

\_jpg\_buf\_len = fb->len;

\_jpg\_buf = fb->buf;

}

}

if(res == ESP\_OK) {

size\_t hlen = snprintf((char \*)part\_buf, 64, \_STREAM\_PART, \_jpg\_buf\_len);

res = httpd\_resp\_send\_chunk(req, (const char \*)part\_buf, hlen);

}

if(res == ESP\_OK) {

res = httpd\_resp\_send\_chunk(req, (const char \*)\_jpg\_buf, \_jpg\_buf\_len);

}

if(res == ESP\_OK) {

res = httpd\_resp\_send\_chunk(req, \_STREAM\_BOUNDARY, strlen(\_STREAM\_BOUNDARY));

}

if(fb) {

esp\_camera\_fb\_return(fb);

fb = NULL;

\_jpg\_buf = NULL;

} else if(\_jpg\_buf) {

free(\_jpg\_buf);

\_jpg\_buf = NULL;

}

if(res != ESP\_OK) {

break;

}

}

return res;

}

The function startCameraServer sets up the HTTP server and registers the stream handler

void startCameraServer() {

httpd\_config\_t config = HTTPD\_DEFAULT\_CONFIG();

config.server\_port = 81;

httpd\_uri\_t stream\_uri = {

.uri = "/stream",

.method = HTTP\_GET,

.handler = stream\_handler,

.user\_ctx = NULL

};

if (httpd\_start(&stream\_httpd, &config) == ESP\_OK) {

httpd\_register\_uri\_handler(stream\_httpd, &stream\_uri);

}

}

The setup function initializes the camera with specific parameters, disables the brownout detector, and configures the WiFi connection. The static IP is set before connecting to avoid conflicts.

void setup() {

WRITE\_PERI\_REG(RTC\_CNTL\_BROWN\_OUT\_REG, 0); // Disable brownout detector

Serial.begin(115200);

Serial.setDebugOutput(true);

Serial.println();

camera\_config\_t config;

config.ledc\_channel = LEDC\_CHANNEL\_0;

config.ledc\_timer = LEDC\_TIMER\_0;

config.pin\_d0 = Y2\_GPIO\_NUM;

config.pin\_d1 = Y3\_GPIO\_NUM;

config.pin\_d2 = Y4\_GPIO\_NUM;

config.pin\_d3 = Y5\_GPIO\_NUM;

config.pin\_d4 = Y6\_GPIO\_NUM;

config.pin\_d5 = Y7\_GPIO\_NUM;

config.pin\_d6 = Y8\_GPIO\_NUM;

config.pin\_d7 = Y9\_GPIO\_NUM;

config.pin\_xclk = XCLK\_GPIO\_NUM;

config.pin\_pclk = PCLK\_GPIO\_NUM;

config.pin\_vsync = VSYNC\_GPIO\_NUM;

config.pin\_href = HREF\_GPIO\_NUM;

config.pin\_sscb\_sda = SIOD\_GPIO\_NUM;

config.pin\_sscb\_scl = SIOC\_GPIO\_NUM;

config.pin\_pwdn = PWDN\_GPIO\_NUM;

config.pin\_reset = RESET\_GPIO\_NUM;

config.xclk\_freq\_hz = 20000000;

config.pixel\_format = PIXFORMAT\_JPEG;

// Initialize with high specs to pre-allocate larger buffers

if(psramFound()) {

config.frame\_size = FRAMESIZE\_SVGA;

config.jpeg\_quality = 10; // 0-63, lower means higher quality

config.fb\_count = 2;

} else {

config.frame\_size = FRAMESIZE\_CIF;

config.jpeg\_quality = 12;

config.fb\_count = 1;

}

// Camera init

esp\_err\_t err = esp\_camera\_init(&config);

if (err != ESP\_OK) {

Serial.printf("Camera init failed with error 0x%x", err);

delay(1000);

ESP.restart();

}

// Set static IP before connecting

if (!WiFi.config(local\_IP, gateway, subnet)) {

Serial.println("Static IP Configuration Failed");

}

// Connect to WiFi network

WiFi.begin(ssid, password);

Serial.println("Connecting to WiFi");

// Wait for connection

while (WiFi.status() != WL\_CONNECTED) {

delay(500);

Serial.print(".");

}

Serial.println("");

Serial.println("WiFi connected");

Serial.print("Camera Stream Ready! Go to: http://");

Serial.print(WiFi.localIP());

Serial.println(":81/stream");

// Start streaming web server

startCameraServer();

}

void loop() {

delay(1);

}

**Lane/Line Detection with Binary Thresholding:**

**Imports and Configurations:**

It is very necessary to import the required libraries for the function of the code for image handling, sending commands, calculating time, etc.

import cv2

import numpy as np

import socket

import time

import requests

For the configuration values you will need to change it according to your components. You must specify the same ip address of your bot. Also be sure to use the same UDP port that you have used in your firmware code.

camera\_ip = "192.168.82.14" # ESP32-CAM IP

robot\_ip = "192.168.82.10" # ESP32 Robot IP

udp\_port = 8888 # Same port as used in the Firmware Code

stream\_url = f"[http://{camera\_ip}:81/stream](about:blank)" # Stream link of the esp32 Camera

speed = 150 # Speed to control the bot

These are the initial parameters for the functions below.

# Initialize adaptive contour area tracking

contour\_areas\_history = []

MAX\_HISTORY\_SIZE = 10

MIN\_CONTOUR\_AREA\_FACTOR = 0.5 # Start with 50% of the average

# Initialize FPS variables

prev\_frame\_time = 0

new\_frame\_time = 0

# Time control for commands

last\_command\_time = 0

COMMAND\_DELAY = 0

**Functions and Parameters:**

1. **stream\_video:**

def stream\_video(stream\_url):

session = requests.Session()

bytes\_data = bytes()

try:

response = session.get(stream\_url, stream=True)

if response.status\_code == 200:

for chunk in response.iter\_content(chunk\_size=1024):

bytes\_data += chunk

a = bytes\_data.find(b'\xff\xd8')

b = bytes\_data.find(b'\xff\xd9')

if a != -1 and b != -1:

jpg = bytes\_data[a:b+2]

bytes\_data = bytes\_data[b+2:]

frame = cv2.imdecode(np.frombuffer(jpg, dtype=np.uint8), cv2.IMREAD\_COLOR)

if frame is not None:

# Rotate the frame 180 degrees to fix upside down image

frame = cv2.rotate(frame, cv2.ROTATE\_180)

yield frame

except Exception as e:

print(f"Stream error: {e}")

return None

This function is used to capture the live video feed from the esp32 Ip camera by using a HTTP connection. It iteratively receives data chunks, accumulating them until complete JPEG image markers are detected. Upon finding these markers, it extracts the JPEG data, decodes it using OpenCV, and corrects any upside-down orientation by rotating the frame 180 degrees.

1. **send\_udp\_command:**

def send\_udp\_command(command):

global last\_command\_time

current\_time = time.time()

if current\_time - last\_command\_time < COMMAND\_DELAY:

return "Command skipped - too soon"

with socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM) as sock:

sock.sendto(command.encode(), (robot\_ip, udp\_port)) # Send command

response, \_ = sock.recvfrom(1024) # Receive response

return response.decode()

This function is used to send the commands to the bot. There is a limit to how many commands we can send to the bot, to avoid flooding commands to the bot. If you experience any type of lag while sending, executing the commands and the video feed it is recommended to increase the COMMAND\_DELAY.

1. **calculate\_steering\_angle:**

def calculate\_steering\_angle(frame, largest\_contour):

height, width = frame.shape[:2]

M = cv2.moments(largest\_contour)

cx = int(M["m10"] / M["m00"]) if M["m00"] != 0 else width // 2

center\_offset = cx - (width // 2)

steering\_angle = -float(center\_offset) \* 0.1

return steering\_angle, cx

This function determines the “cx ”centroid(centre point) from the current frame and calculates the offset or the deviation and also returns that for appropriate countermeasures.

1. **apply\_steering\_control:**

def apply\_steering\_control(steering\_angle):

if abs(steering\_angle) < 14:

send\_udp\_command(f'f:{speed}')

return "Go Straight"

elif steering\_angle > 0:

send\_udp\_command(f'l:{speed}')

return "Turn Left"

else:

send\_udp\_command(f'r:{speed}')

return "Turn Right"

As said before the offset values are used in this function to align the robot in the Lane/Line accordingly by sending the appropriate commands.

1. **get\_adaptive\_min\_contour\_area:**

def get\_adaptive\_min\_contour\_area():

if not contour\_areas\_history:

return 10000

avg\_area = sum(contour\_areas\_history) / len(contour\_areas\_history)

return avg\_area \* MIN\_CONTOUR\_AREA\_FACTOR

This function is used to calculate the area and set a dynamic threshold for the minimum contour to detect the Lane/Line accordingly.

1. **main():**

The program starts by capturing a live video feed from the ESP32-CAM using an HTTP stream. Each frame is then converted to grayscale and thresholded to create a binary image. This helps in distinguishing the lane markings from the background. A region of interest (ROI) is defined to focus processing on the lower half of the frame, where the lanes are most likely to be present.

def main():

prev\_frame\_time = time.time()

for frame in stream\_video(stream\_url):

height, width = frame.shape[:2]

gray\_frame = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)

\_, binary\_frame = cv2.threshold(gray\_frame, 100, 255, cv2.THRESH\_BINARY\_INV)

roi\_vertices = np.array([[(0, height // 2), (width, height // 2), (width, height), (0, height)]], dtype=np.int32)

mask = np.zeros\_like(binary\_frame)

cv2.fillPoly(mask, roi\_vertices, 255)

roi\_result = cv2.bitwise\_and(binary\_frame, mask)

It is very important to adjust the lower limit of the threshold inside of the cv2.threshold function according to your lighting conditions to avoid detection of duplicate Lane/Line Contours.

These are a few examples of different threshold values.

* cv2.threshold(gray\_frame, 100, 255, cv2.THRESH\_BINARY\_INV)
* cv2.threshold(gray\_frame, 50, 255, cv2.THRESH\_BINARY\_INV)
* cv2.threshold(gray\_frame, 120, 255, cv2.THRESH\_BINARY\_INV)
* cv2.threshold(gray\_frame, 80, 255, cv2.THRESH\_BINARY\_INV)

Contours (shapes) are extracted from the processed binary image. The program maintains a history of detected contour areas and dynamically adjusts the minimum contour size required to detect lanes. This adaptive approach improves robustness in different road conditions.

# Find contours in the region of interest

contours, \_ = cv2.findContours(roi\_result, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

# Update contour area history

if contours:

largest\_area = max([cv2.contourArea(cnt) for cnt in contours])

contour\_areas\_history.append(largest\_area)

# Keep history size within limits

if len(contour\_areas\_history) > MAX\_HISTORY\_SIZE:

contour\_areas\_history.pop(0)

# Determine the minimum valid contour area dynamically

MIN\_CONTOUR\_AREA = get\_adaptive\_min\_contour\_area()

# Filter out small contours that are unlikely to be lanes

valid\_contours = [cnt for cnt in contours if cv2.contourArea(cnt) > MIN\_CONTOUR\_AREA]

By maintaining a history of contour sizes, the program can adapt to changing road conditions and avoid false detections caused by noise.

Once a valid lane is detected, the system calculates its center and determines how much it deviates from the frame's center. This deviation is used to compute the steering angle. The appropriate movement command is then sent to the ESP32 robot over a UDP connection.

if valid\_contours:

lanes\_detected = True

largest\_contour = max(valid\_contours, key=cv2.contourArea)

cv2.fillPoly(overlay, [largest\_contour], (0, 255, 0))

highlighted\_frame = cv2.addWeighted(frame, 1, overlay, 0.3, 0)

cv2.drawContours(highlighted\_frame, [largest\_contour], -1, (0, 255, 255), 2)

steering\_angle, center\_x = calculate\_steering\_angle(frame, largest\_contour)

steering\_direction = apply\_steering\_control(steering\_angle)

# Draw guiding bullet point

cv2.circle(highlighted\_frame, (center\_x, height - 50), 5, (0, 0, 255), -1)

cv2.line(highlighted\_frame, (width // 2, height - 50), (center\_x, height - 50), (0, 0, 255), 2)

# Draw steering direction above the bullet point

text\_size = cv2.getTextSize(steering\_direction, cv2.FONT\_HERSHEY\_SIMPLEX, 1, 2)[0]

text\_x = center\_x - text\_size[0] // 2

text\_y = height - 80

cv2.putText(highlighted\_frame, steering\_direction, (text\_x, text\_y), cv2.FONT\_HERSHEY\_SIMPLEX, 1, (0, 0, 255), 2)

else:

# No lanes detected or all contours are below threshold

send\_udp\_command('s')

message = "No Lanes Detected"

text\_size = cv2.getTextSize(message, cv2.FONT\_HERSHEY\_SIMPLEX, 1, 2)[0]

text\_x = width // 2 - text\_size[0] // 2

text\_y = 2 \* (height // 3)

cv2.putText(highlighted\_frame, message, (text\_x, text\_y), cv2.FONT\_HERSHEY\_SIMPLEX, 1, (0, 0, 255), 2)

If no valid lanes are detected, the robot stops to avoid moving in the wrong direction.

To help with debugging and performance monitoring, the program overlays detected lanes, draws the region of interest, and displays the FPS (frames per second). This ensures that the lane detection process is smooth and real-time.

# Draw ROI outline

cv2.polylines(highlighted\_frame, [roi\_vertices], True, (255, 0, 0), 2)

# Display the current adaptive threshold

cv2.putText(highlighted\_frame, f'Min Area: {int(MIN\_CONTOUR\_AREA)}', (10, 60),

cv2.FONT\_HERSHEY\_SIMPLEX, 0.7, (0, 255, 0), 2)

# Calculate and display Frame Rates

new\_frame\_time = time.time()

fps = 0 if (new\_frame\_time-prev\_frame\_time == 0) else (1/(new\_frame\_time-prev\_frame\_time))

prev\_frame\_time = new\_frame\_time

fps = int(fps) if fps > 0 else 0

cv2.putText(highlighted\_frame, f'FPS: {fps}', (10,30), cv2.FONT\_HERSHEY\_SIMPLEX, 1, (0,255,0), 2)

cv2.putText(binary\_frame, f'FPS: {fps}', (10,30), cv2.FONT\_HERSHEY\_SIMPLEX, 1, (0,255,0), 2)

# Display the frames

cv2.imshow('Binary Frame', cv2.resize(binary\_frame, (720, 540)))

cv2.imshow('Lane Detection with Steering', cv2.resize(highlighted\_frame, (720, 540)))

if cv2.waitKey(1) & 0xFF == ord('q'):

break

send\_udp\_command('s')

cv2.destroyAllWindows()

if \_\_name\_\_ == '\_\_main\_\_':

main()

Then finally it sends a stop command to ensure that the bot stops and completes the code execution.