DEPARTMENT OF COMPUTER & SOFTWARE ENGINEERING

COLLEGE OF E&ME, NUST, RAWALPINDI

Subject Name
Digital Image Processing

PI Navigation System

SUBMITTED TO:
Dr. Usman Akram
Dr. Asad Mansoor

SUBMITTED BY:
Student Name

1. Wahaaj Nasir

Reg#413238

2. Syed Adnan Aijaz

Reg#432028

3. Muhammad Ali Riaz

Reg#432103

4. Ramsha Fatima

Reg#417858
DE- 44 Dept C&SE

Objectives:

Advancements in artificial intelligence have significantly enhanced self-driving technology, largely due to the superior performance of deep learning models that have made fully autonomous vehicles a reality. Nonetheless, classical image processing techniques can also deliver robust performance and operate in pseudo real-time, offering a viable alternative to deep learning approaches. In this term project, you will develop the visual component of a self-driving car optimized to run on a Raspberry Pi 5, demonstrating that high-performance, cost-effective autonomous navigation is achievable using traditional methods.

Dataset:
The dataset that we will be using has been collected from the College. You are free to capture more videos/images if you see fit to test and refine your algorithm. The videos can be downloaded from the following link:
https://drive.google.com/file/d/1dJYyjc6u08ob8WTFlGNBBppujia_SzGz/view?usp=sharing

In this applied project, you are tasked with designing and developing a robust self-driving solution using classical image processing techniques that runs entirely on a Raspberry Pi equipped with a Pi Camera.
The system should address the following core functionalities:

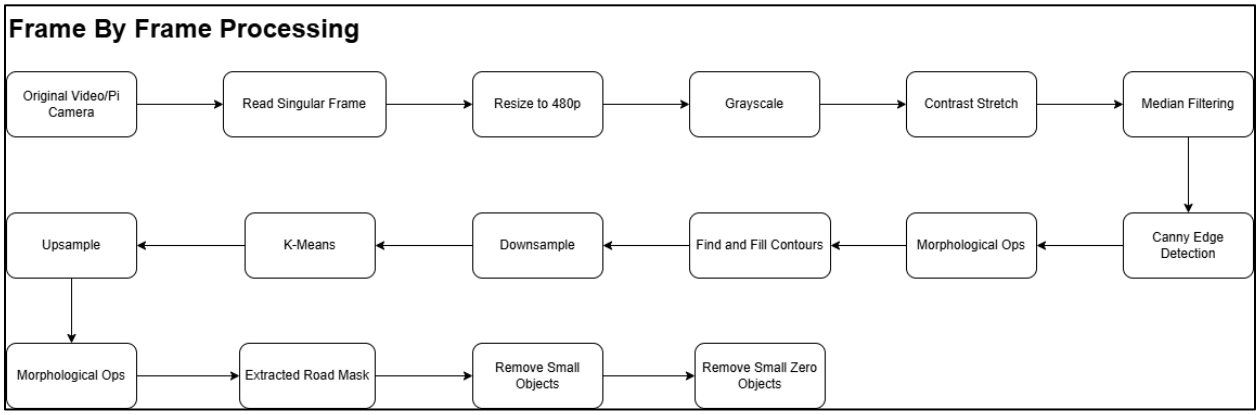
- **Stop/Move Decision**: Analyze the environment to determine whether the car should stop or continue moving, integrating obstacle detection as well.
- **Directional Control**: Establish the appropriate direction of movement—left, right, forward, or backward—by interpreting lane geometry and any dynamic obstacles in the car's path.
- **Lane Detection**: Reliably detect road lanes using classical techniques such as edge detection, adaptive thresholding, and Hough transforms, with further refinements like polynomial fitting for curved roads.
- **On-Device Processing**: Ensure that all image processing and decision-making steps are executed in real time on a Raspberry Pi, optimizing computational efficiency through region of-interest (ROI) restriction, down sampling, and parallel processing where applicable.
- **Output Display and Debugging**: Provide real-time output on a laptop via terminal, including not only the final driving decisions but also intermediate outputs (e.g., processed frames, detected edges, and lane markings) for debugging and system tuning.

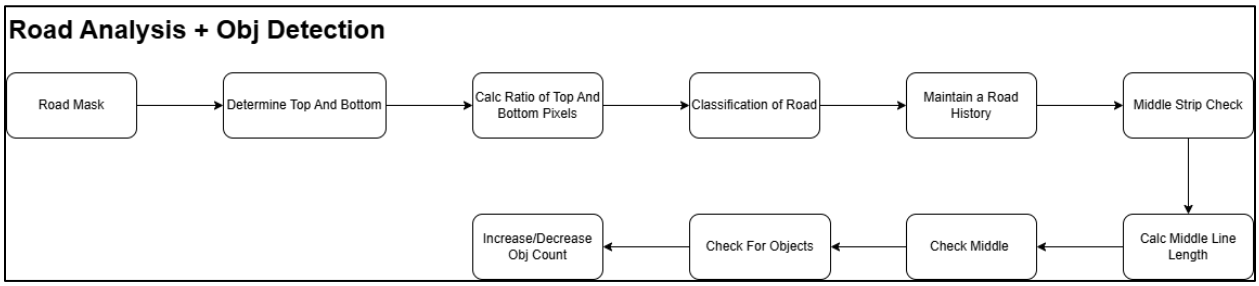
Flow Diagrams:

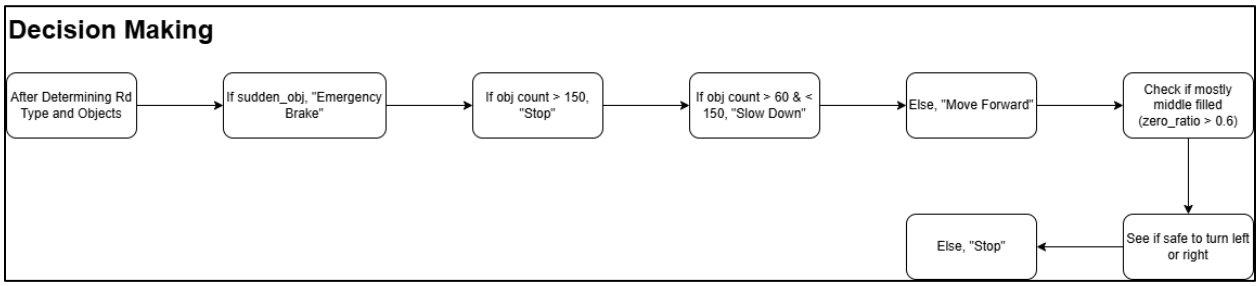
For ease of understanding of both us and the instructors, the flow of the project has been divided into the following 3 sections:

- Frame-By-Frame Processing
- Road Analysis + Object Detection
- Decision Making

The flow diagrams of the following sections are as below:







Part 1 – Frame By Frame Processing:

In this section, we will discuss how each and every frame of the database is captured and processed.

Setting up the PI Camera:

```
picam2 = Picamera2()
picam2.configure(
    picam2.create_preview_configuration(
        main={"format": 'RGB888', "size": (640, 480)}
    )
)
```

This simple block of code sets up our PI Camera to capture frames in 480p and RGB Format. This RGB is further converted into grayscale for our processing. The original idea was to develop the algorithm for 720p, but that provided us with only 5-10 FPS. Using 480p and scaling the values from the original algorithm, we are able to obtain around 13-15 FPS consistently

Preprocessing:

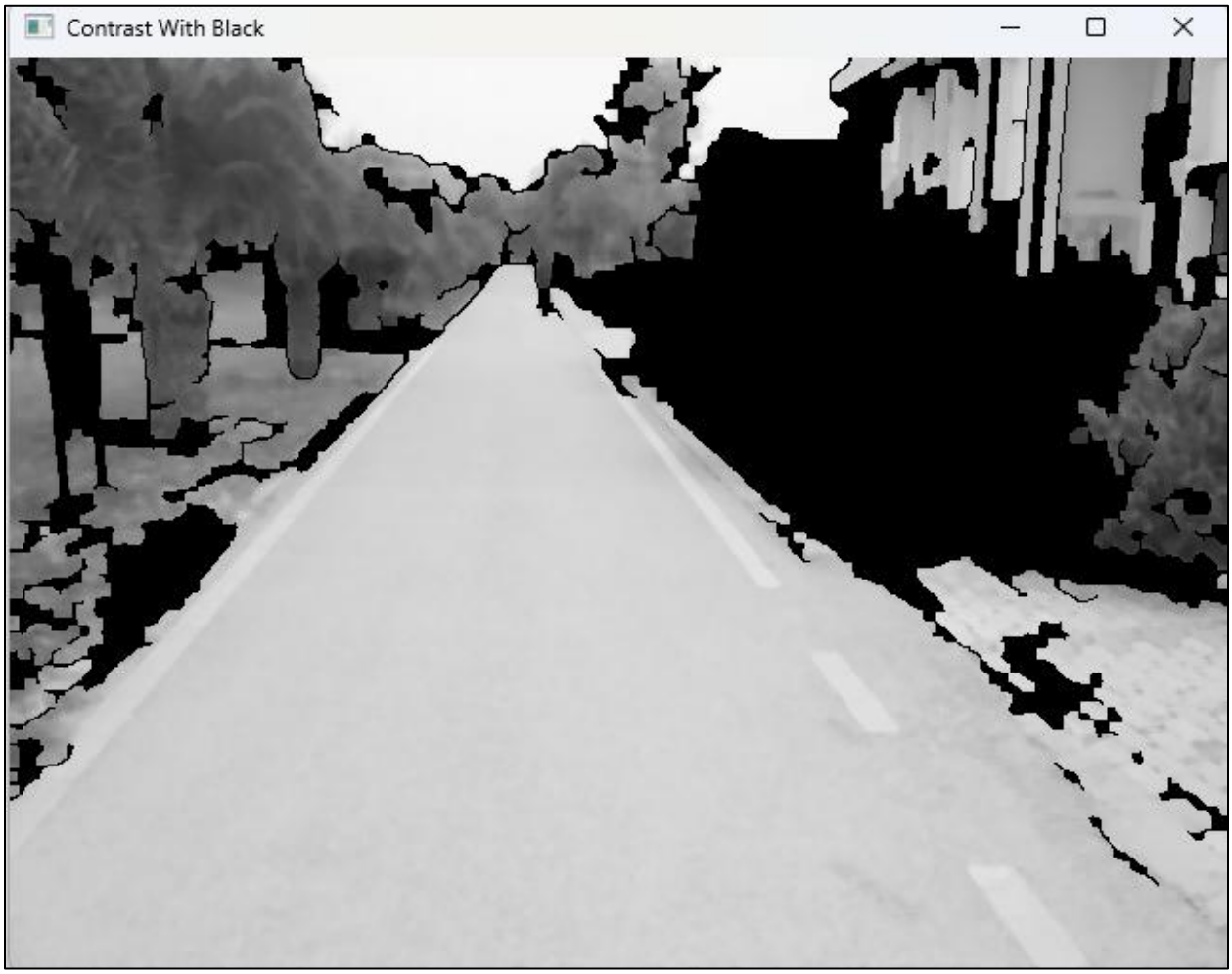
```
detect_obj_count = 0
road_classification_history = []
prev_zero_ratio = 0
prev_frame_time = 0
new_frame_time = 0
kernel_close = cv.getStructuringElement(cv.MORPH_RECT, (5, 5))
```

The above lines of code are global variables we need to use later. These include **`detect_obj_count`**, **`road_classification_history`**, **`prev_zero_ratio`**, **`prev_frame_time`**, **`new_frame_time`** and **`kernel_close`**. The use of these variables will be discussed later at the appropriate time.

```
new_frame_time = time.time()
gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)

contrast = contrast_stretching(gray)
median_filtered = cv.medianBlur(contrast, 5)
edges = cv.Canny(median_filtered, 50, 150)

closed_edges = cv.morphologyEx(edges, cv.MORPH_CLOSE, kernel_close)
contours, _ = cv.findContours(closed_edges, cv.RETR_EXTERNAL,
cv.CHAIN_APPROX_SIMPLE)
contrast_with_black = median_filtered.copy()
cv.drawContours(contrast_with_black, contours, -1, color=0,
thickness=cv.FILLED)
```

The above code is giving a value to **`new_frame_time`**. This is done so that we can determine the FPS of the algo. Then we move to **`contrast_stretching`**, which is done so that we can effectively stretch the grayscale values of the image over the range of 0 to 255. This allows us

to correctively segment out the road from surrounding objects ahead. We utilize Median Blurring to reduce any salt-and-pepper noise, and Canny Edge to find sudden spikes i.e objects. Up next are our ***morphological operations*** . These morph ops are utilizing the output from the Canny Edge detection, to close the boundaries of objects. Up next we use the `cv.findContours` built in function to effectively find the contours around our detected objects. Then using the `cv.drawContours` function, we fill those outlines, classifying them as "objects"



K-Means:

For the purpose of our project, we utilized K-means clustering. This allowed us to effectively segment out our road and other objects. In the main code it is called as:

```
down = cv.resize(contrast_with_black, (0, 0), fx=0.25, fy=0.25,
interpolation=cv.INTER_AREA)
seg_down = kmeans_segmentation_fast(down)
segmented = cv.resize(seg_down, gray.shape[::-1],
interpolation=cv.INTER_NEAREST)
```

As you can see, we first downscale the image by 0.25 both horizontally and vertically by using the `cv.INTER_AREA` interpolation. This interpolation downscales using weighted averages, resulting in a downsampled image but without any aliasing. Then we run it through our

kmeans_segmentation_fast function, which applies k-means on the blurry image. The code for k-means is:

```
def kmeans_segmentation_fast(image, k=3):
    Z = image.reshape((-1, 1)).astype(np.float32)
    criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 10, 1.0)
    _, labels, _ = cv.kmeans(Z, k, None, criteria, 5,
cv.KMEANS_RANDOM_CENTERS)
    label_img = labels.reshape(image.shape)
    h, w = label_img.shape
    guide_band = label_img[int(h * 0.85):, int(w * 0.4):int(w * 0.6)]
    counts = [np.sum(guide_band == i) for i in range(k)]
    road_label = np.argmax(counts)
    mapped = np.zeros_like(label_img, dtype=np.uint8)
    for i in range(k):
        mapped[label_img == i] = 255 if i == road_label else 127
    return mapped
```

Our code first flattens the image into a single 1D array **z**. This step is crucial as the k-means built-in algorithm requires a flattened 1D array. Next we define a **criteria**. This criteria tells the k-means how long its supposed to run. For this the **cv.TERM_CRITERIA_EPS** and **cv.TERM_CRITERIA_MAX_ITER** are set to **1.0** and **10** respectively. This means that the k-means stops when either the distances of the centeroids is 1.0 or we have reached 10 iterations.

After that, we reshape the image back to the original shape. Then we define a **guide band**, which covers the bottom 15% of the image and 20% of the middle of that. We assume that our road is located in this area, so we get the label from this area with the maximum count. The **mapped** variable then takes that label and labels it **255** (white), while everything else is labeled as **127** (gray).

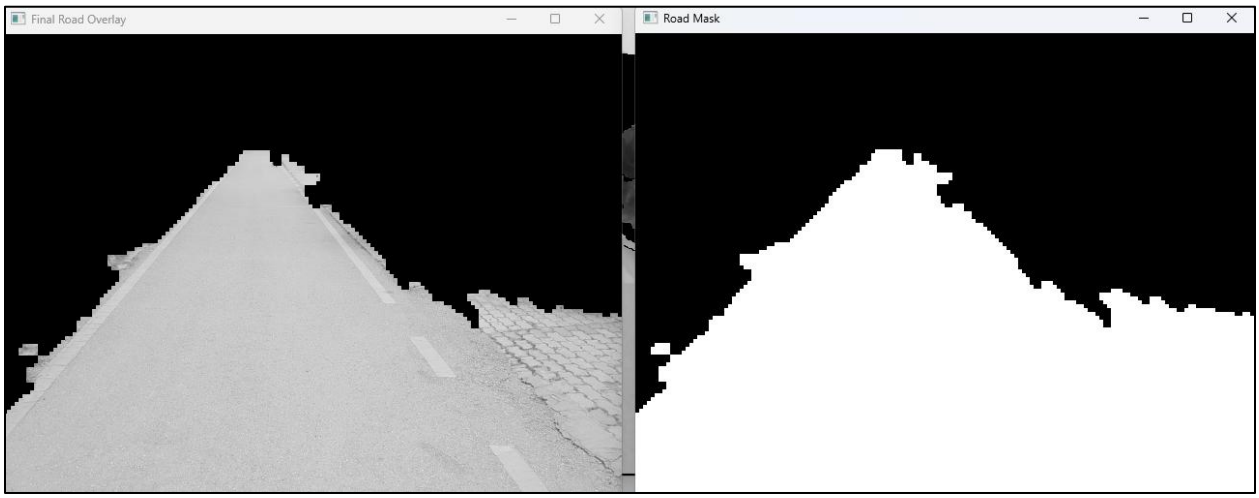
After doing all of this, we upsample the returning image using **cv.INTER_NEAREST.** This preserves the labels we get from the image.

Road Mask:

```
opened = cv.morphologyEx(segmented, cv.MORPH_OPEN, kernel_close)
closed = cv.morphologyEx(opened, cv.MORPH_CLOSE, kernel_close)
road_mask = np.zeros_like(closed)
road_mask[closed == 255] = 255

h, w = road_mask.shape
lower_75_mask = np.zeros_like(road_mask)
lower_75_mask[int(h * 0.25):, :] = 255
road_mask = cv.bitwise_and(road_mask, lower_75_mask)

road_mask = remove_small_objects(road_mask, 11000)
road_mask = remove_small_zero_objects(road_mask, 2000)
```

Again, here we utilize the morpho ops to fill holes and remove more noise that could come. Next, we create a road mask which converts our `label_img` from K-means into a mask that we utilize for recognizing the road. We then reduce it to only 75% of the screen. This is done because we don't want the sky being classified as road, as we can see that k-means is

classifying the sky as the road. Next, we remove any objects in the road that k-means considered as road, but are actually much smaller than the area of the road. This reduces things like people in white clothing and white cars being classified as road. Next we remove the smaller objects that are tiny on the screen i.e leaves, grass on road etc. These objects aren't a problem for our car and thus should not be included in its decision to stop or move.



Failure

As you can see, some part of the footpath is identified as road. I was unable to cater my algorithm to segment out those regions, as that requires texture analysis and that would be too computationally expensive.

Part 2 – Road Analysis + Object Detection:

Now we move on to Part 2, which involves our road classification and object detection.

Overlays, Center, Top and Bottom:

```
overlay = cv.bitwise_and(gray, gray, mask=road_mask)
vis_overlay = cv.cvtColor(overlay.copy(), cv.COLOR_GRAY2BGR)

center_x = w // 2
center_y = int((h // 2 + (50 * 0.66)))
detection_y = int((center_y - (50 * 0.66)))

top_y = np.argmax(np.any(road_mask == 255, axis=1)) + 33
bottom_y = h - np.argmax(np.any(road_mask[::-1] == 255, axis=1)) - 1

# Magenta and Yellow lines for top and bottom of the screen respectively
cv.line(vis_overlay, (0, top_y), (w, top_y), (255, 0, 255), 2)
cv.line(vis_overlay, (0, bottom_y), (w, bottom_y), (0, 255, 255), 2)
```

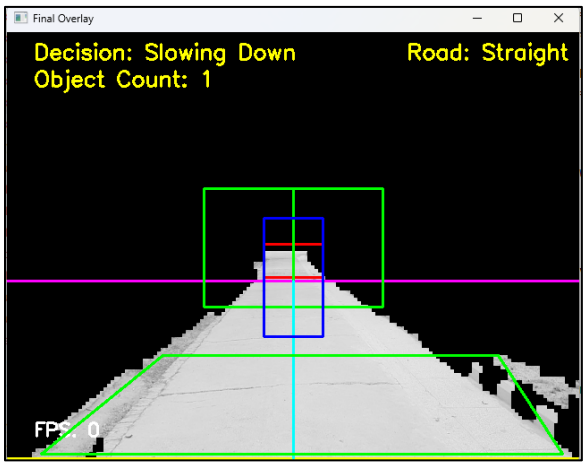
This code allows us to determine the middle, top and bottom of the road. These values are essential for our detection rectangles later on. As you can see, we have offset our values a bit by about 33 pixels. This is so that it does not classify a singular pixel on the top somewhere as the top, and instead comes down a bit to what we assume to be a complete line. The same goes for the bottom of the screen, as we move it up by 1 pixel (since mostly our road will be on the

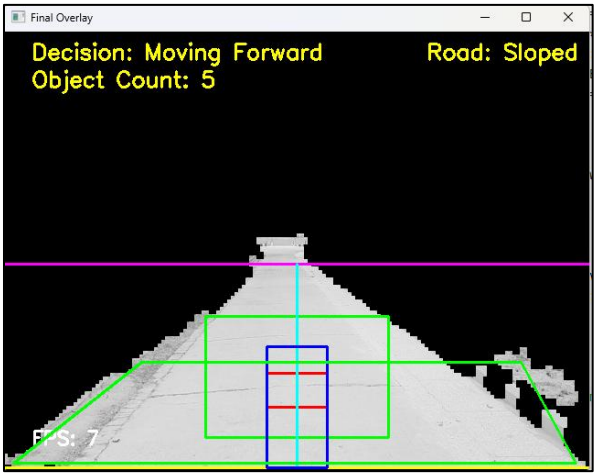
bottom anyways).

Sloped or Straight:

```
top_px = np.sum(road_mask[top_y, :] == 255)
bottom_px = np.sum(road_mask[bottom_y, :] == 255)

# Added a road history because algo has tendency to go from straight to
sloped to straight
road = "Straight"

if bottom_px != 0 and top_px / bottom_px < 0.15:
    road_classification_history.append("Sloped")
else:
    road_classification_history.append("Straight")
    center_y = h // 2 + 33

if len(road_classification_history) > 10:
    road_classification_history.pop(0)

if road_classification_history.count("Sloped") >= 7:
    road = "Sloped"
    center_y = bottom_y - 66
    detection_y = center_y - 33
```

This is a detection algorithm for a sloped or straight road. We get the total number of pixels from the top and bottom rows and find their ratio. If the ratio is greater than `0.15`, we know that this is a straight road. Thus we put our `center_y` at `h // 2 + 33`. We maintain a history for this because as a car is constantly moving, the ratio will change. If it determines a slope for 7 iterations, it changes our values for center and detection downwards, to suit the slope. In practical output, this can be shown as follows:





Detection Rectangles and Lines:

```
check_height = int(75 * 0.5)
check_width = int(50 * 0.66)
left_x, right_x = int(center_x - check_width),int(center_x + check_width)
vertical_strip = road_mask[center_y - check_height:center_y, left_x:right_x]
vertical_clear = np.all(vertical_strip == 255)

# Red Rectangle
cv.rectangle(vis_overlay, (left_x, center_y - check_height), (right_x,
center_y), (0, 0, 255), 2)
#Green Rectangles
cv.rectangle(vis_overlay, (center_x - 100, detection_y - 66), (center_x,
detection_y + 66), (0, 255, 0), 2)
cv.rectangle(vis_overlay, (center_x, detection_y - 66), (center_x + 100,
detection_y + 66), (0, 255, 0), 2)

# Middle Line for visualization
middle_line = road_mask[top_y:bottom_y, center_x]
middle_line_length = np.sum(middle_line == 255)
cv.line(vis_overlay, (center_x, top_y), (center_x, bottom_y), (255, 255, 0),
2)

middle_roi_w = 66 + 66 if detect_obj_count > 50 else 66
middle_roi_h = 66
middle_roi = road_mask[center_y - middle_roi_h:center_y + middle_roi_h,
                       center_x - middle_roi_w // 2:center_x + middle_roi_w
// 2]
object_detected = np.any(middle_roi == 0)

if object_detected:
    detect_obj_count += 1
else:
    # Reduce obj count, so that it realistically slows down and doesn't just
immediately go to breaking to account
    # for sway in camera
    detect_obj_count = max(detect_obj_count - 1, 0)

# Middle Blue Rectangle
cv.rectangle(vis_overlay,
             (center_x - middle_roi_w // 2, center_y - middle_roi_h),
             (center_x + middle_roi_w // 2, center_y + middle_roi_h),
             (255, 0, 0), 2)
```

The code at the top defines the centers and strips that we utilize for object detection as well as the decision for turning left and right.

The above code utilizes 3 main rectangles:

- **Red:** This rectangle has the purpose of defining whether our center is clear or not. If this is not clear, we know that the car should stop
- **Blue:** This rectangle is for the object detection on long straight roads, as well as for slowing the car down. You can see that if we detect any objects in this area, we slowly

increase the object count. The object count is then utilized for our further processing for decisions

- **Green:** These are two rectangles on either side of the center. They determine whether we can turn right or left using the code defined here:

```
# Define bands and checks
center_band = road_mask[center_y - check_height:center_y, left_x:right_x]
center_clear = np.all(center_band == 255)


left_band = road_mask[detection_y - 66:detection_y + 66, 0:center_x]
right_band = road_mask[detection_y - 66:detection_y + 66, center_x:]
left_clear = not np.any(left_band == 0) and has_clear_gap(left_band)
right_clear = not np.any(right_band == 0) and has_clear_gap(right_band)
```

Part 3 – Road Decision:

The following code is now responsible for our directional control, as well as stopping.

```
# Object density in blue box
zero_ratio = np.sum(middle_roi == 0) / middle_roi.size

sudden_jump = zero_ratio - prev_zero_ratio > 0.4  # e.g., 40% increase
prev_zero_ratio = zero_ratio

if sudden_jump and zero_ratio >= 0.3:
    decision = "Emergency Stop"

if detect_obj_count > 150:
    decision = "Stopping"
elif 60 < detect_obj_count <= 150:
    decision = "Slowing Down"
else:
    decision = "Moving Forward"

if zero_ratio >= 0.6:
    if left_clear:
        decision = "Turning Left"
    elif right_clear:
        decision = "Turning Right"
    else:
        decision = "Stopping"
elif zero_ratio >= 0.3:
    decision = "Slowing Down"
elif not vertical_clear:
    decision = "Stopping"
else:
    if not (detect_obj_count > 150):
        decision = "Moving Forward"
```

First we determine the `zero_ratio`. The zero ratio is used to determine if we should turn and to identify a sudden person. A sudden spike in zero ratio means an object has suddenly entered our ROI. This requires our `"Emergency Brake".` Next, we check for our `object_count`. If the

object count gets greater than 150, we send a signal to stop, but if its lower than that but greater than 60, we send a signal to slow down. This check for objects is done to avoid sudden jerks of stopping and speeding up, and give a more gradual slow down. In case we have absolutely no objects, we move forward. Next, we check for middle blockage and if we have space to move left or right. In case we don't, we give the decision to stop. Similarly, if the zero ratio is greater than 0.3 and is not a sudden stop, we give the decision to slow down. And if all these checks are False, we end up with our final decision to **"Move Forward"**

Miscellaneous Code:

The following code is our misc code, for stuff like displaying fps and decision on screen, as well as small boundry box for some crude sort of "lane detection".

```
cv.putText(vis_overlay, f"Decision: {decision}", (30, 30),
cv.FONT_HERSHEY_SIMPLEX, 0.8, (0, 255, 255), 2)
cv.putText(vis_overlay, f"Object Count: {detect_obj_count}", (30, 60),
cv.FONT_HERSHEY_SIMPLEX, 0.8, (0, 255, 255), 2)
text_size = cv.getTextSize(f"Road: {road}", cv.FONT_HERSHEY_SIMPLEX, 0.8,
2)[0]
cv.putText(vis_overlay, f"Road: {road}", (w - text_size[0] - 10, 30),
cv.FONT_HERSHEY_SIMPLEX, 0.8, (0, 255, 255), 2)

fps = 1 / (new_frame_time - prev_frame_time + 1e-5)  # add small value to
avoid div by zero
fps_text = f"FPS: {int(fps)}"
cv.putText(vis_overlay, fps_text, (30, h - 30), cv.FONT_HERSHEY_SIMPLEX, 0.7,
(255, 255, 255), 2)

h, w = road_mask.shape
y_bottom = h - 10
y_upper = int(h * 0.75)

left_bottom, right_bottom = get_lane_edges(road_mask, y_bottom)
left_upper, right_upper = get_lane_edges(road_mask, y_upper)

if all(v is not None for v in [left_bottom, right_bottom, left_upper,
right_upper]):
    # Offset inward by 10 pixels for more accurate lane boundary
visualization
    left_bottom += 10
    right_bottom -= 10
    left_upper += 10
    right_upper -= 10
    cv.polylines(vis_overlay, [np.array([(left_bottom, y_bottom),
                                         (left_upper, y_upper),
                                         (right_upper, y_upper),
                                         (right_bottom, y_bottom)])],
                 isClosed=True, color=(0, 255, 0), thickness=2)
cv.imshow("Final Overlay", vis_overlay)
# cv.imshow("Segmented", segmented)
```

```
# cv.imshow("Road Mask", road_mask)
# cv.imshow("Final Road Overlay", overlay)
```

Youtube Link:
https://youtu.be/D4QxFiu2b9I

GitHub Repo:
https://github.com/WahaajNasir/PI-Navigation