Save as PDF

RU

hi i am a university student taking a module on "edge computing and analytics". i need your help guiding me through a lab. the lab is on "image analytics with raspberry pi using web camera".

#### Memory updated

Hey YR! That sounds like a fun lab — combining hardware and image processing. I'd be happy to guide you through it. Could you share the lab sheet or outline what the lab tasks are? For example:

- What are the objectives of the lab?
- What's the setup (e.g., Raspberry Pi model, camera type)?
- Are you using any specific libraries or tools (e.g., OpenCV, Python, TensorFlow Lite)?

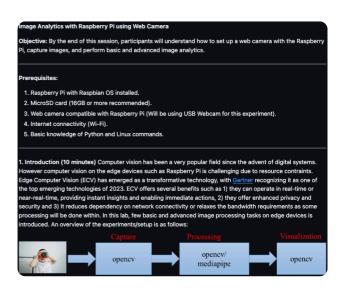
Once I know that, I can walk you through step-by-step.



okay i will be providing you the lab sheet in sections



Perfect! Send it over section by section, and I'll walk you through each part and help explain or guide you with the setup and tasks. Ready when you are!



Thanks for sharing the first section! Here's a quick summary and breakdown of what this lab involves so far:

#### **Solution** Lab Objective

You'll learn how to:

- 1. Set up a USB webcam with a Raspberry Pi.
- 2. Capture images using OpenCV.
- 3. Perform basic & advanced image analytics, possibly using OpenCV and MediaPipe.
- 4. Visualize processed output (likely with OpenCV as well).

## Prerequisites Recap

Make sure you have:

- A Raspberry Pi with Raspbian OS.
- At least a 16GB microSD card.
- A USB webcam.
- Internet access (for package installations or updates).
- Basic Python and Linux terminal skills.

#### Concept: Edge Computer Vision (ECV)

- ECV means doing computer vision tasks *on the edge device* (i.e., your Raspberry Pi) instead of sending everything to the cloud.
- Benefits:
  - Real-time processing.
  - Privacy and security.
  - Lower dependency on internet connectivity or cloud services.

# Pipeline Overview

Here's the step-by-step flow your lab will follow:

Let's keep going! When you're ready, upload or paste the next section (probably where you start coding or setting things up).



Great — this section is all about **preparing your Raspberry Pi** for the lab. Here's a step-by-step guide based on what's shown:

## 2. Setting up the Raspberry Pi (Estimated Time: 15 mins)

#### 1. Boot Up and Connect

- Power up your Raspberry Pi.
- Connect it to the internet via Wi-Fi or Ethernet.

#### 2. Update the System

Run these commands in the terminal:

```
bash

sudo apt update
sudo apt upgrade
```

This ensures your Pi has the latest package lists and upgrades installed software.

#### 3. Set Up Python Virtual Environment

This avoids library version conflicts and keeps your workspace clean.

#### Step-by-step:

```
# Install venv (if not already installed)
sudo apt install python3-venv

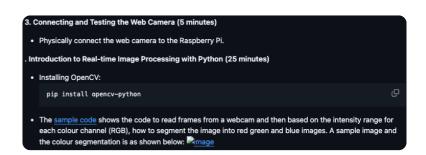
# Create a virtual environment named 'image'
python3 -m venv image

# Activate the virtual environment
source image/bin/activate
```

**▼ Tip:** Once activated, your terminal prompt should change to show (image) at the start — indicating you're in the virtual environment.

You're all set to begin installing libraries and writing code from here.

Ready for the next section whenever you are!



sample code:

# Reference: https://pyimagesearch.com/2014/08/04/opencvpython-color-detection/ import cv2 import numpy as np

#%% Defining a list of boundaries in the RGB color space # (or rather, BGR, since OpenCV represents images as NumPy arrays in reverse order)

# Refer to

```
https://docs.opencv.org/3.4/da/d97/tutorial threshold inRange.
html
boundaries = [
  ([17, 15, 100], [50, 56, 200]), # For Red
  ([86, 31, 4], [220, 88, 50]), # For Blue
  ([25, 90, 4], [62, 200, 50])] # For Green
#%% Normalize the Image for display (Optional)
def normalizelmg (lmg):
  Img= np.float64(Img) #Converting to float to avoid errors due
to division
  norm img = (Img - np.min(Img))/(np.max(Img) - np.min(Img))
  norm img = np.uint8(norm img*255.0)
  return norm img
#%% Open CV Video Capture and frame analysis
cap = cv2.VideoCapture(0)
# Check if the webcam is opened correctly
if not cap.isOpened():
  raise IOError("Cannot open webcam")
# The loop will break on pressing the 'q' key
while True:
  try:
     # Capture one frame
     ret, frame = cap.read()
     output=[]
     # loop over the boundaries
     for (lower, upper) in boundaries:
       # create NumPy arrays from the boundaries
       lower = np.array(lower, dtype = "uint8")
       upper = np.array(upper, dtype = "uint8")
       # find the colors within the specified boundaries and
apply the mask (basically segmenting for colours)
       mask = cv2.inRange(frame, lower, upper)
       output.append(cv2.bitwise_and(frame, frame, mask =
mask)) #Segmented frames are appended
     # Output is appeneded to be of size Pixels X 3 (for R, G,
B)
     red_img = normalizeImg(output[0])
     green_img = normalizeImg(output[1])
```

```
blue_img = normalizeImg(output[2])

# horizontal Concatination for displaying the images and colour segmentations
    catImg =

cv2.hconcat([frame,red_img,green_img,blue_img])
    cv2.imshow("Images with Colours",catImg)

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

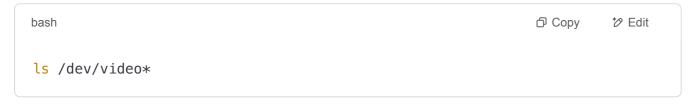
except KeyboardInterrupt:
    break

cap.release()
cv2.destroyAllWindows()
```

Awesome — now you're getting into the **real-time image processing part** using OpenCV! Let's go over the steps and the sample code in simple terms:

# Step 3: Connect & Test Webcam

- 1. Plug the USB webcam into your Raspberry Pi.
- 2. Make sure it's detected:



You should see something like /dev/video0.

## Image Processing with OpenCV (25 mins)

#### ☑ 1. Install OpenCV inside your virtual environment:

bash

pip install opencv-python

□ Copy ♡ Edit

## Sample Code Breakdown: Color Segmentation (RGB)

Here's what the code does step-by-step:

#### ● ● ● 1. Define RGB (BGR) color ranges

```
boundaries = [
    ([17, 15, 100], [50, 56, 200]), # Red
    ([86, 31, 4], [220, 88, 50]), # Blue
    ([25, 90, 4], [62, 200, 50]) # Green
]
```

• Each color has a lower and upper BGR boundary (OpenCV uses BGR, not RGB).

#### 2. Normalize for consistent display

```
python

def normalizeImg(Img):
```

• Optional step to rescale image values between 0-255 for visualization.

#### 3. Capture video from webcam

```
python

□ Copy ♡ Edit

cap = cv2.VideoCapture(0)
```

Captures real-time frames from the USB webcam.

#### 4. Apply color segmentation

```
python

mask = cv2.inRange(frame, lower, upper)
output.append(cv2.bitwise_and(frame, frame, mask=mask))
```

• Isolates only the pixels within the color range using a mask.

#### 5. Display side-by-side view

```
catImg = cv2.hconcat([frame, red_img, green_img, blue_img])
cv2.imshow("Images with Colours", catImg)
```

• Shows the original + segmented images for red, green, and blue channels in a single window.

#### 6. Press q to exit

```
python

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

break
```

# Test Tip:

Run the script and hold up red, green, or blue objects in front of the webcam to test live color segmentation!

Ready for the next part whenever you are!

• Expand the code to segment another colour (say yellow)

Perfect — this task is asking you to **expand the code to segment another color**, specifically **yellow**.

Here's how you can do that:

#### Add Yellow to the Color Boundaries

Update this part of the code:

```
python

boundaries = [
    ([17, 15, 100], [50, 56, 200]), # Red
    ([86, 31, 4], [220, 88, 50]), # Blue
```

```
([25, 90, 4], [62, 200, 50]), # Green
([0, 200, 200], [50, 255, 255]) # ♂ Yellow (new)
]
```

The [0, 200, 200] to [50, 255, 255] range in BGR roughly covers yellow, but you might need to fine-tune it based on lighting and webcam quality.

# Modify Display Section

#### Update this part:

```
# Normalize segmented images
red_img = normalizeImg(output[0])
green_img = normalizeImg(output[1])
blue_img = normalizeImg(output[2])
yellow_img = normalizeImg(output[3]) # New

# Display them horizontally
catImg = cv2.hconcat([frame, red_img, green_img, blue_img, yellow_img])
cv2.imshow("Images with Colours", catImg)
```

Now when you run the script and hold up something yellow (like a post-it or banana 🌭 ), it should be segmented and shown in the output window.

Let me know when you're ready to move on!



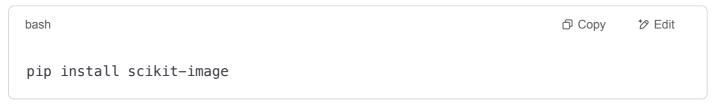
Nice — now you're moving on to **feature extraction** using **Histogram of Oriented Gradients (HoG)**. This is a classic technique in computer vision!

## Concept Recap: What is HoG?

- Histogram of Oriented Gradients (HoG) captures edge orientations in localized parts of an image.
- It's useful for tasks like face detection, object recognition, etc.
- The image is divided into small blocks (patches), and gradient directions (edges) are calculated for each block.
- It works best with **grayscale images**, so converting from RGB is necessary first.

# Step-by-Step Instructions

1. Install the required package:



Here's a simplified version of how to compute and display HoG features:

```
python

    □ Copy

                                                                           ⁺⊘ Edit
from skimage import color
from skimage.feature import hog
from skimage import io
import matplotlib.pyplot as plt
# Load an image (you can replace with frame from webcam)
image = io.imread('einstein.jpg') # Replace with your own image
gray_image = color.rgb2gray(image) # Convert to grayscale
# Compute HoG features
hog_features, hog_image = hog(gray_image,
                                orientations=8,
                                pixels_per_cell=(16, 16),
                                cells_per_block=(1, 1),
                                visualize=True)
# Display original and HoG image
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))
ax1.imshow(gray_image, cmap=plt.cm.gray)
ax1.set_title('Grayscale Image')
ax2.imshow(hog_image, cmap=plt.cm.gray)
ax2.set_title('HOG Features')
plt.show()
```

## Notes:

- You can replace io.imread() with a frame from OpenCV (cap.read()).
- Remember to convert BGR to RGB or grayscale as needed.
- visualize=True gives you the image used for display.

When you're ready, I can also help you integrate this with the live webcam stream. Just send the next part or let me know!

```
sorry below is the sample code. i want you to explain it to me in
detail.
sample code:
import cv2
import numpy as np
from skimage import feature # Refer to https://scikit-image.org/
for full capabalities of scikit-image library
from skimage import exposure
#%% Open CV Video Capture and frame analysis
cap = cv2.VideoCapture(0)
# Check if the webcam is opened correctly
if not cap.isOpened():
  raise IOError("Cannot open webcam")
# The loop will break on pressing the 'q' key
while True:
  try:
     # Capture one frame
     ret, frame = cap.read()
     # resizing for faster detection
     #frame = cv2.resize(frame, (256, 256)) #Uncomment and
see the speed up
     # Converting to gray scal as HOG feature extraction in
scikit-image works only on gray scale image
     image = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
     # Extact the HoG featues from the image
```

```
scikit-image works only on gray scale image
image = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

# Extact the HoG featues from the image
(H, hogImage) = feature.hog(image, orientations=9,
pixels_per_cell=(8, 8),
    cells_per_block=(2, 2), transform_sqrt=True,
block_norm="L1",
    visualize=True)

# Rescale intensity to be within 0-255 (contrast stretching)
hogImage = exposure.rescale_intensity(hogImage,
out_range=(0, 255))
hogImage = hogImage.astype("uint8")
```

```
# Converting gray to RGB
hogImg =
cv2.cvtColor(hogImage,cv2.COLOR_GRAY2RGB)

# Horizontal concatenation to show both input image and
its HoG features.
    catImg = cv2.hconcat([frame,hogImg])
    cv2.imshow("HOG Image", catImg)

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

except KeyboardInterrupt:
    break

cap.release()
cv2.destroyAllWindows()
```

No problem! Let's break this **HoG-based real-time webcam processing code** down **line-by-line** so you fully understand what's going on:

# 

```
import cv2
import numpy as np
from skimage import feature
from skimage import exposure
```

- cv2 OpenCV, used for capturing video and image operations.
- numpy for array manipulations (though not heavily used here).
- skimage.feature where the hog() function lives.
- skimage.exposure for **rescaling contrast** of the HoG image (to make it viewable).



```
python

cap = cv2.VideoCapture(0)

if not cap.isOpened():
    raise IOError("Cannot open webcam")
```

- Tries to open the default webcam ( 0 ).
- Throws an error if it fails.

# Main Loop: Real-Time Frame Processing

```
python
while True:
    try:
    ret, frame = cap.read()
```

- Continuously grabs frames from the webcam.
- frame is a NumPy array representing the current image.

# (Optional) Resize for Speed

 Uncommenting this would shrink the image, making HoG computation faster (but possibly losing detail).

## Convert Frame to Grayscale

```
python

image = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

• HoG requires grayscale images because it's based on intensity gradients, not color.

#### HoG Feature Extraction

This is the heart of the analysis. Let's break it down:

- orientations=9: Breaks gradient direction into **9 bins** (each 20° apart).
- pixels\_per\_cell=(8, 8): Each cell is 8×8 pixels.
- cells\_per\_block=(2, 2): HoG features are normalized over 2×2 cells.
- transform\_sqrt=True : Applies sqrt transform to improve lighting invariance.
- block\_norm='L1': Type of block normalization (helps standardize contrast).
- visualize=True: Returns an actual image representing HoG features.

H is the raw HoG feature vector (usually used for ML), hogImage is the image you can actually see.

## Normalize HoG for Display

```
python

hogImage = exposure.rescale_intensity(hogImage, out_range=(0, 255))
hogImage = hogImage.astype("uint8")
```

- **Rescales** pixel values to fit within 0–255 (the standard grayscale range).
- Converts float image to uint8, so it can be displayed with OpenCV.

# Convert HoG Image to RGB

• Converts grayscale to RGB just for **horizontal concatenation** with the original frame.

# Display: Side-by-Side

```
python

catImg = cv2.hconcat([frame, hogImg])
cv2.imshow("HOG Image", catImg)
```

• Shows the original frame next to the HoG feature visualization.

## Quit on Key Press

```
python

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

Pressing q exits the loop gracefully.

# **✓ Cleanup**

```
python

Copy be Edit

cap.release()

cv2.destroyAllWindows()
```

Releases the webcam and closes any OpenCV windows.

# **Summary**

This script does real-time:

- 1. Webcam capture with OpenCV.
- 2. Grayscale conversion for processing.
- 3. Feature extraction with HoG (edge/texture info).
- 4. Visualization of those features.
- 5. Combines both original and processed views side-by-side.

Let me know if you'd like to:

- Tune the HoG parameters.
- · Save frames/images.
- Integrate it into a face or object detection task. Or continue to the next lab section!

 Run the code with and without resizing the image and observe the resultant frame rate. It is important to note that for edge computing, downsizing the image will speed up the compute and many such informed decisions are critical.

Change the patch size in line 25 (feature.hog) and observe the changes in the results

This section gives you **two key performance tuning tasks** — great for understanding how optimization works in edge computing contexts like Raspberry Pi. Let's break them down:

#### 1. Resize vs. No Resize – Observe Frame Rate

python

# frame = cv2.resize(frame, (256, 256)) # Try enabling this line

#### Why it matters:

• Original frame might be large (e.g., 640×480 or higher), which slows down hog() computation.

- Resizing to smaller dimensions (like 256×256) reduces the number of pixels to process, resulting in:
  - Higher frame rate
  - Faster responsiveness
  - But potentially loss of detail

#### Try:

- Run the code with this line **commented out** (full-size image).
- Then uncomment it to resize and compare the speed and quality.

# 2. Change Patch Size in feature.hog()

```
pixels_per_cell=(8, 8)
cells_per_block=(2, 2)
```

#### Why it matters:

These define the **granularity** of feature extraction:

- pixels\_per\_cell → How big each "cell" is for computing gradients.
- cells\_per\_block  $\rightarrow$  How many cells to group for normalization.

#### Try tweaking:

Setting	What Happens
pixels_per_cell=(4, 4)	Smaller cells → finer detail, more compute
pixels_per_cell=(16, 16)	Larger cells → coarser features, faster
cells_per_block=(1, 1)	Less normalization, faster
cells_per_block=(3, 3)	More normalization, slower but potentially more stable features

#### 

- Observe changes in both performance (frame rate) and visual detail (HoG output).
- Think about the trade-offs between speed vs. accuracy key in edge computing!

Let me know if you'd like help logging frame rates, saving output, or integrating this with a detection model. Ready for the next section whenever you are!

can you explain cells\_per\_block in more detail? what does less or more normalisation mean?

Absolutely — this is a great question! Let's break down what <code>cells\_per\_block</code> in Histogram of Oriented Gradients (HoG) means, and what's really happening behind the scenes.

# ☐ First, Visualize the Breakdown

When HoG processes an image, it does so in this hierarchy:

- 1. **Image** → divided into small squares called **cells** (e.g. 8×8 pixels).
- 2. From each **cell**, a histogram of gradient directions is computed.
- 3. Then, a group of neighboring **cells is combined into a block**.
- 4. The **block is normalized** and that's where <code>cells\_per\_block</code> matters.
- cells\_per\_block=(2, 2) means:
- Each block will contain a 2×2 grid of cells (i.e., 4 cells total).
- If pixels\_per\_cell=(8,8), then a block would be:
  - $2\times8 = 16$  pixels wide and  $2\times8 = 16$  pixels tall.
- The block slides across the image, overlapping with others, computing gradients and then normalizing them.

# What is "Normalization"? Why does it matter?

Raw gradient values can vary a lot due to:

- · Lighting conditions
- Contrast

· Shadows, etc.

**Normalization** makes these values **comparable and stable** by adjusting them relative to surrounding values — much like adjusting brightness/contrast before feeding data into a model.

## **Example:**

- Imagine a cell has a histogram like [3, 5, 1] (gradient bins).
- Another cell has [30, 50, 10] same shape, just brighter image.
- Without normalization, they look very different.
- With normalization (dividing by block magnitude), they become more similar, focusing on pattern, not brightness.

## More normalization (larger cells\_per\_block):

- Pros:
  - Smoother, more stable features.
  - Better performance in variable lighting.
- Cons:
  - Slower.
  - Might blur local features too much.

## Less normalization (smaller cells\_per\_block):

- Pros:
  - Faster.
  - · Retains very localized detail.
- Cons:
  - More sensitive to noise and lighting.

# TL;DR — Quick Rule of Thumb

cells_per_block	Speed	Detail	Robustness
(1, 1)	Fast	High	Low
(2, 2) (default)	Balanced	Good	Good
(3, 3) or more	Slower	Lower	High

Let me know if you want to visualize this with diagrams or run an experiment to compare performance!

```
example code:
import cv2
import numpy as np
# initialize the HOG descriptor/person detector
hog = cv2.HOGDescriptor()
hog.setSVMDetector(cv2.HOGDescriptor getDefaultPeopleDet
ector())
#sets how many pixels away from the center a person needs to
be before the head stops
center_tolerance = 5;
#%% Open CV Video Capture and frame analysis
cap = cv2.VideoCapture(0)
# Check if the webcam is opened correctly
if not cap.isOpened():
  raise IOError("Cannot open webcam")
# The loop will break on pressing the 'q' key
while True:
  try:
    # Capture one frame
    ret, frame = cap.read()
    # resizing for faster detection
```

```
frame = cv2.resize(frame, (256, 256)) #Uncomment and
see the speed up
    # detect people in the image
     # returns the bounding boxes for the detected objects
     boxes, weights = hog.detectMultiScale(frame, winStride=
(1,1), scale = 1.05)
     boxes = np.array([[x, y, x + w, y + h] for (x, y, w, h) in
boxes])
     centers = []
     for box in boxes:
       #get the distance from the center of each box's center x
cord to the center of the screen and ad them to a list
       center x = ((box[2]-box[0])/2)+box[0]
       x pos rel center = (center x-70)
       dist to center x = abs(x pos rel center)
       centers.append({'box': box, 'x pos rel center':
x pos rel center, 'dist to center x':dist to center x})
     if len(centers) > 0:
         #sorts the list by distance to center
       sorted boxes = sorted(centers, key=lambda i:
i['dist to center x'])
       #draws the box
       center box = sorted boxes[0]['box']
       for box in range(len(sorted_boxes)):
       # display the detected boxes in the colour picture
          if box == 0:
            cv2.rectangle(frame, (sorted_boxes[box]['box']
[0],sorted_boxes[box]['box'][1]), (sorted_boxes[box]['box']
[2],sorted_boxes[box]['box'][3]), (0,255, 0), 2)
          else:
            cv2.rectangle(frame, (sorted_boxes[box]['box']
[0],sorted boxes[box]['box'][1]), (sorted boxes[box]['box']
[2],sorted_boxes[box]['box'][3]),(0,0,255),2)
       #retrieves the distance from center from the list and
determins if the head should turn left, right, or stay put and turn
lights on
       Center_box_pos_x = sorted_boxes[0]
['x_pos_rel_center']
       if -center tolerance <= Center box pos x <=
center tolerance:
          #turn on eye light
          print("center")
       elif Center_box_pos_x >= center_tolerance:
          #turn head to the right
          print("right")
```

```
elif Center_box_pos_x <= -center_tolerance:
          #turn head to the left
          print("left")
       print(str(Center box pos x))
    else:
       #prints out that no person has been detected
       print("nothing detected")
    #resizes the video so its easier to see on the screen
    frame = cv2.resize(frame,(720,720))
    # Display the resulting frame
    cv2.imshow("frame",frame)
    if cv2.waitKey(1) \& 0xFF == ord('q'):
      break
  except KeyboardInterrupt:
    break
cap.release()
cv2.destroyAllWindows()
```

Perfect — this final part demonstrates **real-time human detection using OpenCV's built-in HoG + SVM detector**, and it's a great example of how HoG can be used for **face/body presence detection**. Let's break it down step by step.

## What This Code Does

- Uses OpenCV's built-in HoG+SVM person detector.
- Tracks detected people and identifies the one closest to the center of the screen.
- Displays bounding boxes:
  - Green for the center-most person
  - Red for others
- Prints "left", "right", or "center" depending on where that person is relative to screen center.

#### Netup

```
python

Docopy ♥ Edit

hog = cv2.H0GDescriptor()

hog.setSVMDetector(cv2.H0GDescriptor_getDefaultPeopleDetector())
```

 Initializes the Histogram of Oriented Gradients descriptor with a pre-trained SVM (Support Vector Machine) model for people detection.

## **■ Webcam and Frame Capture**

```
python

cap = cv2.VideoCapture(0)

if not cap.isOpened():
    raise IOError("Cannot open webcam")
```

#### Frame Preprocessing

```
python

frame = cv2.resize(frame, (256, 256))
```

• Downsizes the frame for **faster detection** (a common edge computing trick).

#### Detecting People

- Detects people using multi-scale HoG:
  - winStride=(1,1) → Slide window 1 pixel at a time.
  - scale=1.05 → Build an image pyramid and check at different resolutions.

```
boxes = np.array([[x, y, x + w, y + h] for (x, y, w, h) in boxes])
```

Converts (x, y, w, h) to (x1, y1, x2, y2) format.

## Calculating Center & Distance

- Calculates the horizontal center of each bounding box.
- Then, calculates how far it is from an assumed screen center (x=70).

```
python

dist_to_center_x = abs(x_pos_rel_center)

Discopy ♥ Edit
```

Stores the absolute distance from center — used to find the closest person to center.

# Sort and Display Boxes

Sorts detections by closeness to screen center.

- Draws **green box** for the closest detection (presumably the one being tracked).
- Draws red boxes for the rest.

## Decision Logic – Head Position

```
if -center_tolerance <= Center_box_pos_x <= center_tolerance:
    print("center")
elif Center_box_pos_x >= center_tolerance:
    print("right")
elif Center_box_pos_x <= -center_tolerance:
    print("left")</pre>
```

- Based on how off-center the closest person is, the system suggests:
  - Turn left, right, or stay centered
- This could control a robot head or light in a real setup.

# Final Display

```
python

frame = cv2.resize(frame,(720,720))
cv2.imshow("frame", frame)
```

Enlarges the final result window for easier viewing.

#### Exit

```
python

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



The detectMultiScale() function automatically runs **HoG feature extraction at different image scales** to detect:

- Both small and large people in the frame.
- Like zooming in/out during detection.

# **Summary**

Part	What It Does
HOGDescriptor()	Initializes a HoG detector
detectMultiScale()	Performs person detection with sliding window & pyramid
Bounding boxes	Draws on screen to show who was found
Logic	Determines left/right/center offset
Practical Use	Could control camera, lights, or robotic head

Let me know if you'd like to:

- Control hardware (like servos) based on detection.
- Extend this to track multiple people.
- Replace people detection with face detection using cv2.CascadeClassifier.

Ready to move on or dive deeper if you'd like!

can you explain the code in more detail?

Absolutely, YR — let's dive **deeper** into the logic, line by line, block by block. The script you're working with is an awesome real-time **people detection** application using **OpenCV's built-in HOG + SVM person detector**, and includes **positional tracking logic** that could be used to control a robot's head or a spotlight, for instance.

# Section-by-Section Detailed Explanation

#### 1. Importing Libraries

```
python

import cv2
import numpy as np
```

- cv2 : OpenCV, used for video capture, resizing, object detection, and drawing on frames.
- numpy: Used to manipulate arrays, especially for working with bounding boxes.

#### 2. Initializing the HOG Person Detector

```
python

hog = cv2.H0GDescriptor()

hog.setSVMDetector(cv2.H0GDescriptor_getDefaultPeopleDetector())
```

- cv2.H0GDescriptor(): This creates a Histogram of Oriented Gradients (HOG) descriptor object.
- .setSVMDetector(...): Loads a **pre-trained SVM model** trained to detect pedestrians.

  OpenCV already includes this detector trained on the INRIA Person Dataset.
- ▶ HOG + SVM: HOG captures edge-orientation features, and SVM is a classifier trained to detect "people-shaped" gradients.

#### 3. Camera Setup

```
python

cap = cv2.VideoCapture(0)
if not cap.isOpened():
    raise IOError("Cannot open webcam")
```

- Opens the default webcam ( device 0 ).
- If the webcam fails to initialize, it raises an error.

# 4. Real-Time Loop for Frame-by-Frame Processing

#### This loop:

- Captures frames in real time.
- Detects people.
- Calculates how centered the closest person is.
- Provides directional feedback (left, right, center).
- Draws boxes.

# ♣ 5. Frame Capture and Optional Resize

```
python

ret, frame = cap.read()

frame = cv2.resize(frame, (256, 256))
```

- ret confirms the frame was captured successfully.
- Resizing to 256×256 helps speed up detection, especially useful on low-power edge devices (like Raspberry Pi).

# & 6. People Detection using HOG

- detectMultiScale: Runs the HoG-based sliding window over the image to detect people.
- winStride=(1,1): Moves window 1 pixel at a time (very precise, but slower).
- scale=1.05: Builds an image pyramid to detect people at different scales (sizes).

#### 7. Format the Boxes

```
python
                                                                  Copy
                                                                            ⁺⊘ Edit
boxes = np.array([[x, y, x + w, y + h] for (x, y, w, h) in boxes])
```

Converts (x, y, w, h) to (x1, y1, x2, y2) format for easier drawing later.

#### 8. Calculate Center Offsets

```
python
                                                                  □ Copy
                                                                            ⁺⊘ Edit
centers = []
for box in boxes:
    center_x = ((box[2]-box[0])/2) + box[0]
    x_pos_rel_center = (center_x - 70)
    dist_to_center_x = abs(x_pos_rel_center)
    centers.append({'box': box, 'x_pos_rel_center': x_pos_rel_center,
'dist_to_center_x':dist_to_center_x})
```

#### Let's break this down:

- center\_x : Calculates the X coordinate of the center of the detected person.
- x\_pos\_rel\_center: How far the person is from the **reference center point** (assumed to be x = 70 — for 256px wide image, this is close to center).
- dist to center x: Just the **absolute distance**, used to find who is closest to the center.

#### Each detected person is stored with:

- Their bounding box
- Their position relative to center
- Their distance from center

#### 9. If Someone Is Detected...

```
python

if len(centers) > ∅:
```

If any people are detected...

## 10. Sort by Distance to Center & Draw Boxes

```
python

Sorted_boxes = sorted(centers, key=lambda i: i['dist_to_center_x'])

center_box = sorted_boxes[0]['box']
```

- sorted\_boxes: A list of detections, sorted by who is closest to screen center.
- The first box in that list is the primary target.

```
python

for box in range(len(sorted_boxes)):
   if box == 0:
      cv2.rectangle(...) # Green box for center-most
   else:
      cv2.rectangle(...) # Red box for others
```

- Draws bounding boxes on all detections:
  - Green for the person closest to center.
  - Red for the rest.

#### in 11. Determine Direction: Left / Right / Center

python 🗗 Copy 🤣 Edit

```
Center_box_pos_x = sorted_boxes[0]['x_pos_rel_center']
if -center_tolerance <= Center_box_pos_x <= center_tolerance:
    print("center")
elif Center_box_pos_x >= center_tolerance:
    print("right")
elif Center_box_pos_x <= -center_tolerance:
    print("left")</pre>
```

- center\_tolerance: A threshold range around the center. If the person is within this, they're "centered".
- Prints:
  - "center": If they're mostly aligned.
  - "right": If they're to the right of the center.
  - "left": If they're to the left.
- You could use this logic to:
- Turn a camera head using a servo
- Trigger a spotlight
- Send movement commands to a robot

# 12. Resize for Display & Show Frame

```
python

frame = cv2.resize(frame, (720, 720))
cv2.imshow("frame", frame)
```

• Upscales the frame to 720×720 just for display (easier to see).

# X 13. Exit on Keypress or Interrupt

```
python

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

break

Press q to stop the loop.

# √ 14. Clean Up

```
python

cap.release()

cv2.destroyAllWindows()
```

Releases webcam and closes OpenCV windows.

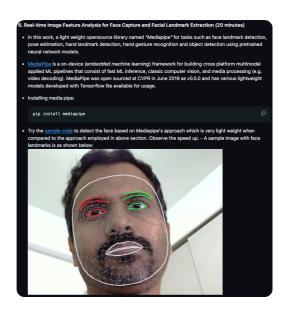
# **Summary**

Functionality	Purpose
HOG + SVM	Real-time people detection
Distance-from-center logic	Find the "main" person in view
Direction logic	Decide left/center/right based on X-position
Drawing + output	Visually show what's happening

Let me know if you'd like help:

- Integrating this with servo motors (e.g., Pi-controlled head movement),
- Saving detection data (e.g., to CSV or log),
- Adding face detection instead of person detection.

Happy to take it to the next level with you!



i want you to explain the sample code to me in detail.

```
sample code:
import cv2
import mediapipe as mp
# Initialize Mediapipe Face Mesh
mp face mesh = mp.solutions.face mesh
face_mesh =
mp_face_mesh.FaceMesh(static_image_mode=False,
                    max num faces=1,
                    min_detection_confidence=0.5,
                    min_tracking_confidence=0.5)
mp_drawing = mp.solutions.drawing_utils
mp_drawing_styles = mp.solutions.drawing_styles
# Open the camera feed
cap = cv2.VideoCapture(0)
if not cap.isOpened():
  print("Error: Could not access the camera.")
  exit()
while cap.isOpened():
  ret, frame = cap.read()
  if not ret:
    print("Failed to grab frame.")
    break
  # Convert frame to RGB
  rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
```

```
# Process the frame
  results = face mesh.process(rgb_frame)
  # Draw landmarks
  if results.multi face landmarks:
    for face landmarks in results.multi face landmarks:
       mp drawing.draw landmarks(
         image=frame,
         landmark list=face landmarks,
connections=mp_face_mesh.FACEMESH_TESSELATION,
         landmark drawing spec=None,
connection drawing spec=mp drawing styles.get default fac
e mesh tesselation style()
      )
      mp_drawing.draw_landmarks(
         image=frame,
         landmark list=face landmarks,
connections=mp face mesh.FACEMESH CONTOURS,
         landmark drawing spec=None,
connection drawing spec=mp drawing styles.get default fac
e_mesh_contours_style()
      )
  # Display the frame
  cv2.imshow('Mediapipe Face Mesh', frame)
  # Exit on pressing 'q'
  if cv2.waitKey(5) \& 0xFF == ord('q'):
    break
cap.release()
cv2.destroyAllWindows()
```

Awesome! This final section is all about using **MediaPipe** — a modern, lightweight, real-time framework for computer vision tasks like **face mesh and landmark detection**.

Let me walk you through the **sample code in full detail**, so you understand everything it's doing.

## What the Code Does

- It captures video from your webcam.
- Uses MediaPipe's Face Mesh model to detect facial landmarks in real time.
- Draws **facial tesselation** (a fine mesh of triangles) and **facial contours** (eyebrows, eyes, lips, etc.) on the video feed.
- Displays the annotated output live.

# Line-by-Line Breakdown

#### \*

#### 1. Import Required Libraries

```
python

import cv2
import mediapipe as mp

import mediapipe as mp
```

- cv2 : For webcam input, image processing, and display.
- mediapipe: Provides pre-trained ML models for face/hand/pose detection and drawing utilities.

# ♦ 2. Initialize MediaPipe Face Mesh

```
python

p
```

#### Parameter explanations:

- static\_image\_mode=False: Enables video stream mode faster, more efficient. If set to
   True, it assumes every frame is a new image.
- max\_num\_faces=1 : Only track **1 face**. You can increase it for multi-person detection.
- min\_detection\_confidence=0.5: Threshold for initial face detection (higher = stricter).
- min\_tracking\_confidence=0.5: Threshold for tracking the face over time.

### 3. Drawing Utilities

```
mp_drawing = mp.solutions.drawing_utils
mp_drawing_styles = mp.solutions.drawing_styles
```

• These modules help draw **face mesh landmarks and connections** with predefined styles (colors, thicknesses, etc.).

### 4. Open Webcam

```
python

cap = cv2.VideoCapture(0)

if not cap.isOpened():

print("Error: Could not access the camera.")

exit()
```

- Opens the webcam stream.
- Exits the program gracefully if the webcam fails to initialize.

## 5. Frame-by-Frame Loop

```
python

☐ Copy 炒 Edit

while cap.isOpened():
```

```
ret, frame = cap.read()
```

Captures frames from the webcam in a continuous loop.

#### **6. Convert BGR to RGB**

 MediaPipe models expect RGB input, but OpenCV gives us BGR, so we need to convert the format.

#### 7. Process Frame with Face Mesh Model

```
python

Copy by Edit

results = face_mesh.process(rgb_frame)
```

- Runs face landmark detection on the RGB frame.
- results.multi\_face\_landmarks will contain facial keypoints for each detected face.

#### 8. Draw Landmarks

```
python

if results.multi_face_landmarks:
    for face_landmarks in results.multi_face_landmarks:
        mp_drawing.draw_landmarks(...)
        mp_drawing.draw_landmarks(...)
```

#### Two types of overlays:

1. Tesselation:

connections=mp\_face\_mesh.FACEMESH\_TESSELATION

• A mesh of ~468 landmarks, connected to form a wireframe over the face.

#### 2. Contours:

```
python

Copy * Edit

connections=mp_face_mesh.FACEMESH_CONTOURS
```

Key outline areas: lips, eyebrows, eyes, nose, jawline, etc.

These styles make it visually easy to understand facial structure and movement.

### 9. Display Output Frame

Displays the annotated frame with mesh and contours in a live window.

#### = 10. Exit on 'q' Key Press

• Waits 5 ms between frames and exits the loop if 'q' is pressed.

#### √ 11. Cleanup

```
cap.release()
cv2.destroyAllWindows()
```

Releases webcam resources and closes the window.

# Why MediaPipe is Awesome for Edge

- Lightweight: Much faster than traditional HoG + SVM + tracking approaches.
- Accurate: Detects 468 facial landmarks with high precision.
- Efficient: Runs well on Raspberry Pi, Android, even in browsers via WebAssembly!

#### You Can Extend This To:

- Control expressions/emotions analysis.
- Track head pose.
- Run filters (Snapchat-style) by anchoring graphics to landmarks.
- Build an attention tracker, drowsiness detector, or face unlock system.

Let me know if you want to go into:

- Saving facial keypoints,
- Using multiple faces,
- Or linking face movement to a robotic actuator.

You've officially reached pro-level face analysis now

[Optional] An opency alternative (no dependence on mediapipe) of the face detection is available in the
 <u>sample code</u>. If you are using this code, make sure you download the <u>Haar cascade model</u> manually and save
 it as 'haarcascade\_frontalface\_alt2.xml' in the same folder as the code.

i want you to explain the sample code to me in detail.

#### sample code:

# For the code to work the Open source Haar Cascade model has to be downloaded and kept in the same folder.

# Please download the .xml from

https://raw.githubusercontent.com/opencv/opencv/master/data/

```
haarcascades/haarcascade_frontalface_alt2.xml
import cv2
# Initiate the Face Detection Cascade Classifier
haarcascade = "haarcascade frontalface alt2.xml"
detector = cv2.CascadeClassifier(haarcascade)
#%% Open CV Video Capture and frame analysis
cap = cv2.VideoCapture(0)
# Check if the webcam is opened correctly
if not cap.isOpened():
  raise IOError("Cannot open webcam")
# The loop will break on pressing the 'q' key
while True:
  try:
    # Capture one frame
    ret, frame = cap.read()
    # resizing for faster detection
    frame = cv2.resize(frame, (256, 256)) #Comment and see
the speed up
    # Converting to gray scale as feature extraction works
only on gray scale image
    image_gray = cv2.cvtColor(frame,
cv2.COLOR BGR2GRAY)
    # Detect faces using the haarcascade classifier on the
"grayscale image"
    faces = detector.detectMultiScale(image_gray)
    # returns the bounding boxes for the detected objects and
display them using rectangles
    for face in faces:
       (x,y,w,d) = face
       # Draw a white coloured rectangle around each face
using the face's coordinates
       # on the "image_template" with the thickness of 2
       cv2.rectangle(frame,(x,y),(x+w, y+d),(255, 255, 255), 2)
    #resizes the video so its easier to see on the screen
```

frame = cv2.resize(frame,(720,720))

```
# Display the resulting frame
    cv2.imshow("frame",frame)

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

except KeyboardInterrupt:
    break

cap.release()
cv2.destroyAllWindows()
```

Great! You're now looking at an **OpenCV-only alternative** to face detection — no MediaPipe, just traditional **Haar Cascades**, which are simple, efficient, and work surprisingly well for many basic applications.

Let's break the code down **line-by-line** and explain what each part does and how it all fits together:

## What Is Haar Cascade?

Haar Cascades are an **old-school computer vision method** (before deep learning became popular). They use simple, fast classifiers trained with thousands of face/non-face examples. Still effective for real-time, lightweight face detection — especially on Raspberry Pi or low-resource devices.



#### 1. Import OpenCV

python ☐ Copy 🍪 Edit
import cv2

 Brings in the OpenCV library which provides webcam access, image processing, and the face detection tools you'll be using.

#### 2. Load Haar Cascade Classifier

```
python

haarcascade = "haarcascade_frontalface_alt2.xml"

detector = cv2.CascadeClassifier(haarcascade)
```

- This loads a pre-trained Haar cascade model from the .xml file.
- The model (haarcascade\_frontalface\_alt2.xml) contains rules to detect frontal faces.
- CascadeClassifier creates an object we can use to run detection on images.
- Important: You must place the .xml file in the same folder as your Python script.

#### 3. Initialize Webcam

```
python

cap = cv2.VideoCapture(0)

if not cap.isOpened():

raise IOError("Cannot open webcam")
```

- Starts capturing video from the default webcam ( 0 ).
- If it fails, the program raises an error and exits.

## **2** 4. Main Loop: Frame-by-Frame Analysis

```
python

while True:
    try:
    ret, frame = cap.read()
```

• Grabs a frame from the webcam on each loop iteration.

### 5. Resize Frame for Speed

```
python

frame = cv2.resize(frame, (256, 256))
```

- Resizing makes detection faster and less computationally heavy.
- You can comment it out to see the effect on detection accuracy and frame rate.

### 6. Convert to Grayscale

```
python

image_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

by Edit
```

- Haar Cascade detectors only work on grayscale images.
- This simplifies processing, since color isn't needed for detecting facial structure.

#### 7. Detect Faces



#### This is the key function:

- Scans the image using sliding windows at multiple scales.
- Returns a list of rectangles (x, y, w, h) where faces are detected.
- You can also pass in extra parameters like scaleFactor, minNeighbors, minSize to finetune detection.

### 8. Draw Rectangles

```
python

for face in faces:
    (x,y,w,d) = face
    cv2.rectangle(frame,(x,y),(x+w, y+d),(255, 255, 255), 2)
```

- Loops over each detected face and draws a **white rectangle** with thickness = 2.
- cv2.rectangle(...) takes top-left and bottom-right coordinates.

### 9. Display the Frame

```
python

frame = cv2.resize(frame,(720,720))
cv2.imshow("frame", frame)
```

- Enlarges the output frame so it's easier to view.
- Displays the video in a window titled "frame".

#### 10. Quit on Key Press

```
python

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

Pressing 'q' will break the loop and stop detection.

### √ 11. Cleanup

python 🗗 Copy 🤣 Edit

```
cap.release()
cv2.destroyAllWindows()
```

- · Releases the webcam.
- Closes all OpenCV windows.

# Summary

Feature	Description	
Haar Cascade	Fast and lightweight face detector	
XML model	Contains pre-trained face detection rules	
Grayscale conversion	Needed for Haar to work	
Rectangle drawing	Visual feedback of detected faces	
Real-time	Efficient enough for basic use on edge devices	

# **☑** Bonus: Optional Detector Tuning

If you want better control or accuracy:

```
python

faces = detector.detectMultiScale(
   image_gray,
   scaleFactor=1.1,
   minNeighbors=5,
   minSize=(30, 30)
)
```

Parameter	Meaning	
scaleFactor	How much image is reduced at each scale (1.1 = 10%)	
minNeighbors	How many overlapping detections are needed to confirm a face	
minSize	Minimum window size (helps ignore small false positives)	

Let me know if you want to:

- Compare Haar vs MediaPipe side-by-side,
- Extract and log face positions,
- Or try face recognition next.

You've now mastered 3 methods of face detection — you're leveling up fast!

[Optional] Homework/Extended Activities:
 1. Explore more advanced OpenCV functionalities like SIFT, SURF, and ORB for feature detection. These features alongside HoG could be used for image matching (e.g. face recognition)

Awesome — you've now reached the **Optional Homework / Advanced Section**, which introduces some **next-level OpenCV tools** for feature detection and image matching: **SIFT**, **SURF**, and **ORB**. Here's a breakdown of what these are and how they compare, especially in the context of **face recognition**, **object detection**, or **image matching**.

## **Q** What Are SIFT, SURF, and ORB?

These are **feature detectors and descriptors** used to extract **keypoints** and describe them in a way that allows comparison between different images.

Technique	Full Name	Strengths	License
SIFT	Scale-Invariant Feature Transform	Highly accurate, rotation + scale invariant	Patented (was non-free, now open in OpenCV ≥4.4)
SURF	Speeded-Up Robust Features	Faster than SIFT, also scale/rotation invariant	Still patented (not included in OpenCV by default)
ORB	Oriented FAST and Rotated BRIEF	Open-source, fast, good for real-time apps	Free (ideal for Raspberry Pi)

## **What You Can Do with Them:**

- Detect features in two images (e.g. face in webcam vs stored template)
- Match them using descriptors
- Track objects, recognize items, perform stitching or alignment

## Example Use Case: ORB for Matching

```
python

    □ Copy

                                                                           ⁺⊘ Edit
import cv2
# Load images
img1 = cv2.imread('face1.jpg', 0) # Query image
img2 = cv2.imread('face2.jpg', 0) # Train image
# ORB detector
orb = cv2.ORB_create()
# Find the keypoints and descriptors
kp1, des1 = orb.detectAndCompute(img1, None)
kp2, des2 = orb.detectAndCompute(img2, None)
# Brute-force matcher
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
matches = bf.match(des1, des2)
# Sort matches by distance (lower = better)
matches = sorted(matches, key=lambda x: x.distance)
# Draw top 10 matches
matched_img = cv2.drawMatches(img1, kp1, img2, kp2, matches[:10], None,
flags=2)
cv2.imshow("Matches", matched_img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

## **#** Homework Ideas

- 1. Compare **ORB vs SIFT** in detecting features on faces.
- 2. Try to **match webcam input** to a stored face image (e.g., for basic face recognition).
- 3. Extend your **Haar or MediaPipe script** to capture and save face ROIs, then match with SIFT/ORB.

Would you like help setting up a working demo of one of these (like ORB matching with live webcam capture)? Just say the word, and I'll walk you through it!

yes i would like you to help me set up a working demo of one of these

Awesome, let's build a working demo using ORB (Oriented FAST and Rotated BRIEF) to do real-time face matching using your webcam. ORB is fast, lightweight, and works great on the Raspberry Pi — perfect for your lab setup.

## Goal of This Demo

#### We'll do:

- 1. Capture a **reference face image** (e.g., your face from the webcam).
- 2. Use ORB to extract features from:
  - The reference image, and
  - Live webcam frames.
- 3. Compare features and draw **matching keypoints** in real time.
- 4. Print **match score** as a basic face recognition indicator.

# X Step-by-Step Guide

### Step 1: Prepare Your Environment

Make sure OpenCV is installed with ORB:

bash 🗗 Copy 🤣 Edit

#### Step 2: Capture Reference Face (one-time setup)

Save this as capture\_reference\_face.py:

```
    □ Copy

                                                                             ⁺⊘ Edit
python
import cv2
cap = cv2.VideoCapture(0)
print("Press 's' to save your face, or 'q' to quit.")
while True:
    ret, frame = cap.read()
    cv2.imshow("Capture Reference Face", frame)
    key = cv2.waitKey(1)
    if key == ord('s'):
        # Save the face image
        cv2.imwrite("reference_face.jpg", frame)
        print("Reference image saved!")
        break
    elif key == ord('q'):
        break
cap.release()
cv2.destroyAllWindows()
```

Run this, press s when your face is centered in the frame — it will save reference\_face.jpg.

### Step 3: ORB-Based Real-Time Matching

Save this as orb\_face\_match.py:

python 🗗 Copy 🤣 Edit

```
import cv2
# Load reference image and convert to grayscale
reference_img = cv2.imread("reference_face.jpg")
reference_gray = cv2.cvtColor(reference_img, cv2.COLOR_BGR2GRAY)
# Initialize ORB detector
orb = cv2.0RB_create(nfeatures=1000)
ref_kp, ref_des = orb.detectAndCompute(reference_gray, None)
# Initialize matcher
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
cap = cv2.VideoCapture(0)
while True:
    ret, frame = cap.read()
    frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    # Detect keypoints and descriptors in current frame
    kp, des = orb.detectAndCompute(frame_gray, None)
    if des is not None and ref_des is not None:
        matches = bf.match(ref_des, des)
        matches = sorted(matches, key=lambda x: x.distance)
        # Draw top 20 matches
        matched_img = cv2.drawMatches(reference_img, ref_kp, frame, kp,
matches[:20], None, flags=2)
        # Show match quality
        match_quality = sum([m.distance for m in matches[:20]]) /
len(matches[:20])
        cv2.putText(matched_img, f"Match Score: {int(match_quality)}", (10,
30),
                    cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2)
        cv2.imshow("ORB Face Matching", matched_img)
    else:
        cv2.imshow("ORB Face Matching", frame)
    if cv2.waitKey(1) \& 0xFF == ord('q'):
        break
```

```
cap.release()
cv2.destroyAllWindows()
```

#### Match Score Interpretation

- Lower score = better match.
- You can experiment with thresholds like:

```
python

if match_quality < 50:
    print("Match found!")
else:
    print("No match")</pre>
```

#### What You've Achieved

- Real-time face matching using ORB descriptors.
- · Fast enough for Raspberry Pi.
- Fully OpenCV-only no need for MediaPipe or deep learning.

#### Would you like to:

- Use this for multiple people?
- Save multiple reference faces?
- Trigger some action (e.g. light or alert) if match is found?

Let me know how you'd like to extend this!

2. Build an eye blink detection system for drowsiness detection.

Perfect — this is a classic and very practical project:



Let me guide you through building a simple but effective version using **MediaPipe**, which makes facial landmark tracking easy, fast, and Raspberry Pi–friendly.

## How It Works (Concept)

We use **eye aspect ratio (EAR)** to detect blinking:

- You track 6 landmarks per eye (MediaPipe gives you those).
- Compute the ratio of vertical eye height to horizontal width.
- If this ratio drops below a threshold (eye closed), we count a blink.
- If the eye stays closed for a long time → likely drowsiness.

## Requirements

Install MediaPipe and OpenCV:

bash

pip install mediapipe opencv-python

□ Copy ♡ Edit

## **Step-by-Step Blink Detection Code**

Save this as eye\_blink\_drowsiness.py:

```
import cv2
import mediapipe as mp
import math
import time

# Initialize mediapipe face mesh
```

```
mp_face_mesh = mp.solutions.face_mesh
face_mesh = mp_face_mesh.FaceMesh(max_num_faces=1, refine_landmarks=True)
mp_drawing = mp.solutions.drawing_utils
# Eye landmark indices (MediaPipe face mesh)
LEFT_EYE = [33, 160, 158, 133, 153, 144]
RIGHT_EYE = [362, 385, 387, 263, 373, 380]
# EAR calculation
def calculate_EAR(landmarks, eye_indices, image_w, image_h):
    p1 = landmarks[eye_indices[0]]
    p2 = landmarks[eye_indices[1]]
    p3 = landmarks[eye_indices[2]]
    p4 = landmarks[eye_indices[3]]
    p5 = landmarks[eye_indices[4]]
    p6 = landmarks[eye_indices[5]]
    # Convert to pixel coordinates
    def to_pixel(p): return (int(p.x * image_w), int(p.y * image_h))
    # Vertical distances
    A = math.dist(to_pixel(p2), to_pixel(p6))
    B = math.dist(to_pixel(p3), to_pixel(p5))
    # Horizontal distance
    C = math.dist(to_pixel(p1), to_pixel(p4))
    EAR = (A + B) / (2.0 * C)
    return EAR
# Blink detection thresholds
EAR\_THRESHOLD = 0.22
CONSEC_FRAMES = 10
blink_count = 0
frame_counter = 0
drowsy = False
# Start webcam
cap = cv2.VideoCapture(0)
start_time = time.time()
while cap.isOpened():
    success, frame = cap.read()
    if not success:
        break
```

```
frame = cv2.flip(frame, 1)
    rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    results = face_mesh.process(rgb_frame)
    h, w, _ = frame.shape
    if results.multi_face_landmarks:
        for landmarks in results.multi_face_landmarks:
            left_EAR = calculate_EAR(landmarks.landmark, LEFT_EYE, w, h)
            right_EAR = calculate_EAR(landmarks.landmark, RIGHT_EYE, w, h)
            avg_EAR = (left_EAR + right_EAR) / 2.0
            if avg_EAR < EAR_THRESHOLD:</pre>
                frame_counter += 1
                if frame_counter >= CONSEC_FRAMES:
                    drowsy = True
            else:
                if frame_counter >= 1:
                    blink_count += 1
                frame_counter = 0
                drowsy = False
            # Draw landmarks
            mp_drawing.draw_landmarks(
                frame,
                landmarks,
                mp_face_mesh.FACEMESH_TESSELATION,
                mp_drawing.DrawingSpec(color=(0, 255, 0), thickness=1,
circle_radius=1),
            )
            # Show EAR and drowsiness warning
            cv2.putText(frame, f"EAR: {avg_EAR:.2f}", (30, 30),
cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 255, 0), 2)
            cv2.putText(frame, f"Blinks: {blink_count}", (30, 60),
cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 255, 0), 2)
            if drowsy:
                cv2.putText(frame, "DROWSINESS DETECTED!", (30, 90),
cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 0, 255), 3)
    cv2.imshow("Eye Blink Detection", frame)
    if cv2.waitKey(5) \& 0xFF == ord('q'):
        break
cap.release()
cv2.destroyAllWindows()
```

