Assignment 1 - part 2

This document is the second part of the first mandatory assignment.
According to the plan you should now have implemented methods to detect the pupil and glints. You might worked on eye corner detection (optional). This week you will be working on methods to improve you results and extend your implementation with Limbus (iris region) detection using some of the techniques from the "Segmentation lecture" Iris detection can then be used to remove those reflections that are not within the iris boundary. There are still a couple of weeks before you have to hand in this assignment, but next week the exercises will be allocated to linear algebra. You therefore have plenty of time to make further improvements to the assignment.

Questions and numbers

The text contains several questions that will guide you through the assignment. Sections marked with a ([1]) are required to do; higher numbers are optional but is intended to improve the overall report. Individual questions marked with '*' are not required (e.g. if time is short), but can help you to a better solution if time permits.

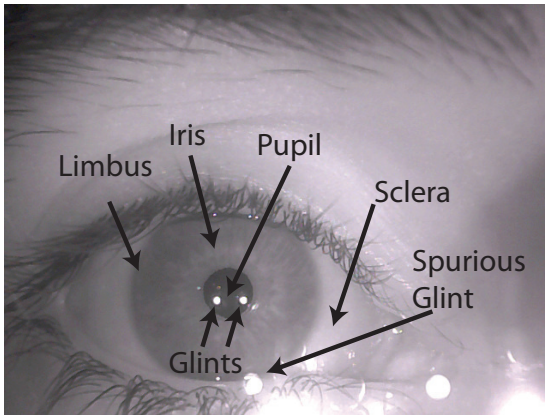

Figure 1: Various eye features

1 Pupil Detection using clustering [1] (Expected time: <=1 hrs)

As discussed in the lecture, automatic segmentation of images can be done though machine learning techniques such as k-means. In the following, you will experiment with k-means clustering (segmentation) to automatically detect the pupil and perhaps the iris. The main purpose of using k-means in this exercise is to avoid defining threshold values. The *detectPupilKMeans* function in the Assignment1.py performs K-means clustering on an image using both intensity and position as feature vectors.

Hints about the function: *centroids* consists some info about the intensity of each cluster. *Label* is actually an one dimensional image (a list) of cluster labels for each pixel.

1. Call the *detectPupilKMeans* function from *update()*. Make your program load the *eye1.avi* file.

In the following questions you will experiment with the *distanceWeight* and the number of clusters *K* as to determine good parameter settings.

2. Run *detectPupilKMeans* when changing *distanceWeight* from 1 to 50 in steps of 8. What is the effect of this change?

3. For which values of *distanceWeight*, does *detectPupilKMeans* give a good detection the pupil? Use the default value of *K*.

4. Find values of K (from $2-20$) that separates the pupil? Use the default value for *distanceWeight*.

5. Find values of *K* and *distanceWeight* that separates the pupil as one cluster. How well do these values apply to other sequences?
6. [2] Can you improve the result by smoothing the image *smallI* in function `detectPupilKMeans`?
7. Implement BLOB detection on *labelIm* to detect the pupil. Notice, clustering is done on *smallI*, which is smaller than the input image.
8. What could be the advantage of using k-means clustering before blob detection?
9. Implement a method that uses the results from K-means to automatically set the thresholds for pupil-blob detection.
10. Can K-means be used instead of blob detection?

Hints: You can either use the intensity of the darkest cluster for thresholding the original image and then doing the blob detection OR you can extract the darkest cluster and doing the blob detection on that cluster image.

2 Iris/Limbus detection

The aim of the this section is to implement methods that detect the limbus (iris boundary) through variants of the Hough transform. Limbus detection may be more challenging than detecting the pupil. You will therefore start with testing your implementation for pupil detection.

2.1 Blob detection of the iris[1] (Expected time: 0.5 hrs)

11. Implement a function (*GetIrisUsingThreshold*) that attempts to find the iris center using an approach similar to the ones used for pupil and glint detection. Just by changing the threshold values interactively you may realize that the iris can be hard to detect using such an approach. Give it a try, anyway.

2.2 Edges and gradients[1] (Expected time: 2 hrs)

1. Define a function `getGradientImageInfo(I)` that given a gray scale image returns images gradient Images (G_x,G_y), gradient magnitude (G_m) and gradient directions(G_d). Recall that
   - Orientation: θ(i,j) = atan(G_x(i,j) , G_y(i,j)) * 180/π
   - Gradient magnitude: ∇ I(i,j) = sqrt(pow(G_y(i,j),2) + pow(G_y(i,j),2)

   Hint: You might want to use numpy.gradient or cv.sobel for making the gradient image.
2. Describe the properties of the gradients, gradient magnitude and gradient directions in the eye images.
3. [3] Make quiver plots using matplotlib (of a subset) of the gradients in the eye image similarly to those found in 2. In order not to get too many vectors you may want to subsample the number of gradients (through slicing). What do you observe? Do these images provide any information on the location of the pupil, iris and glints (hint. Look at where the gradients meet and their length)?

The pupil and iris are well described by ellipses. A common step is to find good candidates for the boundary in the image where the gradient magnitude are large and then fit an ellipse to a subset of these points (see figure 2 right).

In the following steps your aim is to improve the localization of the pupil and iris contours. The term, *improve* may in this context mean either that the precision gets better or that you can relax the assumptions on the values of e.g. intensity thresholds or sizes.

Through the previous steps you have hopefully found a good localization of the pupil and a rough estimate of its size. You may want to use this information to make a contour-based localization of the pupil by first detecting edges and then fit an ellipse to a subset of the found edge pixels. When using contour-based methods you can use the gradient magnitude calculated in the whole image, but you can also save computations by only calculating the gradients along 1D directions. In this exercise you will experiment with both method. Common for these methods

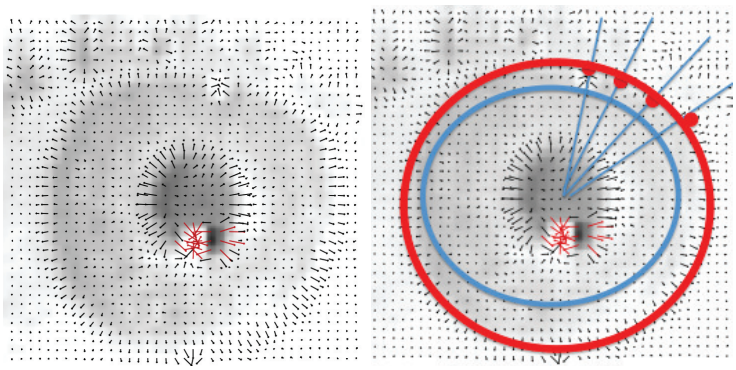

Figure 2: (left) Quiver plot of the intensity gradients. Do not use this image in your report :)

is to measure edge-ness in the image along the curve normals.

Recall a circle parameterized by t with radius R and can be written as a function in two parameters

$$(x(t), y(t)) = R * (cos(t), sint(t))$$

and the gradient in t can be found by differentiating $x(t)$ and $y(t)$

$$(x'(t), y'(t)) = R * (-sin(t), cos(t))$$

4. The function *getCircleSamples*, will given a radius center, C, and radius and the number of curve point samples, return a list of *nPts* tuples of (x, y, dx, ty) sample points on the circle, (x, y) and the curve gradients (dx, dy).

The function *circleTest* below shows how you can obtain curve samples from a circle using *getCircleSamples()* located in *SIGBTools.py*. The output from `circleSampls(...)` is a tuple of the points (x,y) and the curve gradient(dx,dy).

```
def circleTest():
  nPts = 20
  C = (100,100)
  circleRadius = 40;
  P= getCircleSamples(center=C, radius=circleRadius, nPoints=nPts)
  t=0;
      for (x,y,dx,dy) in P:
          <do something clever>
```

5. Extend *circleTest* so that it shows the circle points in an image. Use the center of the detected pupil blob as the center of the circle.

6. Extend *circleTest* so that it shows the curve normal for each point. Hint use *cv2.Line(img,p1,p2)* to draw the line between the points *p1* and *p2* in `img`. You probably want to scale the length of the curve normal to be longer than 1.

7. Implement a method that given the initial position (and size) of the pupil will find the boundary of the pupil using the maximum gradient magnitude along the normals. The aim is to be able to get the contour of the pupil/iris more robustly (e.g. thresholds and sizes do not need to be defined as accurately). Pseudocode for locating the boundary of the pupil/iris is given below:

```
def findEllipseContour(img, gradientMagnitude, estimatedCenter, estimatedRadius,nPts=30):
      P= SIGBTools.getCircleSamples(center=estimatedCenter,radius=estimatedRadius ,nPoints=
          nPts)
```

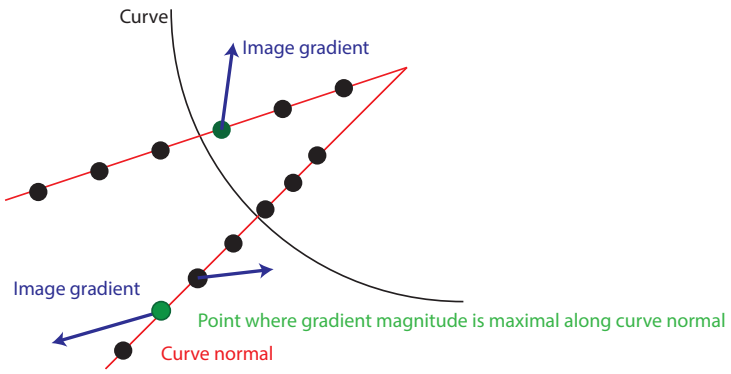

Figure 3: Curve normals(red), image gradients(blue) and feature point/point of maximal gradient (blue)

```
        t=0;
        nPts = 30
        newPupil = np.zeros((nPts,1,2)).astype(np.float32)
        for (x,y,dx,dy) in P:
        #< define normalLength as some maximum distance away from initial circle >
        #< get the endpoints of the normal -> p1,p2>
        #< maxPoint= findMaxGradientValueOnNormal(gradientMagnitude,p1,p2) >
        #< store maxPoint in newPupil>
        t=t+1;
          <fitPoints to model using least squares- cv2.fitellipse(newPupil)>
    return ellipseParameters

def findMaxGradientValueOnNormal(gradientMagnitude,p1,p2):
    #Get integer coordinates on the straight line between p1 and p2
    pts = SIGBTools.getLineCoordinates(p1, p2)
    #normalVals = gradientMaginitude[pts[:,1],pts[:,0]]
      #Find index of max value in normalVals
    #return coordinate of max value in image coordinates
```

8. Finish the implementation of *findMaxGradientValueOnNormal* and display the points of maximum value in a temporary image for verification.
9. Test your method in various sequences. How well does the method to detect the boundary of the pupil?
10. Where does the method perform well and when does it fail?
11. [2] Evaluate if you could use Canny's edge detection as input the to contour method (`cv2.Canny`). What would be the advantages and disadvantages of using Canny here? try it!
12. Use the same method to estimate the iris. How are the results compared to pupil detection?
13. Use the (curve) gradient direction to disregard those pixel gradients directions that are not sufficiently aligned with the curve gradient (e.g. the angle between them is small) - see figure 3. You can for example use the angle between the curve normal and image gradient. Notice the vectors should be normalized to length 1.

Ellipse fitting

14. Evaluate your method (challenges, robustness etc.) in the report and discuss the challenges and advantages of the method.

In the method just implemented, the gradients of all values in the image were computed but many of them were never used in the calculation of the contour. Instead you can calculate the pixel differences along the curve normal and for example use the point of maximum difference along the 1D line.

15. * Implement a method so that you are able to get the contour of the pupil/iris more robustly using the maximum along the normal. Use *cv2.fitEllipse* to fit the points.
16. * Compare the method with the previous methods.

2.3 Hough Transform[2] (Expected time: <1 hrs)

You may have realized that it can be hard to find a generic threshold for pupil-blob detection that allows you to detect the whole pupil. Edges may define boundaries of objects and can give you sufficient information to detect the pupil e.g. through the Hough transform. The Hough transform is a (relatively) robust technique for detecting objects. You will in this section experiment with the (circular) Hough transform for pupil and iris detection. Initially you will experiment with the "plain vanilla" Hough transform implemented in OpenCV. If you have time you can implement a efficient variant afterwards (see end of this document).
The *detectPupilHough* function in the Assignment1.py shows an example on how to detect circles using the Hough transform in OpenCV.

17. Make changes to this function in the following steps.
18. Make *Update* call *detectPupilHough* and ensure your program loads *eye1.avi*.
19. Experiment with the values *accThr* = $100, 200, 400, 600, 700, 800$. What happens when you increase the threshold value and what is the reason for this behavior?
20. Change the values for *maxRadius* and *minRadius* so that it (as good as possible) locates the pupil in the images. It is often better to have too many detections than missing one. Change the parameters of *cv2.HoughCircles* so you have as few missing detections as possible while minimizing the number of false positives.
21. [2] Can you improve the detection results by changing the amount of smoothing performed before using the Hough transform. How does smoothing effect the parameters of *cv2.HoughCircles*.
22. Make a new function *detectIrisHough* that uses the Hough transform to detect the iris. You have to change the parameters to reflect the fact that the iris is larger.
23. Use *detectPupilHough* and *detectIrisHough* to detect the iris and pupil in other sequences.
24. [3] Make new sliders so you can change some of the most relevant parameters of the two functions interactively.

For the report In the report you can discuss whether you can use the same parameters in all the sequences or which changes are needed needed to make the method work in most of the sequences? What are the limitations of the Hough transform for pupil and iris detection. Which parameter settings did you use? How much blurring did you apply?

25. Implement a function

```
def filterGlintsIris(glintList,iris):
```

that removes all those glints that are not within the iris circle. A point p is inside a circle with radius R, if the distance between the point p and the circle center is less than R.

2.4 Hough Transform with a position prior[3] (Expected time: 1 hrs)

The Hough transform is fairly accurate in terms of size and position, but it often finds too many or too few candidates. In the following steps you will combine the detection results from the blob detection with a Hough transform method to detect the pupil and iris. The method will use the already detected pupil center (thresholds, BLOBS / k-means) in which you will use the already detected pupil center. In the following assume the pupil is a circle(3 parameters). When the pupil center has been detected, only one parameter remains, namely the radius. You can use a 1D Hough transform to accumulate the contributions of the edges from the center to the edge pixel e.g. bin all radii from a edge-pixel to the center. The code snippet below is meant as an inspiration for your implementation of the simplified Hough transformation

```
def simplifiedHough(edgeImage,circleCenter,minR,maxR,N,thr):
    '''
    simplifiedHough estimate circles in an edge image with a predefined center and returns a
        list of circles with circle radii
  Parameters:
    edgeImage       :    A binary image of edges (eg obtained from canny)
    pupilCenter     :    Coordinates of the center of the circle
    minR            :    Minumum Radius
    maxR            :    Maximium Radius
    N               :    Number of bins between minR og maxR
    thr             :    Threshold value of the bins

    RETURNS: a list of radi [r_1...,r_k] for which the bins in hough space are above thr
    '''
```

26. Implement the body of *simplifiedHough* so it locates circles using the simplified Hough transform. Hint, you can use the *nonzero* function (numpy) to find the indices of edge pixels.

27. Use your implementation of *simplifiedHough* to locate the pupil and iris and evaluate when the method works and when it fails.

28. Verify your method on different sequences.

29. [2] How can the results from pupil blob detection and detected circles from the Hough transform combined to remove spurious (false) pupils? Implement your ideas.