Gaze-tracking and moving the cursor with gaze direction

- Chethan Hebbar

Face-Detection(First step):

The method:

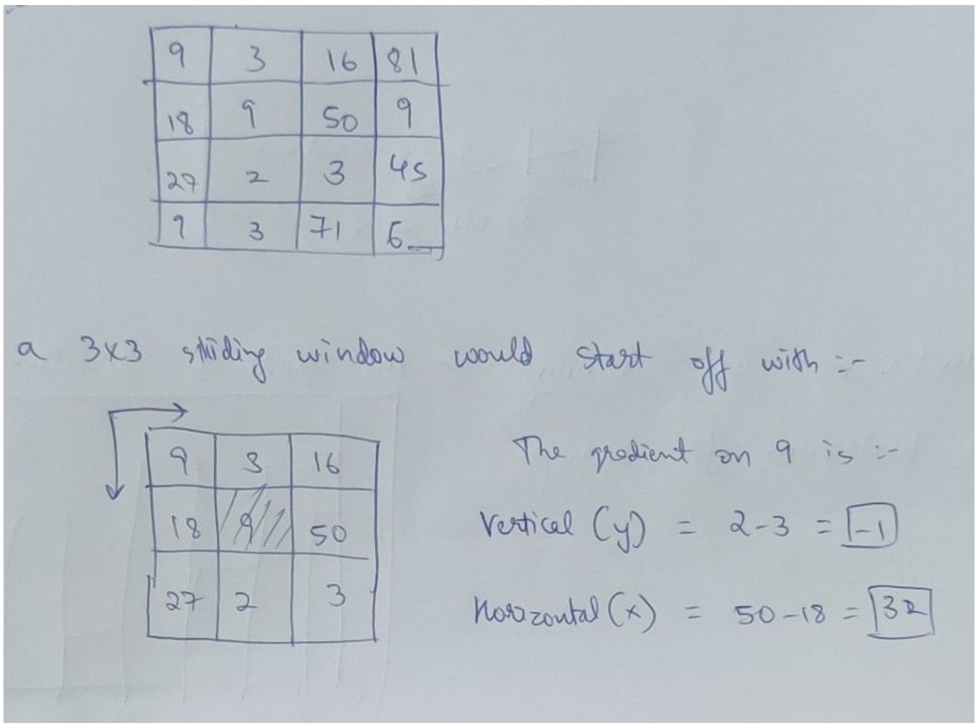
1) We read the image/frame and convert it to grayscale.

2) “get_frontal_face_detector” from dlib library uses HOG and then a linear SVM to classify the images as “with face” or “without face”

3) Feeding the grayscale image to this detector, let us look at it’s working briefly

4) Histogram Oriented Gradients(HOG):

* HOG uses a sliding-window approach on the grayscale image and calculates the gradient values for the pixels. Consider a 4x4 image.



9	3	16	81
18	9	50	9
27	2	3	45
7	3	71	6

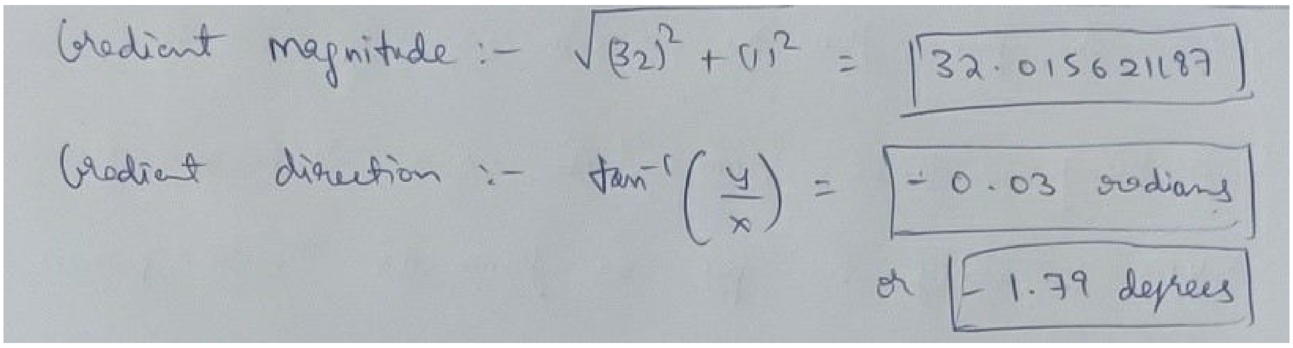
a 3x3 sliding window would start off with :-

9	3	16
18	9	50
27	2	3

The gradient on 9 is :-

Vertical (y) = 2-3 = $\boxed{-1}$

Horizontal (x) = 50-18 = $\boxed{32}$

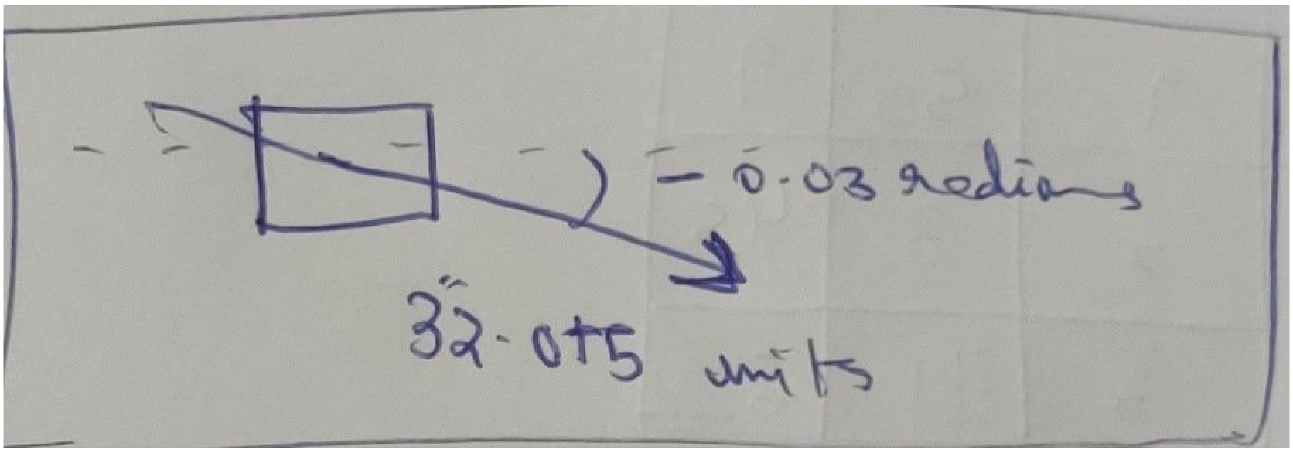


Gradient magnitude :- $\sqrt{(32)^2 + (1)^2} = \boxed{32.015621187}$

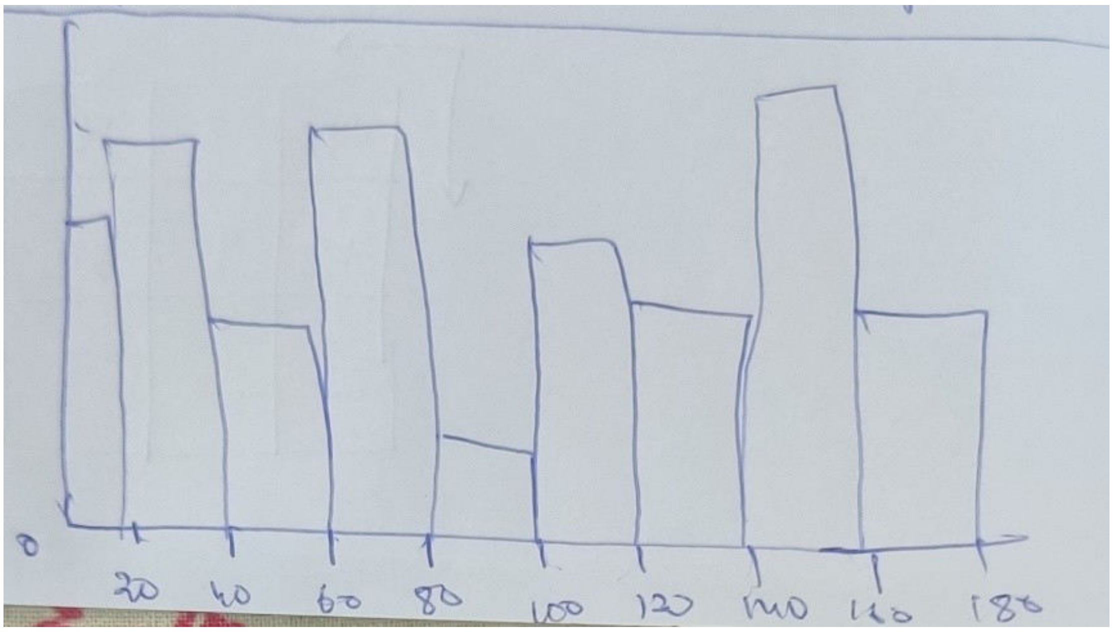
Gradient direction :- $\tan^{-1}\left(\frac{y}{x}\right) = \boxed{-0.03 \text{ radians}}$

or $\boxed{-1.79 \text{ degrees}}$

* Now, after calculating the gradient direction for all the pixels using the sliding window, we have a set of gradient vectors at the pixels, the pixel looks something like this:



* Next, the model performs binning on the gradient directions and we get a histogram, for example:

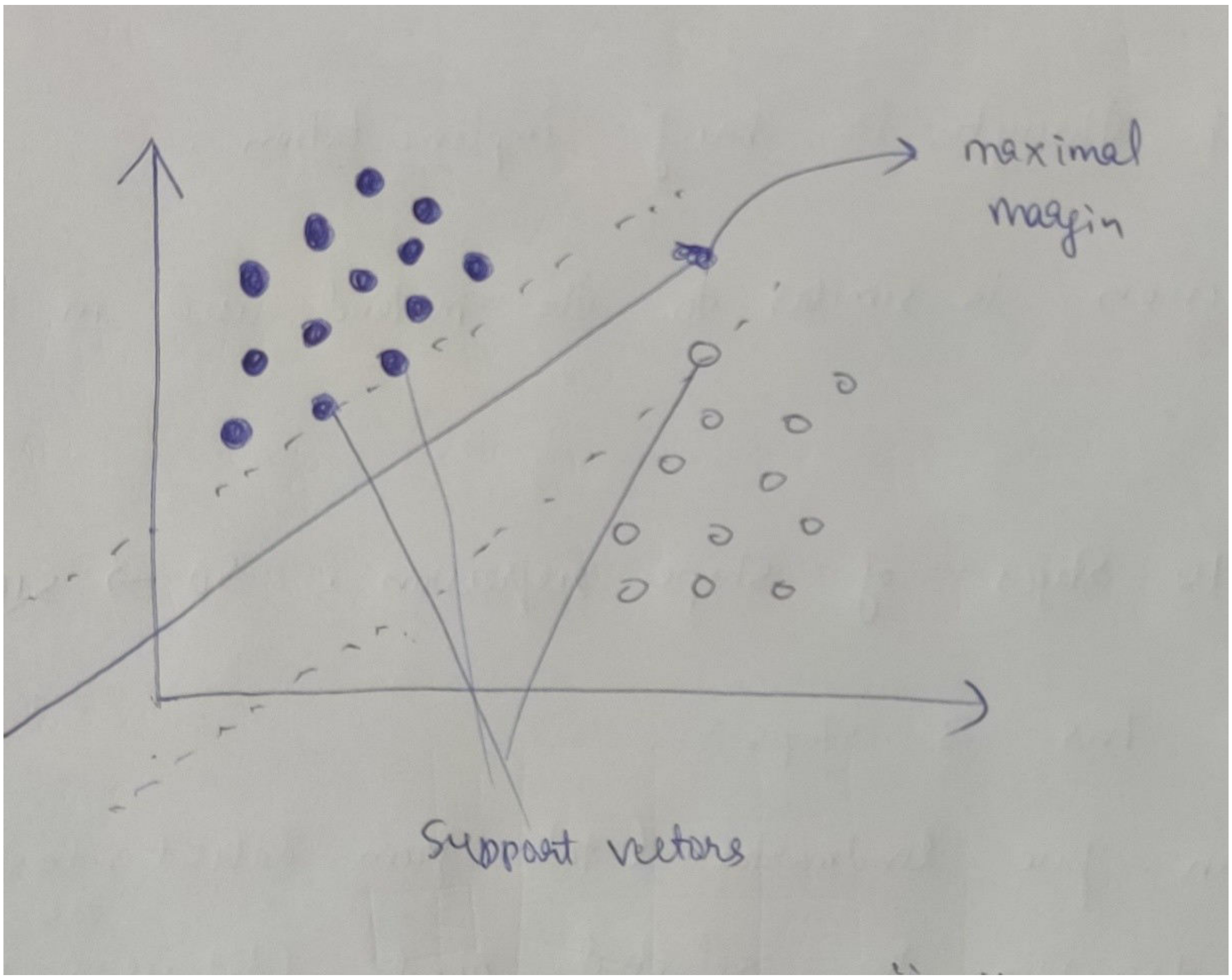


5) After normalisation, the datapoints are sent to the Linear SVM to be classified:

* Given n-datapoints and we have 2 classes, i.e “with face”(1) and “without face”(0).

* The SVM performs “dimension-expansion” => n + 1 dimensions using “kernel functions” [f(xi) where xi is a n-dimensional vector, aka data point]

* A hyperplane is the used to best-fit the points, a hyperplane has the equation [w^T * xi – B = 0] where w is the normal vector to the hyperplane.



* A datapoint on which prediction is to be made, is passed through the kernel function and then classified based on which side of the hyperplane it lies on.

6) Next the model applies a bounding box on the face and returns a tuple of the diagnol of the box, i.e (p1, p2) where p1 p2 are ends of the diagnol.

Landmark Detection(shape prediction):

1) We pass the rectangle coordinates and the gray frame/image to the shape-predictor-68 pretrained model. It has already been trained to produce state-of-the-art results.

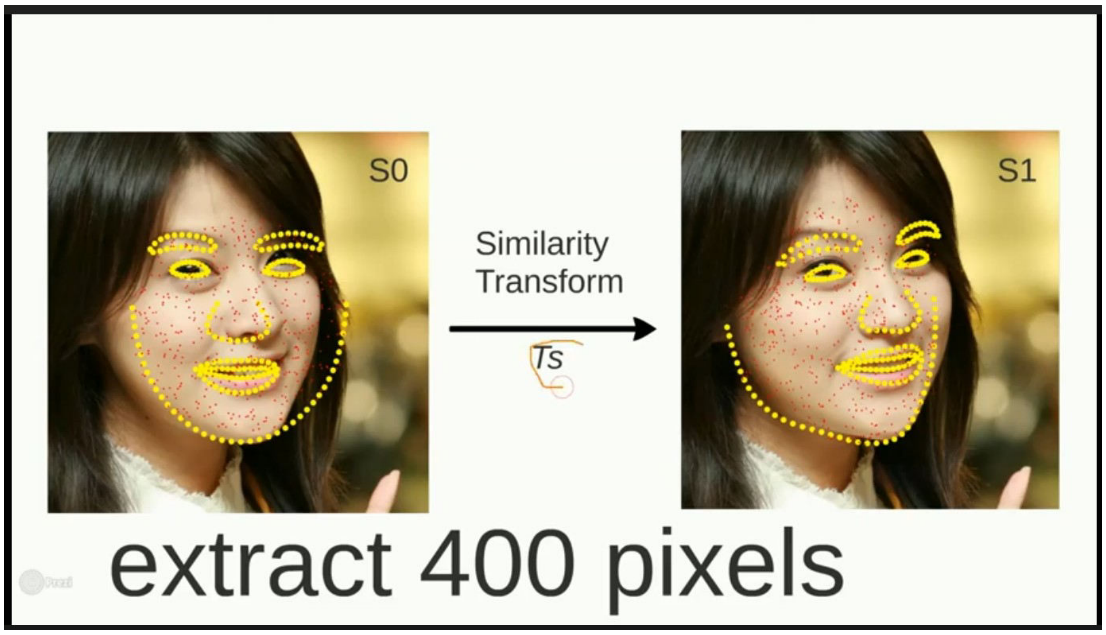
2) Let us go through the implementation briefly.

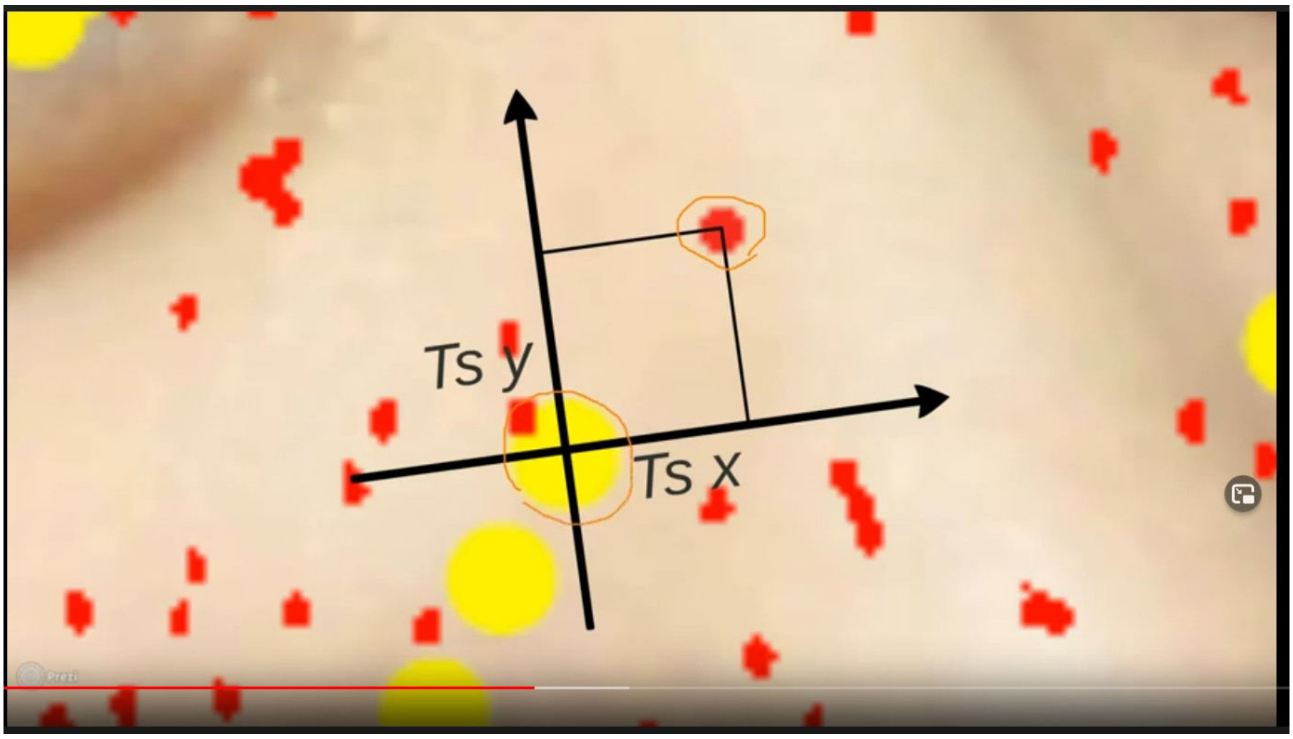
* Shape regression is similar to the method used in the pre-trained model.

* Let us say that the model uses 10 stages of shape-regression, going from S(0) to S(10).

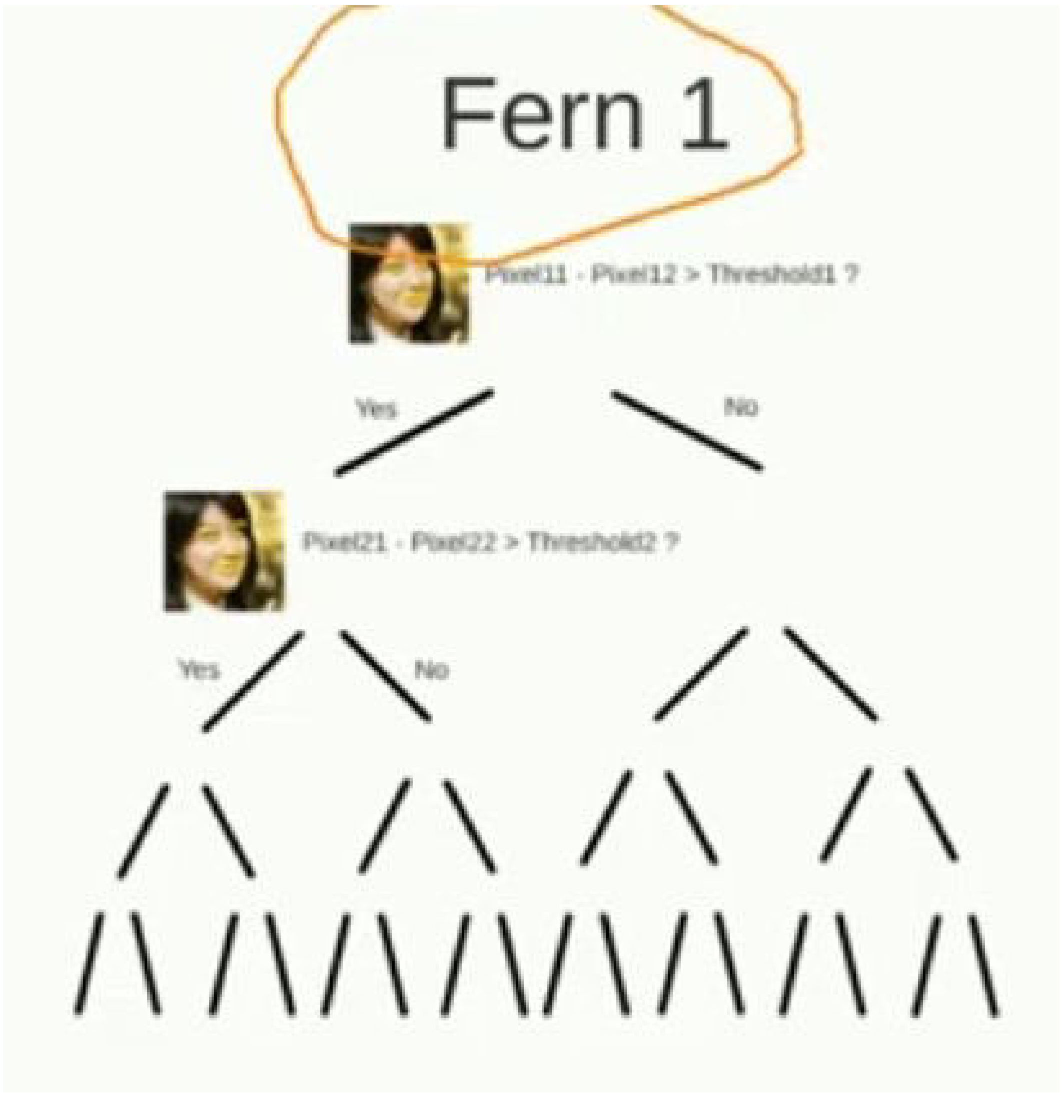
* Every stage uses basically the same startegy:
	> S(0) uses a mean-landmark from the previous datasets while others use the previous landmarks, i.e S(I) uses S(I-1)'s landmarks

	> The landmarks are then 'transformed' along their axes





> Ferns: a fern can be thought of as a constrained binary tree where the same conditional check(done in decision trees) is done. It helps to split data into groups.


Fern 1
Pixel11 - Pixel12 > Threshold1 ?
Yes
No
Pixel21 - Pixel22 > Threshold2 ?
Yes
No

> The DELTA(landmarks) is the loss on which the model is trained, landmarks after applying the transform are slightly changed.

> For this case let us take an example of 400 pixels on which transofrmation is applied and 500 ferns per stage.

* Testing algorithm:

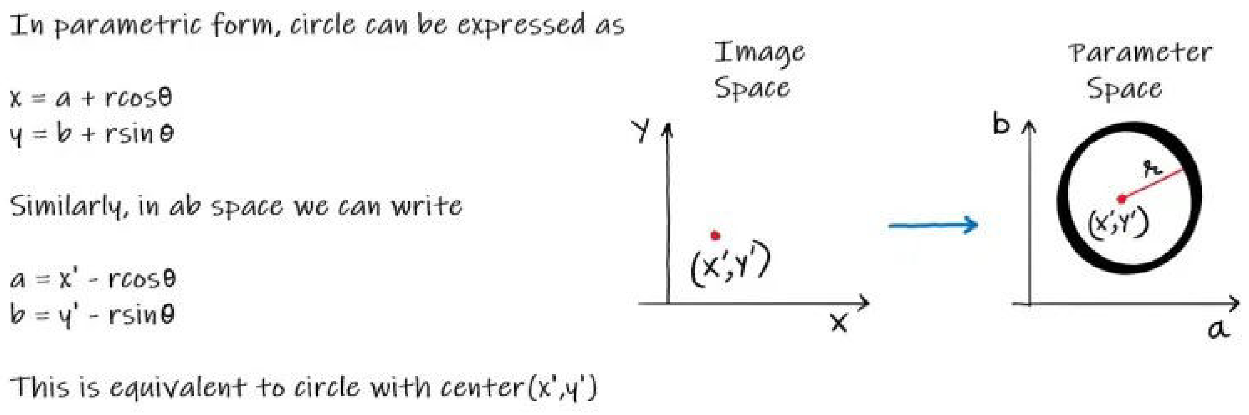
```
> Map the mean facial landmarks to the rectangle sent(i.e the face)

> for stages 1 through 10:
    > Retrieve and map the locations of the 400 sample pixels with
    respect to image using the similarity transform S(mean) to Sc.

    > for j from 1 to 500:
        > Traverse the jth fern by comparing the pixel difference
        feature with the threshold at each level. Reach one of the
```

16 leaves and obtain the DELTA at that leaf and set Sc = Sc + DELTA.

HOUG circle detection:

> We know that a circle can be represented as (x-a)2 + (y-b)2 = r2 where a, b represents the circle center and r is the radius. So, we require 3 parameters (a,b,r) to completely describe the circle.

> We would require a 3D accumulator array. Now, let’s discuss how to fill this accumulator array.
Let’s take a simple case, where the radius r is known to us. In that case, the 3D accumulator array [a,b,r] will become 2D [a,b]. And each point in the (x, y) space will be equivalent to a circle in the (a, b) space as shown below.


In parametric form, circle can be expressed as

$x = a + r\cos\theta$
$y = b + r\sin\theta$

Similarly, in ab space we can write

$a = x' - r\cos\theta$
$b = y' - r\sin\theta$

This is equivalent to circle with center(x',y')

> We will first draw the circles in the ab space corresponding to each edge point. Then we will find the point of intersection (actually the local maxima in the accumulator array) which will correspond to the original circle center.

> The above algorithm is inefficient since it involves dealing with 3D parameter space. Also, we may face a sparsity problem because only a few accumulator cells would be filled. So, to overcome this, OpenCV uses a slightly trickier method known as the Hough gradient method. So, let’s understand how that works.

> As is clear from the name, this method takes into account the gradient information. Earlier for each edge point, we were drawing the corresponding circles in the parameter space and thereby incrementing the accumulator cells. But now instead of drawing the full circle, we only increment the accumulator cells in the gradient direction of each edge pixel.

Implementaion details:

Step 1: Capture the frame and assign var “image” to it.

Step 2: Convert “image” to grayscale, “gray_image”.

Step 3: Pass the “gray_image” to the “face_detector”

Step 4: Retrieve the faces in the frame as “face” in an iteration

Step 5: Pass “face” to the shape-predictor, “get_landmarks”

Step 6: Get the landmarks object returned by “get_landmarks”

Step 7: Using the landmarks for the left eye, use opencv and numpy to form a polygon around the eye and perform bitwise_and on the new “eye” image.

Step 8: Use the “eye” image and perform a median or gaussian blur.

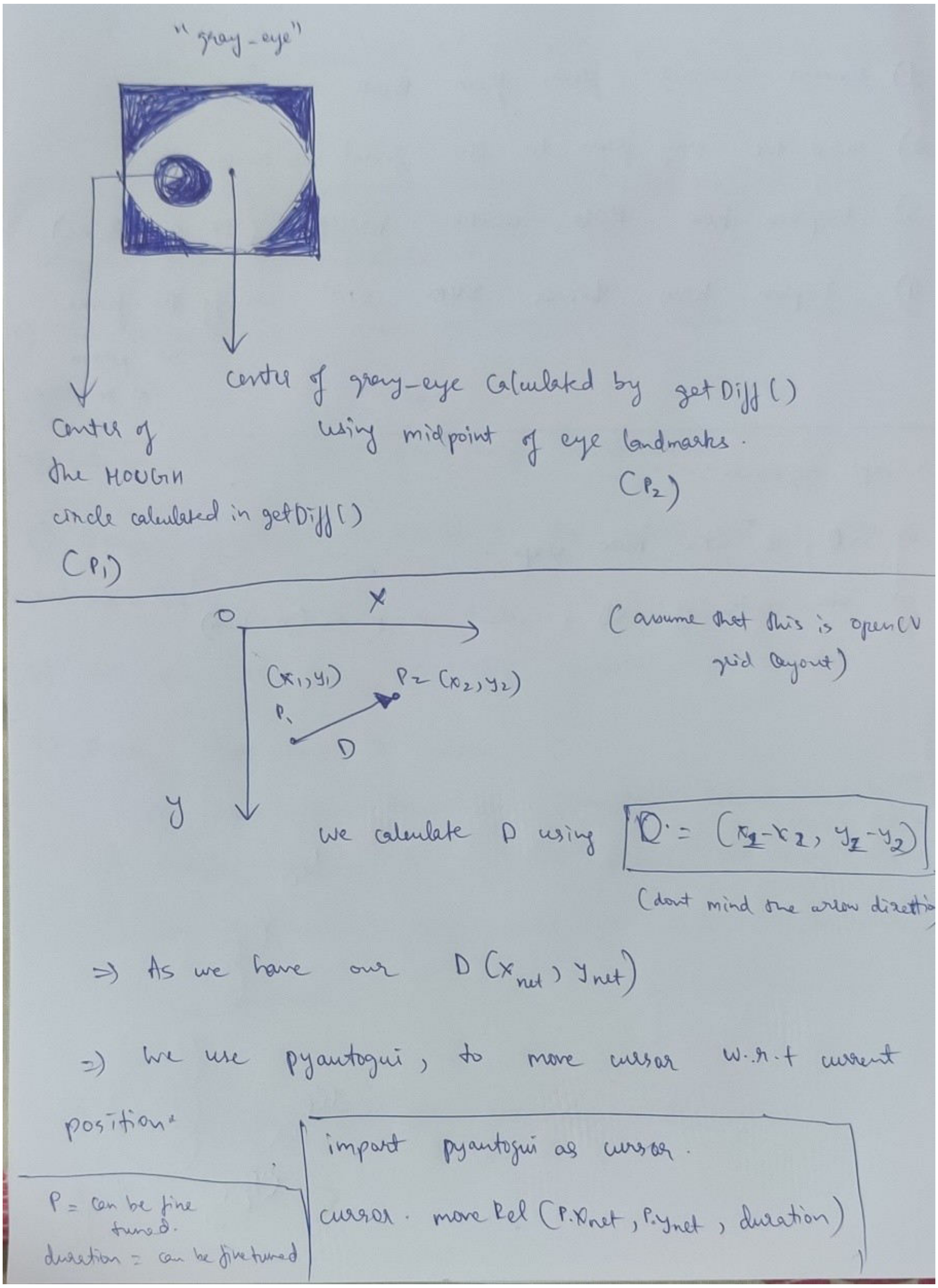
Step 9: Send the image from Step 8 “gray_eye” to a HOUGH circle detector that uses HOUGH gradient method, the paramteres are tuned accordingly.

Step 10: Retrieve the center of the detected circle, two avoid detecting small circles(other than the iris) we set “min_radius” of the detector appropriately.

Step 11: Return the center of the circle and the center of the “gray_eye” image back to the control.

Step 12: Perform steps 7 through 11 for the right eye.

Step 13: Now that we have the centers of irises and the “gray_eye” for both eyes. We pass it to a function “moveCursor” which works like this:



center of gray-eye calculated by getDiff() using midpoint of eye landmarks. (P_2)

Center of the HOUGH circle calculated in getDiff() (P_1)

(assume that this is openCV grid layout)

we calculate D using $\boxed{D := (x_2 - x_2, y_2 - y_2)}$

(dont mind the arrow directio

⇒ As we have our $D (x_{net}, y_{net})$

⇒ We use pyautogui, to move cursor w.r.t current position.

P = can be fine tuned.
duration = can be fine tuned

```
import pyautogui as cursor.

cursor.moveRel(P.xnet, P.ynet, duration)
```

END

Next steps and another good repo used:

1) Want to use CNNs to classify the eye-image as looking center, left, right, top or bottom.

2) One challenge I had with this is creating the dataset(I could not find dataset online, although I did'nt look very closely)

3) Training the CNN can be done given time.

4) I used a well documented repo to get better results, the repo is linked in the README on this repository.

5) It turned out that the HOUGH-Circles method gave better results, but have not implemented CNN.

6) The tool is quite slow on my machine, running it remotely or on a GPU might increase performance significantly.

References:

Paper on HOG ::http://lear.inrialpes.fr/people/triggs/pubs/Dalal-cvpr05.pdf

Paper on Ensemble of regression trees:
https://www.csc.kth.se/~vahidk/papers/KazemiCVPR14.pdf