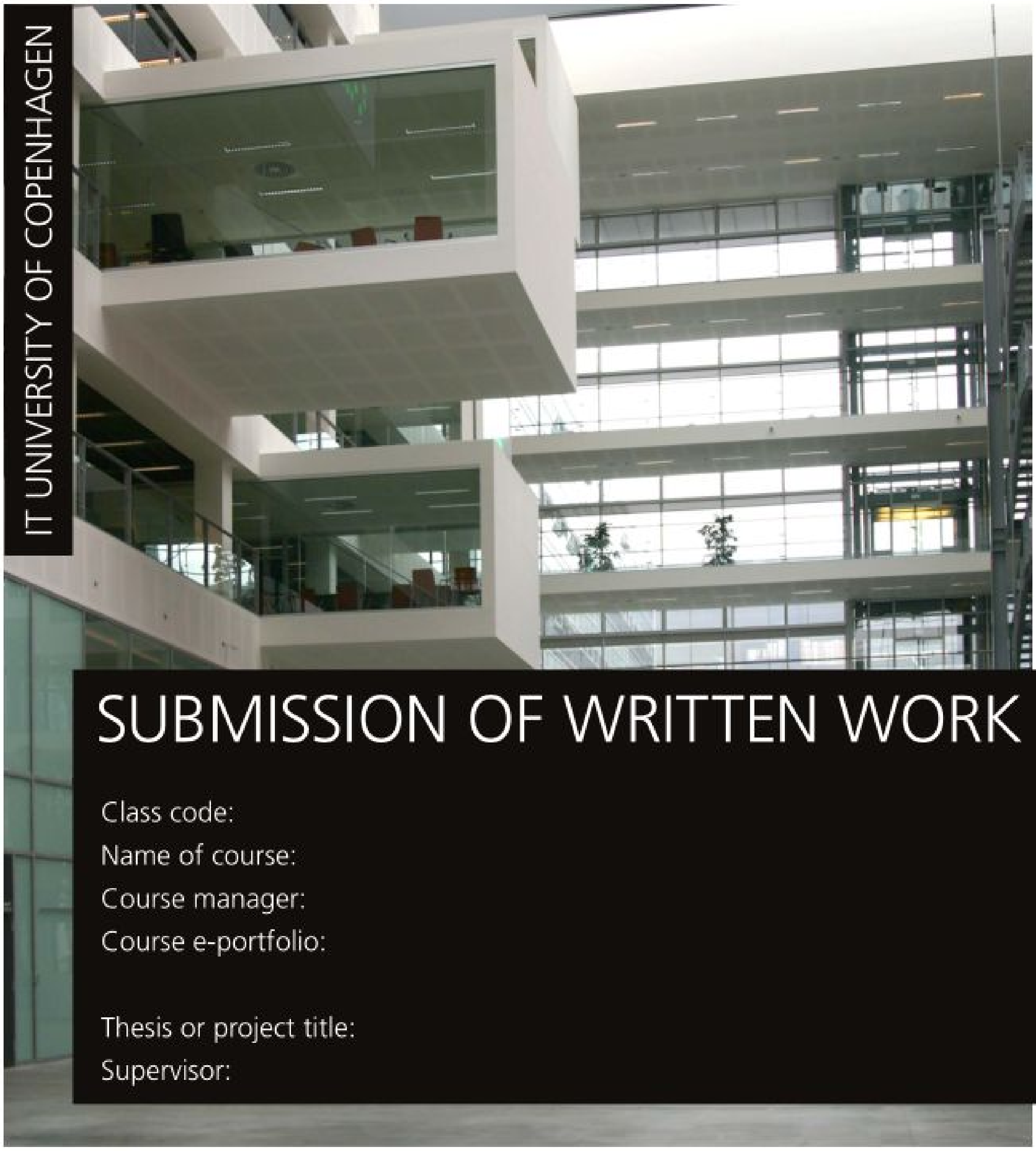


IT UNIVERSITY OF COPENHAGEN

SUBMISSION OF WRITTEN WORK

Class code:

Name of course:

Course manager:

Course e-portfolio:

Thesis or project title:

Supervisor:

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Emil Christian Lynegaard	17-07-1994	ecly@itu.dk
2.	Steffen Haugaard Immerkaer	14-10-1993	shaw@itu.dk
3.			@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk

Assignment #01

Emil Lynegaard (ecly@itu.dk) & Steffen Immerkær (shau@itu.dk)

IT University of Copenhagen (ITU)
Introduction to Image Analysis and Machine Learning
(Spring 2017)
March 8, 2017

Disclaimer:
As we struggled with getting the given infrared cameras to work, we have not included outputs of a custom made video, and instead used the given Input videos in the report and for our output.

Table of contents

Exercise 1.01 **4**
- a) 4
- b) 5
- c) 6
- d) 7

Exercise 1.02 **9**
- a and b) 9
- c) 9
- d and e) 10

Exercise 1.04 **11**
- a) 11
- b) 12
- c) 12
- d) 12
- e) 13

Exercise 1.05 **14**

Exercise 1.06 **15**
- a) 15
- b) 15
- c) 15
- d) 16
- e) 17

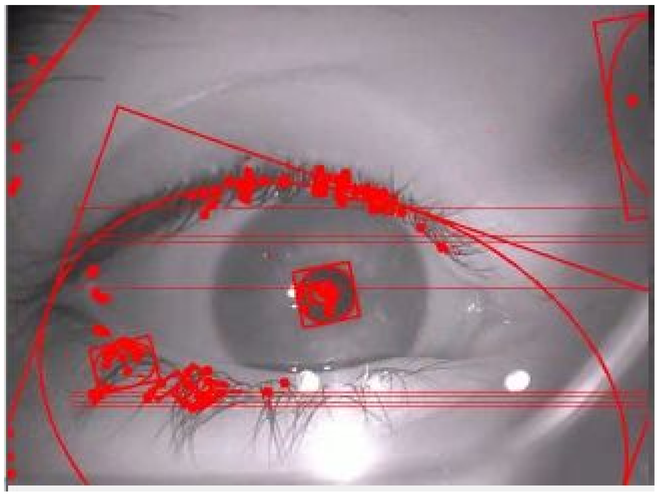
Exercise 1.01

a)

Below is shown the code added to the getPupil() method to achieve the goals of task (a).
The fit our contour to an ellipse or a rectangle depending on the amount of points within the contour. Furthermore we use the ‘calcContourProperties’ method to retrieve the Centroid of the contours, and lastly merely add all our contour fits and centroids to their relevant lists.

```
props = RegionProps()
for contour in contours:
    prop = props.calcContourProperties(contour, ["Centroid"])
    ellipse = cv2.fitEllipse(contour) if (len(contour) > 4) else cv2.minAreaRect(contour)
    centroid = prop.get("Centroid")
    ellipses.append(ellipse)
    centers.append(centroid)
```

With the above code we do no classification (other than the contour calculation itself), whereby we end up with a lot of false positives in our visualization. Depending on the threshold, where we used a standard of 79 - it shows itself as seen below:



Screenshot from Ex101_A01.m4v

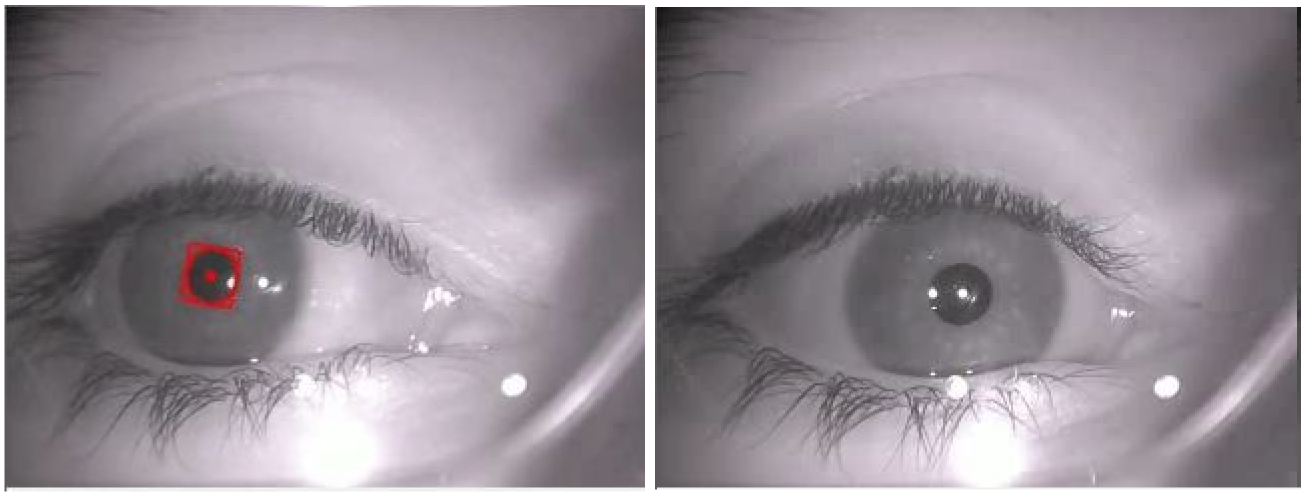
As we can see, ‘findContours()’ manages to find a huge amount of false pupil positives in the image, while it does however also manage to find the one true positive that we’re looking for.

b)

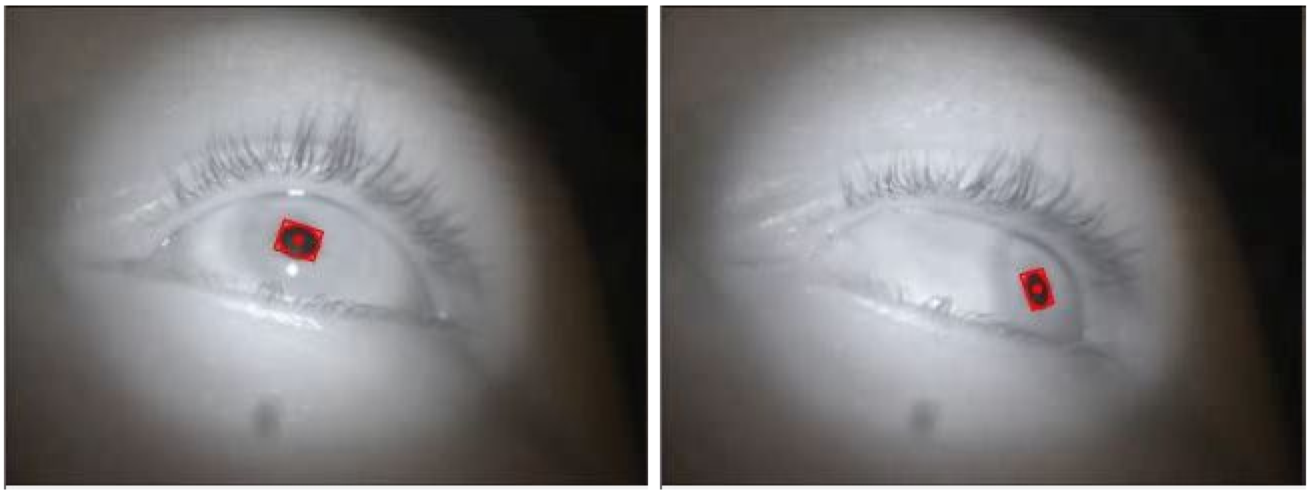
Now we have extracted additional properties from ‘calcContourProperties’ and used them to have our ‘getPupil’ method do some classification. Now we only consider a contour if it has an area between 500 and 6000 (in accordance with tip from assignment) and we also use its extend as a measure of circularity, such that very skewed ellipses are abandoned. The extra code is seen below:

```
area = prop.get("Area")
extend = prop.get("Extend")
if(area > 500 and area < 6000 and extend > 0.5):
    ellipses.append(ellipse)
    centers.append(centroid)
```

Below is shown an example where the new added code performs very well and similarly one of its shortcomings.


Two different screenshot from Ex101_B01.m4v

On the left we can see how it has gotten rid of all the clutter from the previous example, while we on the right see how our method lacks robustness when it comes to glints within the eyes. These glints are very pronounced in video ‘eye04.m4v’ where we with our other example video ‘eye14.m4v’ achieve near perfect results with this simple classification rule as seen below:



Two different screenshots from Ex101_B02.m4v

c)

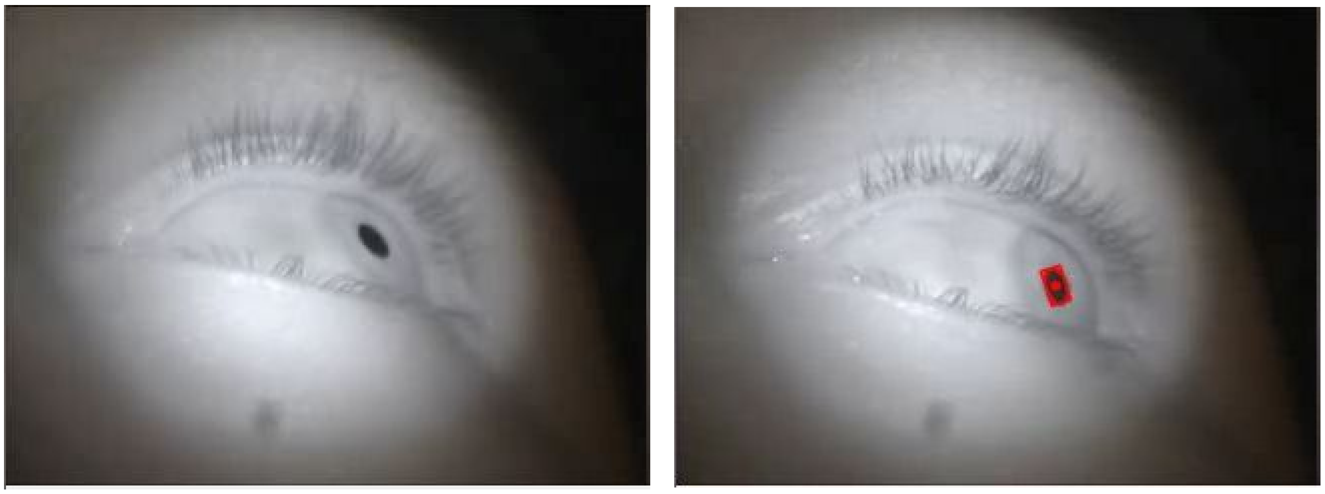
To enhance our classifier further, and to start using the extra return value 'bestPupil' of the 'getPupil' method, we extract the 'Extend' property from the Contour Properties. To keep track of the best positives that get past our classifier, we will specifiy the best candidate as the contour with the largest area. The code below is used to do this:

```
centroid = prop.get("Centroid")
area = prop.get("Area")
extend = prop.get("Extend")
if(area > 500 and area < 6000 and extend > 0.5):
    ellipses.append(ellipse)
    centers.append(centroid)

    if (area > largestArea):#define best pupil as largest area
        largestArea = area
        bestPupil = len(ellipses) - 1
```

We use the extend property to remove contours that are very skewed, as we operate under the assumption that are very circular, hence their extent should be ideally close to 1. Naturally we have to use a much lower value since we do not handle perspective otherwise, and from our tests 0.5 seemed to yield good results. The check for best pupil, merely sets the value based on the index of our newly added object in its list.

Since our code already worked rather well on our test cases, this addition did not add huge improvements to our results. We still very much struggled with glints in one of our videos, but as we are not actively doing anything to circumvent this at the moment, it makes sense as they are very pronounced and break the pupil's circularity.

Most notably, we now find no false positives for our example video 14, and only once throughout the video, loses track of the pupil, while the subjects looks far inwards. Below this is shown:



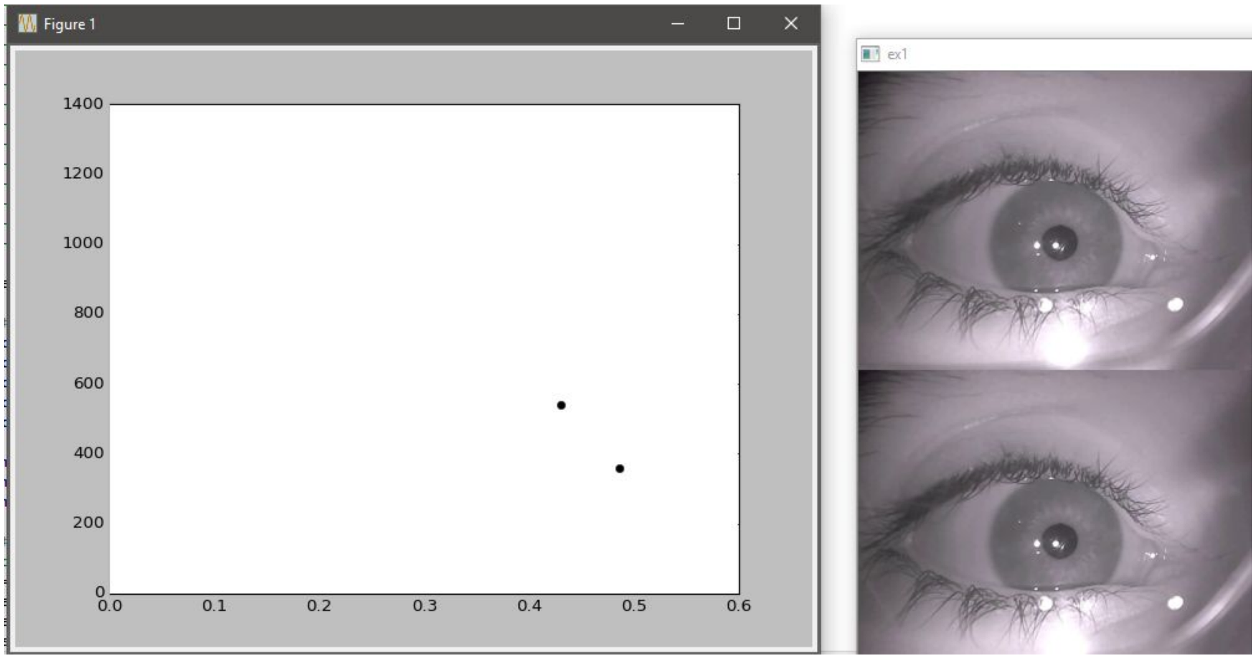
Two screenshots from Ex101_C02.m4v with a failing and succeeding corner case

d)

To plot our data we calculate two lists of each of our candidates extends and areas. When all candidates are found we plot this data by calling the __plotData function. Below is seen the addition to 'getPupil':

```
areas = list()
extends = list()
...
extends.append(extend)
areas.append(area)
...
self.__plotData(extends, areas, bestPupil)
```

As our classification almost never yields multiple positives, the resulting graph is a rather dull looking one, with mostly a single moving data point. Interestingly though, below is depicted one of the edge cases where we see the graph with last depicted results, just before we lose track of the eye. As we can see from the below picture, it happens due to both area and extend dropping below our set values, most likely due to glints. This could potentially be resolved by reducing the threshold, but this would also increase the chance of false positives.

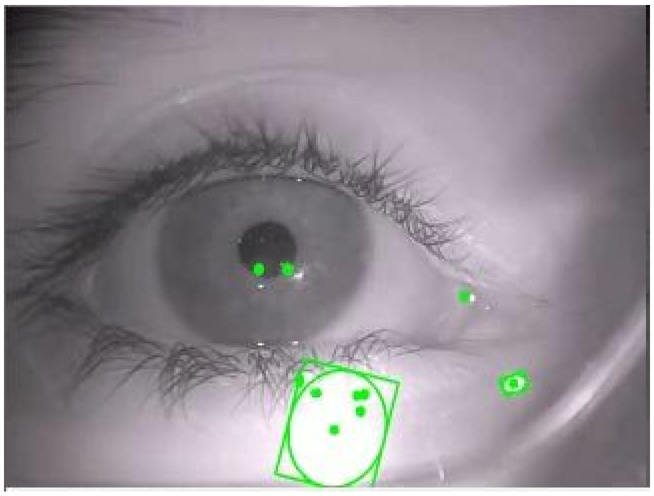
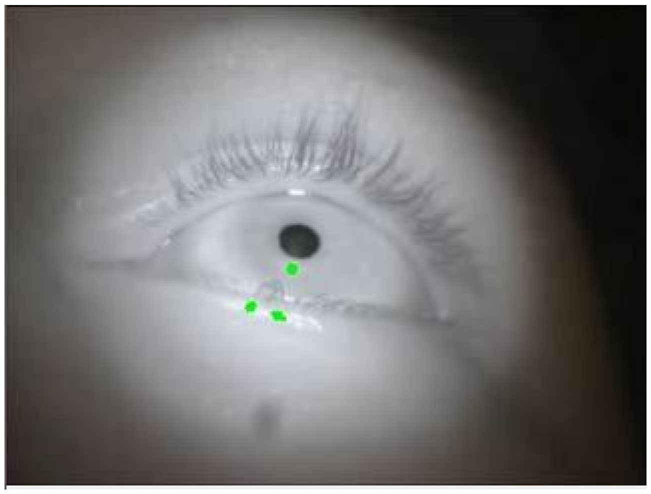


Y=area, X=Extend. From video example 04.

Exercise 1.02

a and b)

For the initial findings of glints in the eye, we use the exact same code that we did in our pupil detection with the exception of our threshold type now being ‘cv2.THRESH_BINARY’ in contrast to our earlier used ‘cv2.THRESH_BINARY_INV. This is due to the fact that we’re now searching for really bright blobs rather than really dark ones. This change as well as using a classification of ‘area > 15’ to avoid noise, results in the following:

Left: Screenshot from Ex102_A02.mp4, Right: Screenshot from Ex102_B02.mp4

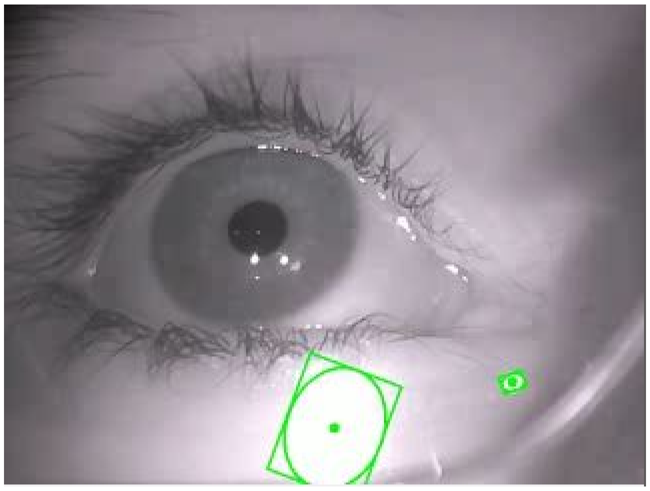
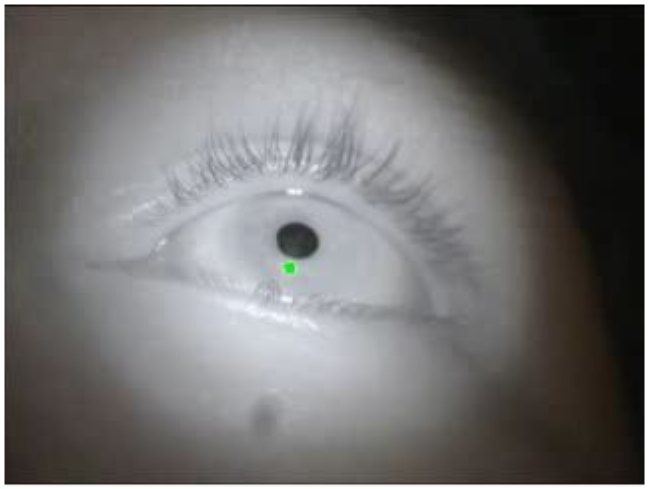
For the most part we find all the true positives as well as an excess of false positives. In some cases on video example 14(B), we do however lose track of the glint we’re searching for, due to it’s size and our removal of glints with areas smaller than 15.

c)

To further our glint finding abilities, we extend the method by firstly, sorting our ellipses based on their area using formula PI*A*B. Secondly, we retrieve the centers of these newly ordered rotated rectangles, by looking at the first tuple, which holds their center.
Finally, since we’ve ordered our lists, we merely return a list of the numbers 0 … numOfGlints, with the exception, that numOfGlints should never exceed the amount of glints we’ve found.
The code is shown below:

```
 #sort based on size, since first index of tuple is the 2 radii
ellipses.sort(key=lambda ellipse: math.pi * ellipse[1][0] * ellipse[1][0], reverse=True)
centers = map(lambda ellipse: (ellipse[0][0], ellipse[0][1]), ellipses)
numOfGlints = numOfGlints if len(ellipses) >= numOfGlints else len(ellipses)
bestGlints = [i for i in range (numOfGlints)]
```

This additional code yields results as seen below:

Left: Ex102_C01.mp4, Right: Ex102_C02.mp4

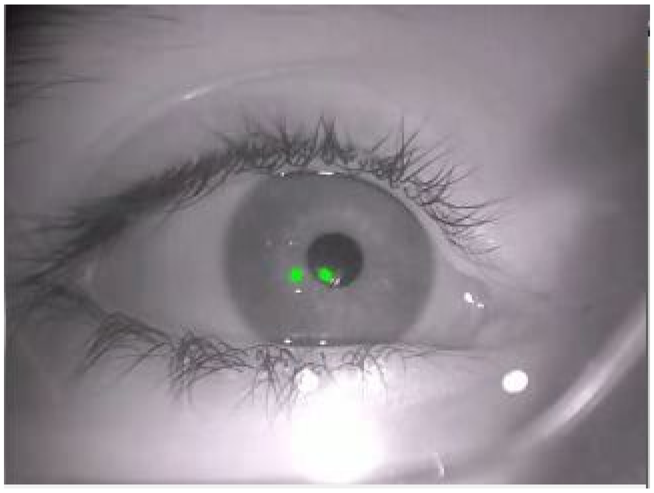
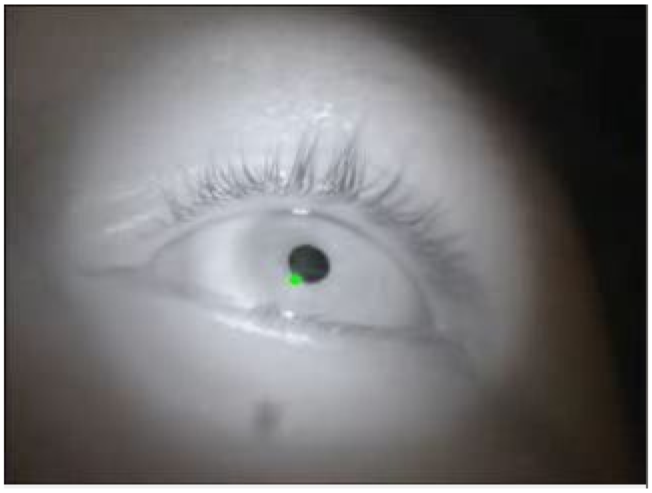
Clearly, this addition gives varying results. On the left image, where we have a large glint that is not on the cornea, hence yielding bad results, while on the right, the only largest glint is exactly the one we're looking for. This is clearly not a very robust measurement for 'best glint', and thus we will solve this in the following task.

d and e)

For calculating the euclidean distance, we use the SciPy code given with task d. For actually sorting our glints based on the euclidean distance to the center of the pupil, we modify our sorting code from task C, in the following way:

```
ellipses.sort(key=lambda ellipse: self.__Distance(ellipse[0], pupilCenter))
```

We no longer need to reverse it, as we are indeed looking for the smallest distance, and other than that, it is merely using the value returned from our Distance method that is used for the sorting. All the other codes remained the same, while this small change, made a big difference in terms of results, as seen below:

Left: Ex102_E01.mp4, Right: Ex102_E02

As we can see now, by adjusting our 'numOfGlints' variable, we find exactly the glints that we're looking for. Seeing as this variable is adjusted by the user, it is easy to merely set it in accordance with the amount of glints that we know are on the cornea. We still occasionally lost track of the glint in video example 14, so we may have to adjust our area classification a bit more.

Exercise 1.04

a)

For finding both the gradient and its magnitude and orientation, we use the code below:

```
sobelx = cv2.Sobel(grayscale, cv2.CV_64F, 1, 0, ksize=5)
sobely = cv2.Sobel(grayscale, cv2.CV_64F, 0, 1, ksize=5)

gradient = np.gradient(grayscale)
magnitude = cv2.magnitude(sobelx, sobely)
orientation = cv2.phase(sobelx, sobely, angleInDegrees=True)
```

We use the built-in numpy function for calculating the gradient based on our grayscale image. Additionally we apply the Sobel filter both horizontally and vertically with which we are able to use OpenCV to derive the gradient magnitude and orientation.

b)

Gradients are very useful for edge detection, as we're trying to find difference in intensity, to distinguish between various objects in an image.

Due to the properties of pupils, specfically their incredibly dark color, gradient magnitude is a great property to further our abilities in terms of finding finding pupils in eye images. We know that for most irises, the gradient magnitude will be very large around the pupil's periphery.

Lastly, seeing as we're able to fairly confidently find pupil's in images, we can use the properties of gradient orientation to better our iris detection capabilities. Since we know that the sclera is white, the gradient orientation at the edge of the iris and the sclera will be pointing towards the pupil. This property allows us to more confidently detect the edges of the iris.

c)

For finding the maximum gradient along a normal, we added the simple code below to the __FindMaxGradientValueOnNormal() function:

```
maxVal = -1
index = -1
for i,val in enumerate(normalVals):
    if val >= maxVal:
        index = i
        maxVal = val

maxPoint = points[index]
```

We simply iterate through our normalVals that have been generated based on the points array. When we have found the largest value and its index, we extract the point at the same index from the points array, and store this point in the maxPoint variable.

d)

To get this task done, we had to add a little extra code outside of the specified 'YOUR CODE HERE' areas. This was the case due to our circleSamples being generated inside the iris rather than outside the iris. While we could move these samples along the normal, we simply scaled the parameter of 'getCircleSamples' instead, such that the call look as follows:

```
circle = getCircleSamples(pupilCenter, pupilRadius * 2, numOfPoints)
```

Now that we have circle samples outside the iris, and we know already know the center of the pupil, our 'FindMaxGradientValueOnNormal' function should be able to almost do the rest. The problem we however naturally ran into, was that it always returned the edge of the pupil, as this was the point with the largest gradient. To circumvent this problem. We moved the pupil center towards the circle sample point, 1.3 times the radius. This ensured that we'd only get points from inner iris and towards the outside of the iris. The code added for calculating the movedPupilCenter is the following:

```
for vals in circle:
    point = (vals[0][0], vals[0][1])
    direction = (point[0] - pupilCenter[0],point[1] - pupilCenter[1])
    length = math.sqrt(math.pow(direction[0], 2) + math.pow(direction[1], 2))
    direction = (direction[0] / length, direction[1] / length)#normalised
    scaledRadius = 1.3 * pupilRadius;# to ensure point is moved away from pupil entirely
    movedPupilCenter = (pupilCenter[0] + direction[0] * scaledRadius, pupilCenter[1] +
direction[1] * scaledRadius)
```

Now that we have this code, we are almost done. Now we use our previously implemented __FindMaxGradientValueOnnormal function, to find the iris periphery (hopefully), gather the periphery points all the way around, and fit an ellipse to these points. The code seen below is what we use for this:

```
#… 'within for vals in circle loop'
    max = self.__FindMaxGradientValueOnNormal(magnitude, orientation,  movedPupilCenter,
          (vals[0][0], vals[0][1]))
    maxVals.append(max)

#out of loop
points = np.asarray(maxVals)

ellipse = cv2.fitEllipse(points)
```

This code alone results in very reasonable iris detection.

e)

To further our already pretty good iris detection abilities, we now add code, to ensure that the largest gradient we find, has the expected orientation within some arbitrary margin.
Below is our updated code for __FindMaxGradientValueOnNormal:

```
curveNormal = (p2[0]-p1[0], p2[1]-p1[1])#direction from center of pupil to periphery point
curveNormalLen = math.sqrt(math.pow(curveNormal[0], 2) + math.pow(curveNormal[1], 2))
curveNormalNormalised = (curveNormal[0] / curveNormalLen, curveNormal[1] / curveNormalLen)
curveDirectionInDegrees = math.atan2(curveNormalNormalised[0], curveNormalNormalised[1]) *
                          180 / math.pi

maxVal = -1
index = -1
for i,val in enumerate(normalVals):
    if val >= maxVal:
        point = points[i]
        if (point[0] < len(orientation) and point[1] < len(orientation[0])):
            currentOrientation = orientation[point[0]][point[1]]
            if (abs(currentOrientation - curveDirectionInDegrees) < 50):
                index = i
                maxVal = val

maxPoint = points[index]
```

First we calculate the curve's direction, in the form of the direction from our movedPupilCenter towards our circleSample. We then normalise this and calculate its direction in degrees. When we have this, we merely add to our conditionals, that this angle must not be more than 50 degrees different from the large gradient's orientation that we were already finding. This addition did not give any noticable changes in our resulting video, hence it appears that we were already finding reasonable points for our iris ellipse.

Exercise 1.05

"Tracking can be done by storing the previous state (e.g. position and size) and use that information in the following frame to detect the eye feature in a sub region of the analyzed image (ROI – Region of Interest). Tips: use the previously found locations to filter those eye feature candidates that are too far away. "

This could be done for example for the pupil by storing the previous frame's pupil candidate's bounding box, adding a margin to account for movement, and then search within this box in the next frame.

"You can use the eye corners and iris circumference to detect the sclera. Implement a method that does this. Could we have used the sclera to detect the other feature?"

A very simple, and thereby perhaps not the most robust way to do this could be to fit it using two second degree polynomials. These polynomials would be fitted using the points for the corner and a point with the x-value of the iris center, and the maximum y-value of the iris boundingbox. Once we have this sclera shape made by the two polynomials we merely subtract the shape of the iris. This method will of course only yield accurate results when the iris fills up exactly the height of the sclera.

Alternatively one could imagine find the top and bottom points, using a hierarchical gradient search from the center of the pupil, but again this would be prone failures in special cases.

"What effect does histogram equalization have on the eye images? Can it be used to improve or simplify your eye feature detection method?"

One can imagine histogram equalization yielding varying results on eye images based on what one is trying to detect. For example, based on our example videos, the skin is fairly bright and thereby closely resembles the sclera in terms of intensity. Histogram equalization on these examples would probably make these easier distinguishable. For pupil detection on our videos, histogram equalization would have no effect or slightly worsen the detection, as our contrast for the pupil is extremely large due to brightness of the video. If our videos were captured with less brightness and thereby everything including the pupil would be dark, histogram equalization would have a very positive effect in terms of pupil detection.

Exercise 1.06

a)

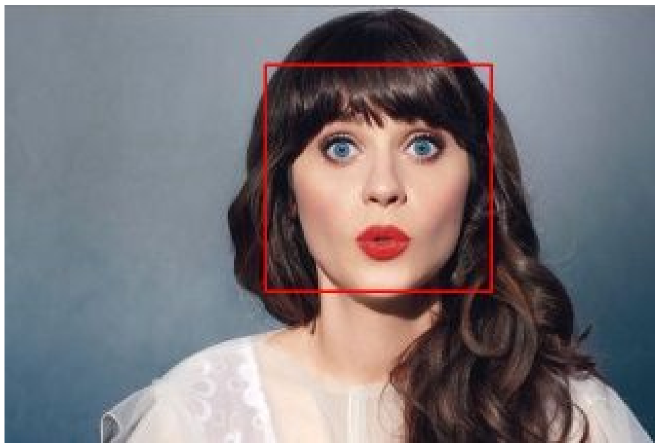
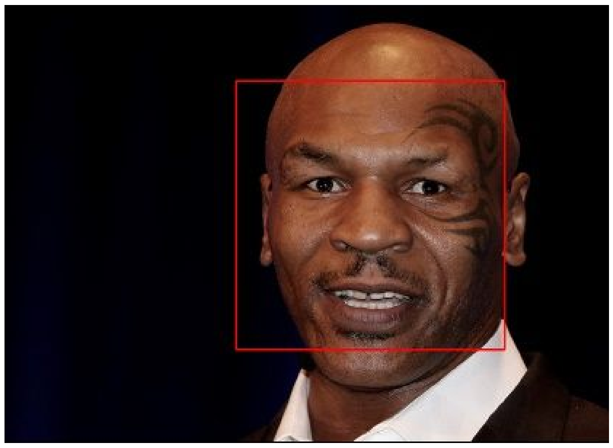
This task is completed by completing (b), (c), (d) and (e).

b)

```
faces = self.__cascade.detectMultiScale(grayscale, 1.3, 5)
```

We use the detectMultiScale method with the “haarcascade_frontalface_default” cascade classifier. We then set the variable “face” to the result of the above, and the prewritten code takes care of drawing rectangles for each face.

These are some examples of it working:

Faces detected using Haar Cascade Classifiers

c)

We slice the image with a simple black/white mask using the coordinates of the detected face/rectangle.

```
# create a mask
mask = np.zeros(image.shape[:2], np.uint8)
(x,y,w,h) = faces[0]
mask[y:y+h, x:x+w] = np.uint8(255)
```

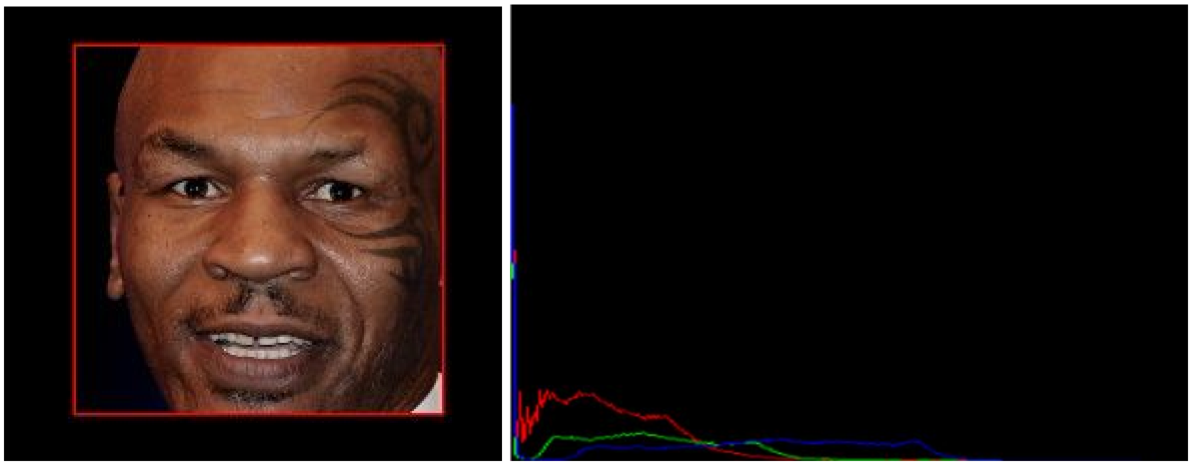
We start by creating a two-dimensional array with the size of the image. This array contains only zeros (black). We then take the first (and only) face, and fill the area with value 255 (white). This mask can then be used to calculate the histogram for a subsection of a image.

d)

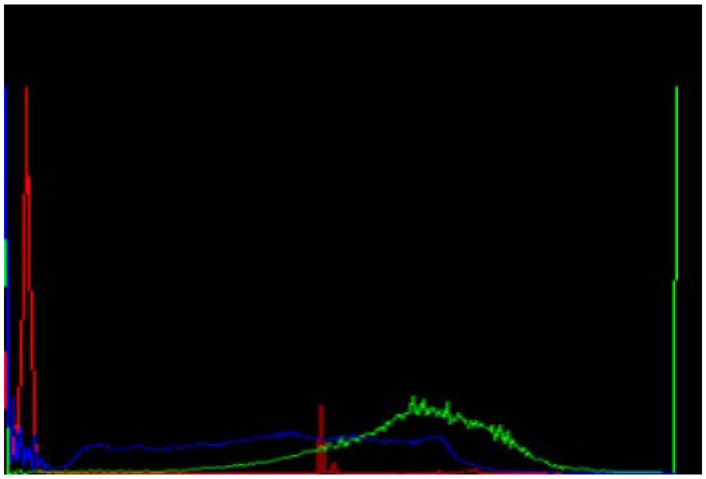
```
histR = cv2.calcHist([image], [0], mask, [256], [0,256])
histG = cv2.calcHist([image], [1], mask, [256], [0,256])
histB = cv2.calcHist([image], [2], mask, [256], [0,256])

hist = np.array([histR,histG,histB])
hist = np.transpose(hist)

hist = self.__getHistogramImage(hist[0], (255,255))
```

We start of by calculating the histogram of each color channel (BGR) - using the mask we just created. We then merge them to a numpy array and transpose it, to fit the prewritten method *__getHistogramImage*.

Below is an example of different histograms for the same face but using different color representations.



Histogram of Tyson's face in RGB



Histogram of Tyson's face in HSV

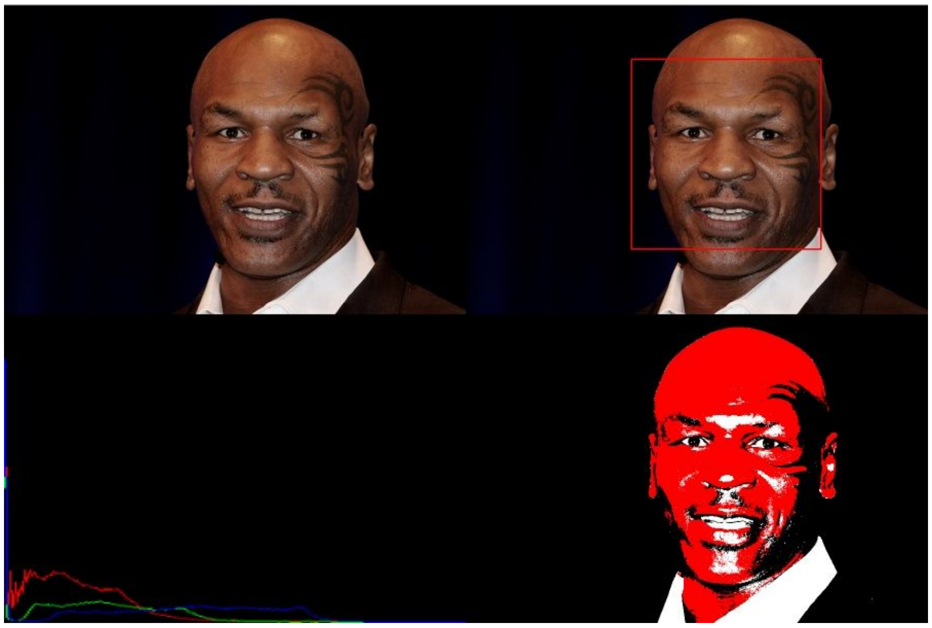
e)

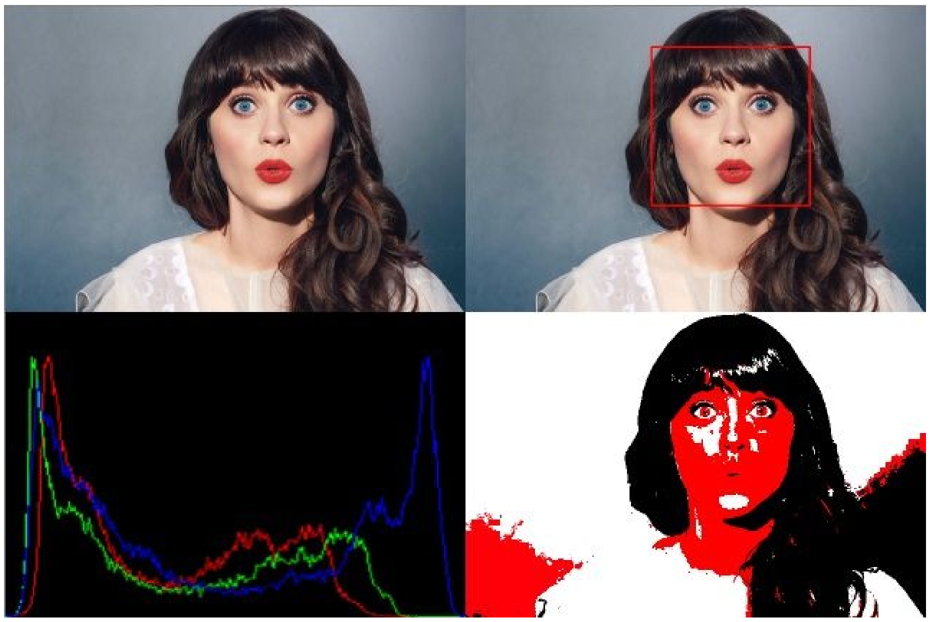
```
faceImage = cv2.bitwise_and(image, image, mask = mask)
hsv_face = cv2.cvtColor(faceImage, cv2.COLOR_BGR2HSV)

h,s,v = cv2.split(hsv_image)
hFace,sFace,vFace = cv2.split(hsv_face)

hValue,th1 = cv2.threshold(h,0,179,cv2.THRESH_BINARY+cv2.THRESH_OTSU)
sValue,th2 = cv2.threshold(s,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)
vValue,th3 = cv2.threshold(v,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)

hRet,threshH= cv2.threshold(h,hValue,179,cv2.THRESH_BINARY)
sRet,threshS = cv2.threshold(s,sValue,255,cv2.THRESH_BINARY)
vRet,threshV = cv2.threshold(v,vValue,255,cv2.THRESH_BINARY)

skin = cv2.merge((threshH,threshS,threshV))
skin = cv2.cvtColor(skin, cv2.COLOR_HSV2BGR)
```

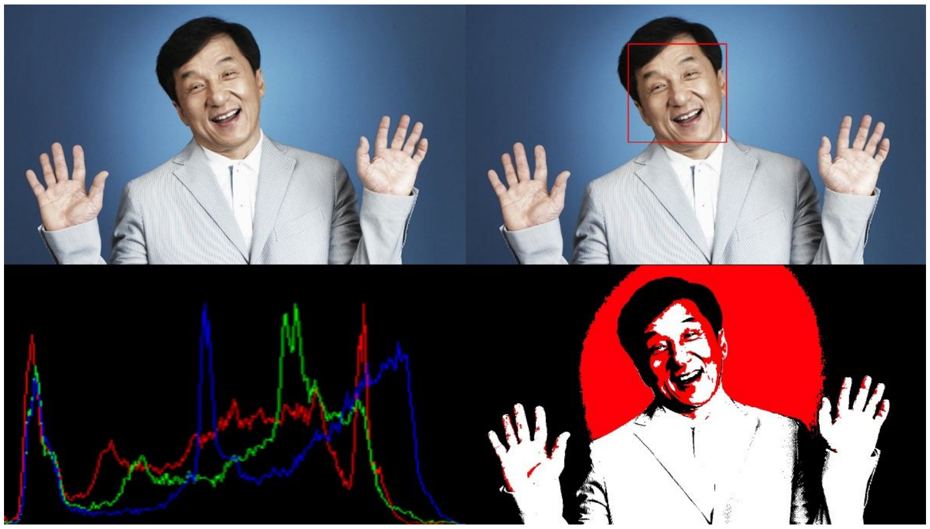
We start off by slicing the part of the image containing the face. Then we convert the sub-image from BGR color space to HSV color space. We chose to use HSV color space based on the observation that it represents the way humans experience colors better than BGR/RGB does. This is due to the face that we are able to separate the V channel from H and S, and thereby being more robust to light changes.

After that we use a binary threshold on the H and S channel of the hsv_face image with the THRESH_OTSU parameter. By doing this we obtain the threshold value for those two channels (hValue and sValue). Then we use those values to create a binary threshold on the H and the S channels of the hsv_image. Lastly we merge the thresholded H and S channels with the original V channel and convert it all back to BGR.

The result is what is shown below.







As you can see, we somewhat manages to capture the skin regions. We feel like we are mostly on the right track, but we still have some final thoughts and conclusions:

For a potentially better result we could have utilised only the H channel as suggested in the paper "*Skin detection using HSV color space".* This means that we could create a binary threshold using only the hue layer and then creating a black and white mask. White where the H value is within the threshold and black everywhere else. Then we can apply the mask to the original image, only showing the pixels in the white part of the mask.