Gaze Tracker

Riccardo Bertoglio

May 7, 2019

Contents

1 Introduction

Gaze Tracker is a Matlab program that localizes the eye centers and plots the estimated gaze on the screen using a Feature-Based approach. The program is thought to work in real-time with a single low cost camera, such those you can find on PCs, tablets and smartphones. Gaze Tracker is the Matlab implementation of the paper by Skodras et al. "On visual gaze tracking based on a single low cost camera" [1]. The proposed program is a free implementation of the paper with some differences and extensions. The main difference is that my program does not take into account the eyelids position as an additional feature to increase robustness of the eye center localization. This is leaved as future work. Also, the program here proposed differs in the choice of some parameters that I modified based on my experiments. Finally, I freely implemented the parts of the paper where not enough details were given to thoroughly understand the process (like in the tracking of the eye centers and anchor points).
As for this document, in Section 3, I give a general and short overview of the code with the help of a flowchart; in Sections 4, 5, 6, 7, I explain the theoretical concepts and the rationale behind the main parts of the code; in Section 8, I explain how to use the program and I give some suggestions on the setup; in Section 9, I discuss on the results obtained and on the strong and weak points of this approach; in Section 10, I propose some future work.

2 State of the art

Several approaches and techniques have been proposed in the literature. For a complete overview of the state of the art please refer to the paper "Low Cost Eye Tracking: The Current Panorama" [2].

3 Code flowchart

In this section I describe the code from a general perspective with the help of a flowchart (Figure 1). We can distinguish five main parts of the code, that is, the main loop, the eye centers detection, the anchor points tracking, the interpolation model and the calibration.

Main loop The outer level of the program is constituted by a while loop that processes all the frames arriving from the webcam. The while loop is interrupted when the user closes the calibration or the test windows or when the calibration is corrupted if `checkCorruptedCalibration` flag is set to `true`. For each frame, the Viola-Jones objects detection algorithm is used to detect a face. If no face is found then I drop the frame and jump to the next iteration of the cycle.

Eye centers detection If a face has been detected then the algorithm localizes the eye centers as described in detail in Section 4 and a spacial continuity

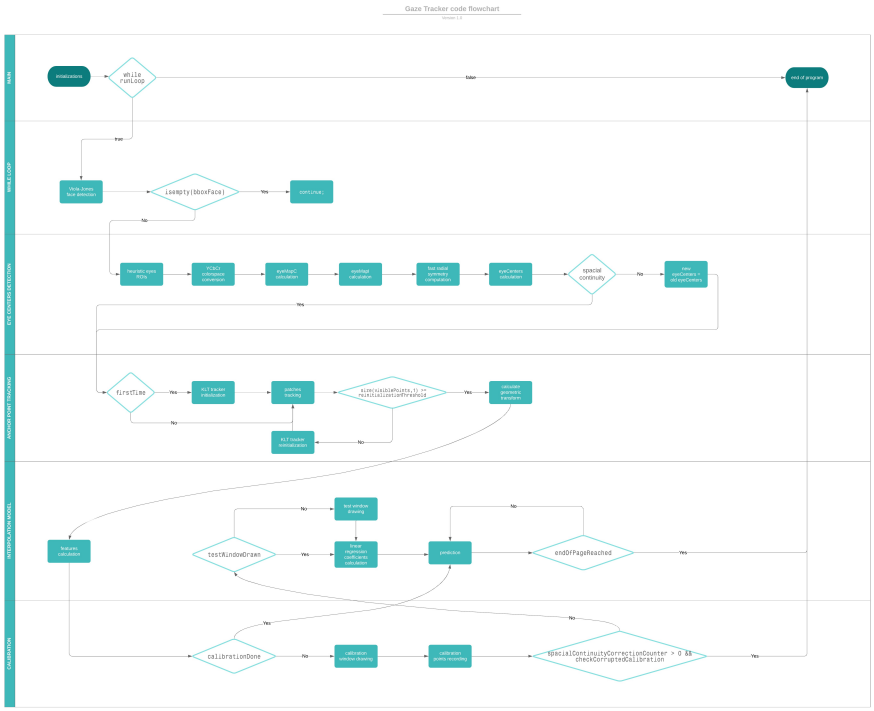


Figure 1: Gaze tracker code flowchart.

check is done. The spacial continuity check compares the old eye centers positions with the new ones and checks if the Euclidean distance between them is below a given threshold `spacialContinuityThrsh`. If the distance is above or equal to the threshold, the new position is discarded and the old one is kept. This check and the relative possible correction is performed separately for each eye.

Anchor points tracking After eye centers localization the anchor points, necessary to build the features, should be found. In case we are processing the first frame, the variable `firstTime` is `true` and so we initialize the Kanade-Lucas-Tomasi (KLT) tracking algorithm to track a rectangular patch as described in detail in Section 5. The center of the rectangle is an anchor point and we build two of these patches near each eye so we have two trackers working. As time goes on less and less features will be found, so, at some point, the trackers will need to be reinitialized. This is done when the number of visible points is below a given threshold `reinitializationThreshold`. This check and the relative possible reinitialization is performed separately for each eye. After the check, we calculate the geometric transform from the old patch position to

the new one to compute the new anchor points positions.

Interpolation model Once we have calculated the positions of the anchor points we can compute the features as horizontal and vertical distances between the anchor points and the eye centers as explained in detail in Section 6. Once calibration has been done (`calibrationDone == true`) I draw a test window with a phrase to read and the plot of the estimated gaze. To predict the gaze I use a linear regression model. Once the user has reached the end of the phrase the plotted gaze should be contained in a red area that will turn green if at least three points of the estimated gaze will fall on it (`endOfPageReached == true`). When the end of the page is reached the test is completed and the program stops the execution.

Calibration The calibration procedure is done only once at the beginning of the execution. The calibration is needed to calculate the coefficients of the linear regression model. The calibration procedure will show a window with some points changing color. When a point turns green the user should look at it and maintain open and fix eyes until the final 'bip' is heard and the point turns red. The calibration pattern consists in 9 points placed in a cross arrangement.

4 Face detection and eye centers localization

Face detection Face detection is performed by means of the Viola-Jones algorithm. The algorithm is available in Matlab as part of the Computer Vision Toolbox. The algorithm takes as input the gray-scale image where to find the face and returns a matrix of bboxes containing all the faces found. The program selects the first one and it is not expected to handle multiple faces in the same frame. If no face is found the program just drops the frame and continue to the next iteration. Sometimes the algorithm fails returning a detected object that is not a face. In this case the program should be able to handle this exception with the spacial continuity check. It is reasonable to suppose that the wrong object is farther away than the spacial continuity threshold that, for sake of explanation, it consists of a small circumference ($\sim$25 pixels radius) around the eye center. If the new detected eye center doesn't fall in this circumference, the new position is discarded and the old one is kept. In Figure 2 you can see the detected face in the red box.

Eye centers localization The eye centers are localized as the result of some steps (Figure 3):

Heuristic eye ROIs The localization of the eyes is performed in a limited Region Of Interest (ROI) defined with respect to the face box found in the previous step. For both cropped ROI images we perform the following steps. In Figure 2 you can see the eye ROIs in cyan boxes.

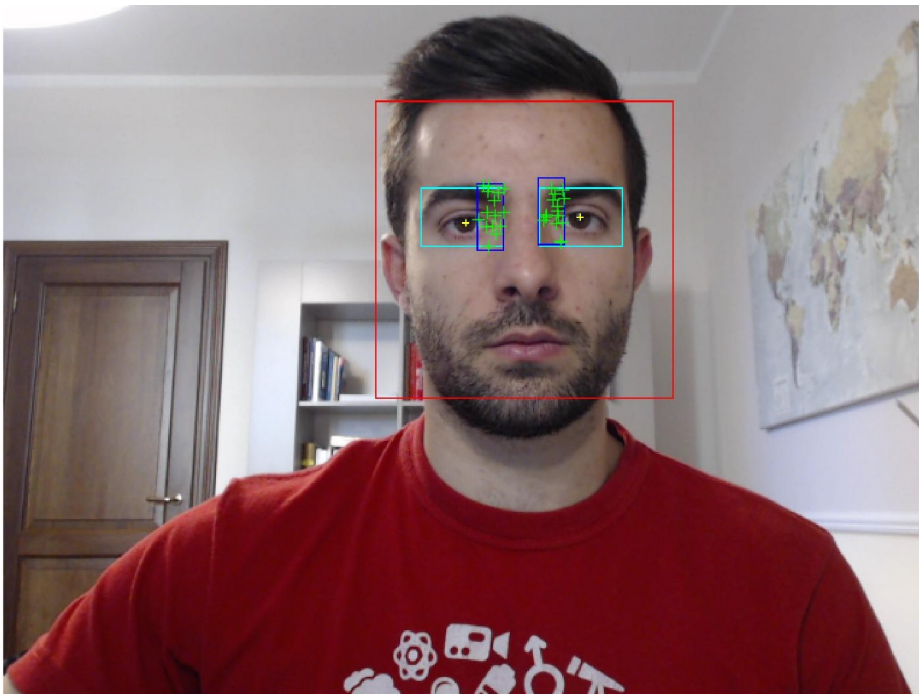


Figure 2: Detected face (red box), eye ROIs (cyan boxes), eye centers (yellow crosses) and tracked patches (blue boxes) with their tracked points (green crosses).

YCbCr color space conversion The cropped images are first converted from RGB to YCbCr color space. This latter color space is ideal to distinguish the skin area from the eye area. The eye area has a blue dominant resulting in higher values in the Cb channel. The skin area has a red dominant resulting in higher values in the Cr channel. Also, the separation of the luminance information makes the skin modeling more resilient to different lighting conditions and uneven illumination. Figure 4 shows a converted image of the left eye ROI.

eyeMapC calculation The eyeMapC is the result of the combination of Cb and Cr channels to highlight the eye area. The eyeMapC is defined as

$$eyeMapC = \frac{1}{3}[Cb^2 + \overline{Cr}^2 + Cb/Cr]$$

where Cb, Cr are normalized to the interval [0,1] and $\overline{Cr}$ means $(1 - Cr)$. The Cb, Cr division could produce inf or NaN pixel values. inf is saturated to the max numerical value and NaN is replaced by 0. Large values on the eye map are observed at the positions of the eye regions and eyebrows where the color difference from the skin pixels is maximized.

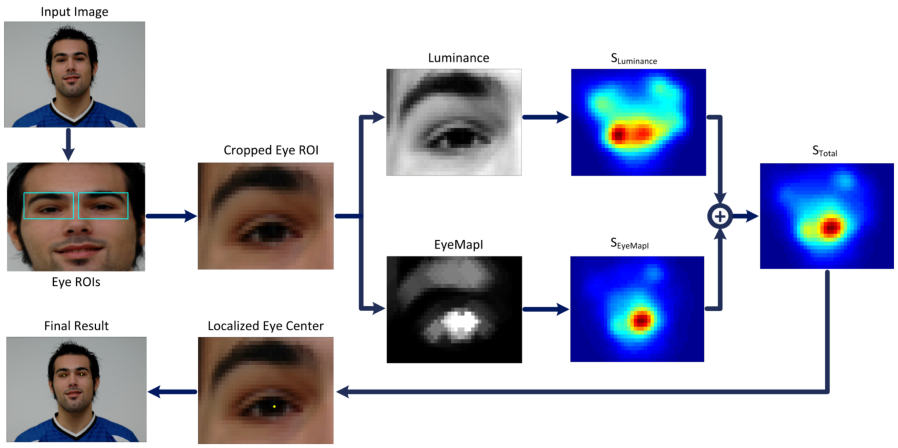


Figure 3: Overview of the proposed system. For clarity purposes, the images showing radial symmetry transform results are pseudo-colored. Image taken from [5]

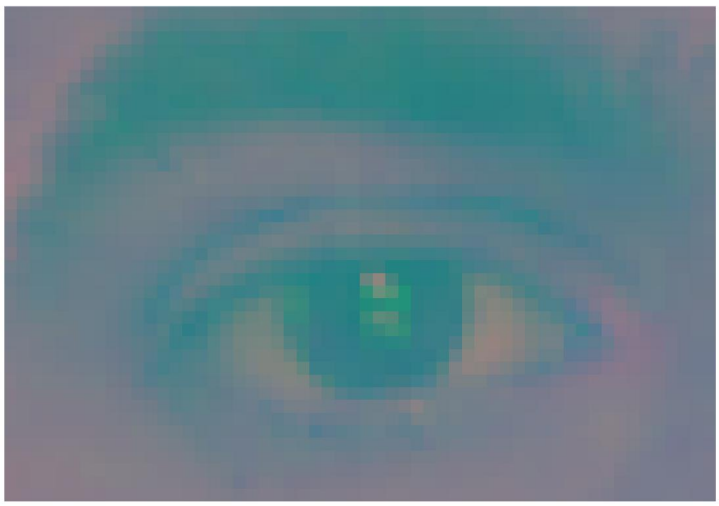


Figure 4: YCbCr converted image of the let eye ROI.

eyeMapI calculation The irises present significantly lower brightness values than the sclera and the skin areas. So, we want to fuse the information of

the $eyeMapC$ with the luminance channel. Therefore, we calculate a new eye map $eyeMapI$, that is, the division of $eyeMapC$ by Y, the luminance channel. The process is graphically represented in Figure 5. The $eyeMapC$ has high values in the iris area while Y has low values in the same area, resulting in a new eye map that further enhances the iris area. Moreover, the use of morphological operations such as dilation and erosion further accentuates the irises darker appearance in the Y component and the brighter appearance in the $eyeMapC$ component. We perform these operations with two flat circular structuring elements called $B1$ and $B2$, for $eyeMapC$ and Y respectively. The radius of these circular elements are defined with respect to the iris radius.

$$eyeMapI = \frac{eyeMapC \oplus B1}{(Y \ominus B2) + \delta}$$

where $\oplus$ and $\ominus$ denote gray-scale dilation and erosion, respectively. δ is used to solve numerical problems deriving from a zero division. A small static number would suffice, yet, experimental tests exhibited improved results when a dynamic, data-driven value of δ is used:

$$\delta = mean(Y \ominus B2)$$

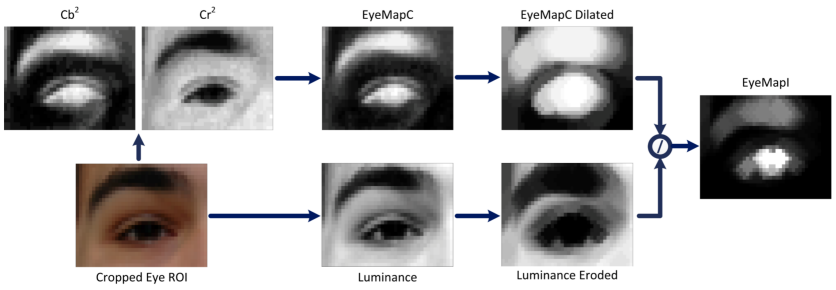


Figure 5: eyeMapI construction. Image taken from [5]

Fast radial symmetry transform The fast radial symmetry transform (FRST) [3] is a transform that utilizes local radial symmetry to highlight points of interest within a scene. Its low computational complexity and fast runtimes makes this method well-suited for real-time vision applications. The transform relies on a gradient-based interest operator that works by considering the contribution of each pixel to the symmetry of pixels around it. The implementation I'm using is a personal modification of the Loy and Zelinski's approach [3] implemented by Kovesi `https://www.peterkovesi.com/matlabfns/`. First, the gradient of a gray-scale image is calculated computing derivatives in x and y via Farid and Simoncelli's 5 tap derivative filters. The results are significantly more accurate than Matlab's gradient function on edges that are at angles other than

vertical or horizontal. This in turn improves gradient orientation estimation enormously. Then, we define a set of discrete radii N. For each n in N we do the following. From each pixel p, a pair of positively and negatively affected pixels $\mathbf{p}_{+af}$, $\mathbf{p}_{-af}$ is determined. The positive affected pixel is defined as the pixel at distance n away from $\mathbf{p}$ in the direction of the gradient vector $\mathbf{g}(\mathbf{p})$. The negative affected pixel is the one in the opposite direction. Figure 6 shows a graphical representation. The gradient increases positively from dark to bright areas. The coordinates of the positively-affected pixel are given by

$$\mathbf{p}_{+af} = \mathbf{p} + round\left(\frac{\mathbf{g}(\mathbf{p})}{\left\|\mathbf{g}(\mathbf{p})\right\|}n\right)$$

while those of the negatively-affected pixel are

$$\mathbf{p}_{-af} = \mathbf{p} - round\left(\frac{\mathbf{g}(\mathbf{p})}{\left\|\mathbf{g}(\mathbf{p})\right\|}n\right)$$

where "round" rounds each vector element to the nearest integer. For each n in N we compute an orientation projection image O_n and a magnitude projection image M_n. These two images are initially zero. In the case of orientation projection images we increment positively affected pixels by 1 while negatively-affected ones are subtracted by 1. In the case of magnitude projection images we sum/substract the quantity $\left\|g(p)\right\|$ respectively.

$$O_n(\mathbf{p}_{+af}(\mathbf{p})) = O_n(\mathbf{p}_{+af}(\mathbf{p})) + 1$$

$$O_n(\mathbf{p}_{-af}(\mathbf{p})) = O_n(\mathbf{p}_{-af}(\mathbf{p})) - 1$$

$$M_n(\mathbf{p}_{+af}(\mathbf{p})) = M_n(\mathbf{p}_{+af}(\mathbf{p})) + \left\|\mathbf{g}(\mathbf{p})\right\|$$

$$M_n(\mathbf{p}_{-af}(\mathbf{p})) = M_n(\mathbf{p}_{-af}(\mathbf{p})) - \left\|\mathbf{g}(\mathbf{p})\right\|$$

Then, the radial symmetry contribution at radius n is defined as the convolution

$$S_n = F_n * A_n$$

where F_n is derived merging the orientation and magnitude projection images as in the following

$$F_n(\mathbf{p}) = \frac{M_n(\mathbf{p})}{k_n}\left(\frac{|\widetilde{O}_n(\mathbf{p})|}{k_n}\right)^{\alpha}$$

with

$$\widetilde{O}_n(\mathbf{p}) = \begin{cases} O_n(\mathbf{p}) & \text{if } O_n(\mathbf{p}) < k_n \\ k_n & \text{otherwise} \end{cases}$$

where A_n is a two-dimensional Gaussian, α is the radial strictness parameter, and k_n is a scaling factor that normalizes M_n and O_n across different radii. The full transform is defined as the average of the symmetry contributions over all radii considered,

$$S = \frac{1}{\|n\|}\sum_{n \in N} S_n$$

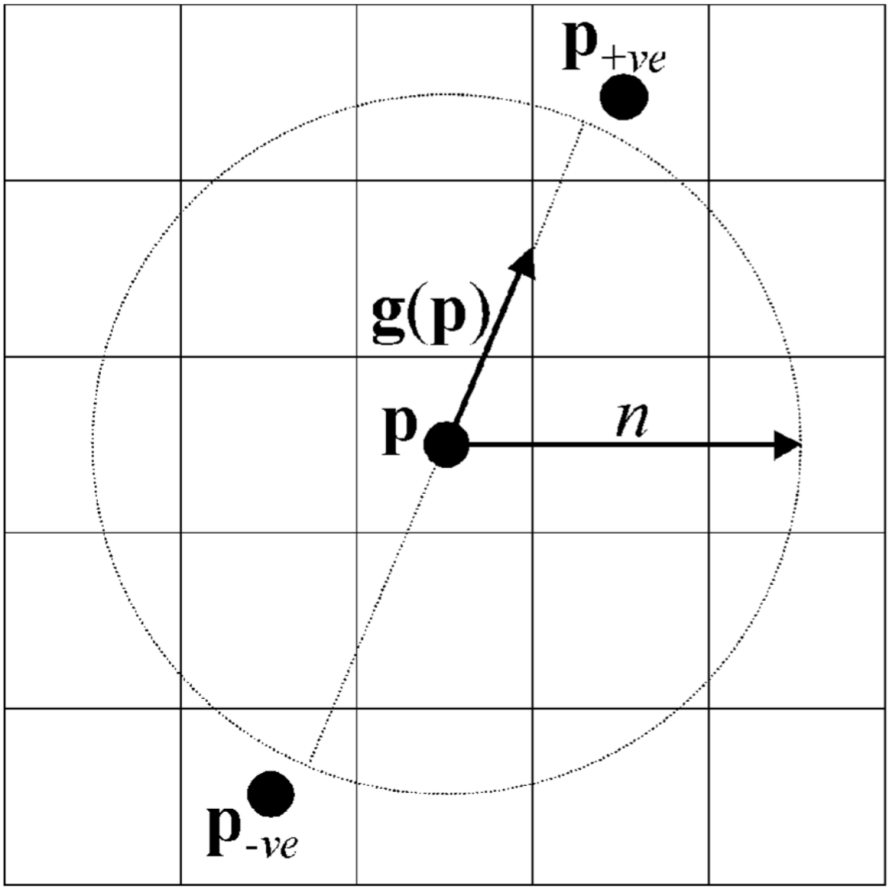


Figure 6: Positively and negatively affected pixels from **p**. Image taken from [3]

The transform can be tuned to look only for dark or bright regions of symmetry. To look exclusively for dark regions, only the negatively-affected pixels need be considered when determining M_n and O_n. Likewise, to detect bright symmetry only positive effected pixels need be considered.
The transform has some parameters to tune:

- set of discrete radii N = $[n_{min}, n_{max}]$
- the Gaussian kernels A_n. The purpose of the Gaussian kernel A_n is to spread the influence of the positively- and negatively-affected pixels as a function of the radius n
- the radial stricteness parameter α. The parameter α determines how strictly radial the radial symmetry must be for the transform to return a high interest value

- small gradients threshold β. Gradient elements with small magnitudes have less reliable orientations, are more easily corrupted by noise, and tend to correspond to features that are not immediately apparent to the human eye. We ignore small gradients by the gradient threshold parameter β

Eye centers calculation The fast radial symmetry transform algorithm is applied to the eroded luminance image `YEroded` and to the `eyeMapI`. Then, the two transforms $S_{YEroded}$, $S_{eyeMapI}$ are summed together and the position of the maximum value pixel is the position of the eye center

$$(x_c, y_c) = argmax(S_{YEroded} + S_{eyeMapI})$$

Figure 2 shows the detected eye centers with yellow crosses.

Spacial continuity check Sometimes the eye center localization procedure may fail due to a wrong face detection from Viola-Jones algorithm or because of some external conditions like eyes blinking or light changes. These will result in a wrong position of one or both eye centers and thus it's essential to track the eye centers positions over time to ensure spacial continuity. That is, if between two successive frames the eye centers positions have varied more than a radial distance threshold, it's likely that this is due to a wrong localization. If a such event occurs, the new position(s) of one/both eye center(s) is/are set as the previous one(s). The threshold can be modified and adapted to current image resolution with the `spacialContinuityThrsh` variable.

5 Anchor points construction and tracking

Anchor points construction The most common approaches consider inner and outer eye points as anchor points. Instead of tracking isolated points, here is used the more efficient alternative of tracking a rectangular image patch and consider as anchor points its center coordinates. We build one patch for each eye. These two patches are located near to the inner sides of the eyes, the ones between the eyes corners and the nose. A patch contains the inner eye corner and the eyebrow edge, therefore comprising a highly textured area containing edges which are easy to track robustly. The dimensions of the patches depend on the interocular distance. The interocular distance is simply the distance between the two detected eye centers. In Figure 2 you can see the tracked patches in blue.

Anchor points tracking Once the patches have been initialized they need to be tracked frame by frame. In order to do this I use the Matlab implementation of the Kanade-Lucas-Tomasi (KLT) tracking algorithm [4]. I use two trackers, one for each eye. Each tracker is initialized using the entire frame image and as ROI the patch previously found. As the point tracker algorithm progresses

over time, points can be lost due to lighting variation, out of plane rotation, or articulated motion. To solve this problem I reinitialize a tracker when the visible point are less than the fixed threshold `reinitializationThreshold`. Since we are tracking a flat patch on a plane we need at least 2 visible points. In Figure 7 you can see the tracked patches at two different degrees of angle.

(a) Face is straight. (b) Face is rotated.

Figure 7: Tracked patches at two degrees of angle. Tracked visible points are the white crosses and tracked centers are the green crosses.

6 Prediction

The prediction model is the piece of code that map image data to screen positions, that is, the gaze direction. Here a linear regression model is used. Second-order polynomial equations are commonly used for 2D mapping and are generally useful to correct curved distortions and to smooth scaling along the screen. However, non-linear terms may introduce big errors especially when approaching to screen borders. As a result, errors during calibration can lead to much larger errors on screen coordinates estimations. Moreover, the more the coefficients to learn, the more the training examples should be. In order to have a fast calibration procedure and for the reasons above explained, a linear regression approach is used. Two mapping functions, one for each direction (along x and y axes), are learned. Given the assumption of independence of gaze estimation in the two axes, two separate feature vectors are formed as horizontal and vertical distances between moving and anchor points. Features for both axes are calculated as the following:

- Horizontal direction (three features)
  - distance between the x-coordinates of each of the eye centers and the ipsilateral anchor point
  - distance of one of the eye centers and the anchor point at the opposite side

- Vertical direction (two features)
  - distance between the y-coordinates of each of the eye centers and the ipsilateral anchor point

An example of gaze prediction is given in Figure 8.

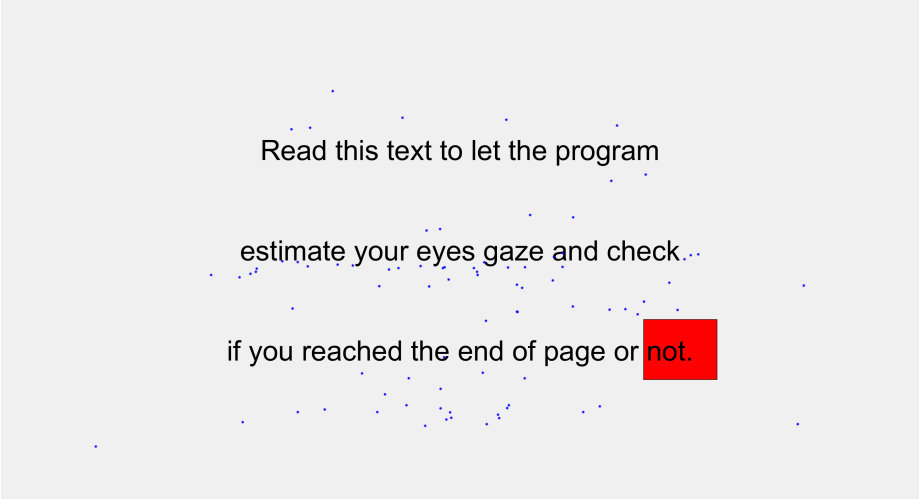


Figure 8: Plotted predicted gaze (blue asterisks).

7 Calibration

The calibration procedure is the phase where we learn the model coefficients. It consists in showing some points to the user, lighting them one by one and registering the feature vectors in that known positions. After calibration I have correspondences between known points on the screen and the relative features, allowing me to train the linear regression model. From experiments performed by the authors of the paper [1] the most effective pattern is a cross shape with 10 calibration points (one counted twice). The proposed cross arrangement of points intends to separate the eye movements in the horizontal and vertical directions, so that the mapping coefficients are calculated independently. In Figure 9 you can see the calibration pattern.

8 Usage instructions

The program has been thought to be intuitive and simple to tune. Here are some guidelines and suggestions:

- the program is compatible with Matlab R2018b or newer versions, backward compatibility it's not guaranteed

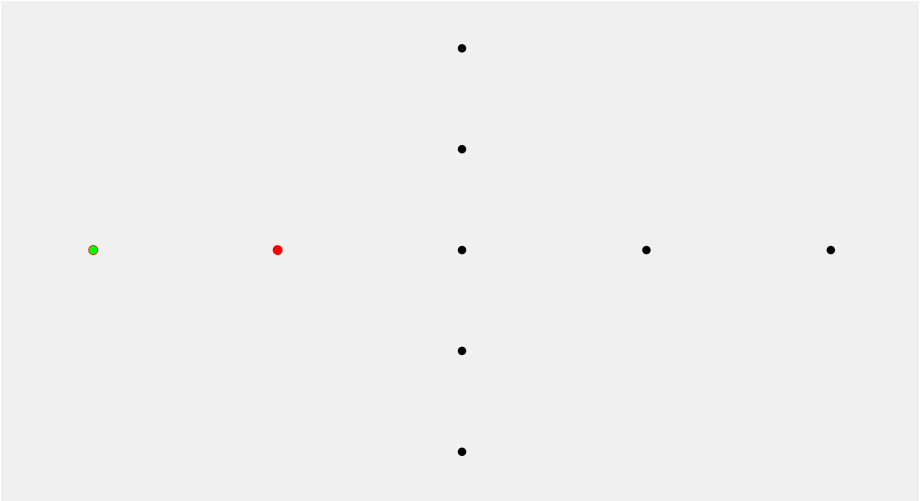


Figure 9: Calibration pattern.

- open Matlab double clicking on the `startup.m` as it adds all the necessary files to the path
- before running the program take a look at the beginning of the code to adjust some parameters:
  - you can enable debug output with the variables `debugImage`, `debugAllFrames` and `debugInfo`
  - set the correct name of your webcam in the webcam constructor `cam = webcam('your_webcam_name')`
  - you can modify the camera resolution with `cam.Resolution`
  - you can enable the flag `checkCorruptedCalibration` that will stop the program if a spacial continuity correction will happen during calibration phase. You can set it to `false` for a general use
  - with the `reinitializationThreshold` variable you can set the minimum (≥ 2) number of valid points below which the tracker needs to be reintialized
  - with the `spacialContinuityThrsh` you can set the maximum accepted Euclidean pixel distance between eye center positions in two successive frames related to a single eye
  - you can personalize the timing and tone of the sounds used during the calibration phase
- the camera should be placed at the same level as the eyes of the user

- it's not important to have a high resolution but instead the camera should represent with high fidelity the colors. If it's not the case calibrate it with a ColorChecker
- make sure there is a good illumination, that the light is uniformly spread and there are no light variations. Avoid for example to have sunlight coming from a window illuminating just part of the face
- maintain your face as steady as possible during the entire execution of the program and try to avoid blinking especially during the calibration phase
- make sure that the detection of the eyes centers is correct in the first frame interacting with the dialog box
- during calibration phase look at the points on the screen (see Figure 9)
  - black: point is inactive
  - red: the point is active. Fix it, you'll hear three bips giving you the time to look at the right point. The frame is kept at the fourth bip.
  - green: the point has been already used
- a test window will appear after calibration, once you have placed three points in the red rectangle the test ends (see Figure 8)

9 Discussion

The implemented proposed approach has both strong and weak points. The use of color information to locate the eye centers is far more reliable than geometric methods in challenging cases. Color information has proven to be very helpful in cases where the iris is merely visible and the circular shape reaches its limit, such as eye closure in a big degree or extreme positions of the eye in the eye socket. Another advantage is that this approach doesn't require high resolution images and or additional hardware equipment. Actually, the less the resolution is, the faster the program is. Of course, the resolution should be high enough to allow the tracker to work properly. The color distribution should constitute a feature resilient to illumination conditions, pose and most appearance changes. Actually, in my experience, I noted that with low color fidelity cameras, especially in the case of uncontrolled illumination conditions, the program is tricked easily. This can be seen in Figure 10 where I was traveling on a train and taking pictures with the low quality integrated camera of my PC. In this situation the program has been tricked by the strong lateral light coming from the window. The right eye center is completely wrong because the detection has been severely affected by the strong lateral light. The left eye center it is not centered in the retina but still it falls in the eye socket. In this case the program has been tricked by shadows and reflections caused by an uneven illumination. In Figure 11you can see that the left most part of the left eye is whiter than the right part.

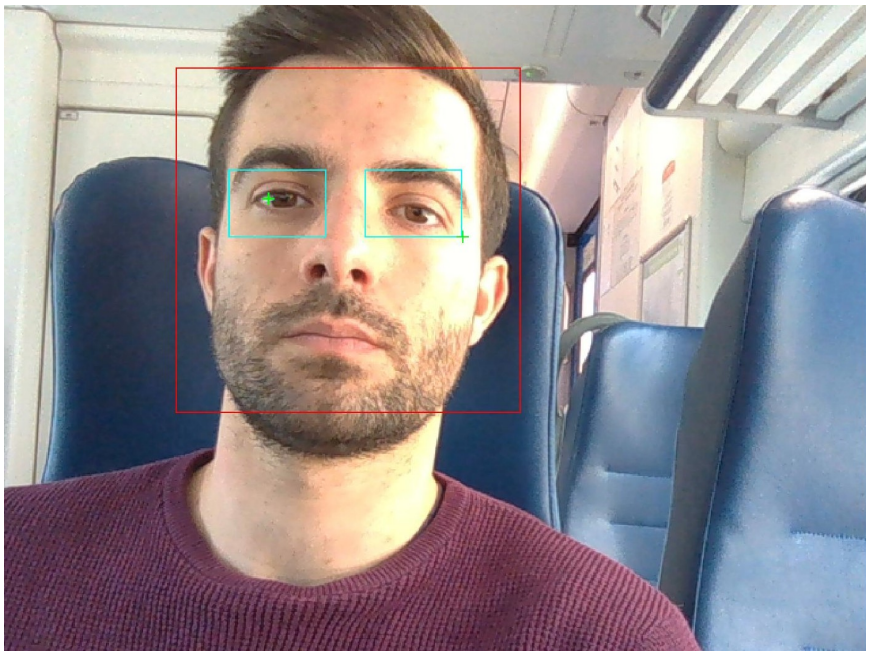
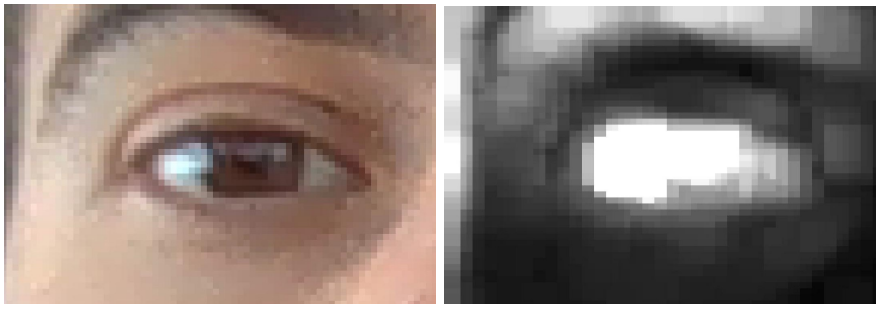


Figure 10: Wrong detection of eye centers (in green).



(a) RGB ROI of a wrong eye detection. (b) eyeMapI of a wrong detection.

Figure 11: On the left you can see the left most part of the eye that is wither than the other. In that region we have a blue dominance and a red scarcity, thus resulting in the eyeMapI on the right. Notice the strong presence of white pixels in the eyeMapI in correspondence of the same area in the RGB image.

An additional feature of the proposed algorithm is its low computational complexity. The most computationally demanding part is the radial symmetry transform, which however presents relatively low complexity, depending linearly on the size of the image i.e. O(KN), where K is the number of pixels and N is the radii (range) of the local neighborhood.

10 Future work

The first thing to do is to complete the implementation of the proposed approach [1] adding the position of the upper eyelid as an additional feature for the vertical direction. This should increase considerably the accuracy of the gaze estimation in the vertical direction. Another important improvement is to the reinitialization process of the tracking algorithm. At the moment the tracker is reinitialized using a rectangular ROI that only approximates the current bounding box being tracked. The rectangular ROI used during the reinitialization have the sides parallel to the x and y axes but the current tracked bounding box could have a degree of rotation with respect to the x-y axes. Another improvement could be the tracking of the face instead of the detection in every frame. This could low the computational load and make the eye centers localization more robust since the dimension of the bounding box will be always the same in each frame. Moreover, the face tracking solves the problem of wrong object detection.

References

[1] Evangelos Skodras, Vasileios G.Kanas, Nikolaos Fakotakis; *On visual gaze tracking based on a single low cost camera*; Signal Processing: Image Communication, Volume 36, Pages 29-42, August 2015.

[2] Onur Ferhat, Fernando Vilariño; *Low Cost Eye Tracking: The Current Panorama*; Computational Intelligence and Neuroscience, February 2016.

[3] Gareth Loy, Alexander Zelinsky; *Fast Radial Symmetry for Detecting Points of Interest*; IEEE transactions on pattern analysis and machine intelligence, August 2003.

[4] Bruce D. Lucas, Takeo Kanade; *An Iterative Image Registration Technique with an Application to Stereo Vision*; International Joint Conference on Artificial Intelligence, 1981.

[5] Evangelos Skodras, Nikolaos Fakotakis; *Precise Localization of Eye Centers in Low Resolution Color Images*; Elsevier, December 2014.