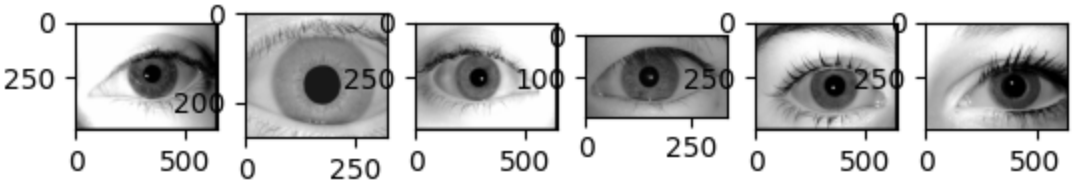
In [ ]:
```
import matplotlib.pyplot as plt
import matplotlib.image as img
import numpy as np

import cv2
import itertools
import math
from typing import Tuple, List

import scipy.ndimage as ndimage
from scipy.signal import convolve2d
```

In [ ]:
```
def rgb2gray(rgb_image):
    return np.dot(rgb_image[..., :3], [0.2989, 0.5870, 0.1140])
```

In [ ]:
```
image_nums = list(range(1, 7, 1))
image_filenames = list(f'iris{i}.jpg' for i in image_nums)
images = list(rgb2gray(cv2.imread(filename)) for filename in image_filenames)
gray_images = list(cv2.cvtColor(cv2.imread(filename), cv2.COLOR_BGR2GRAY) for filen
```

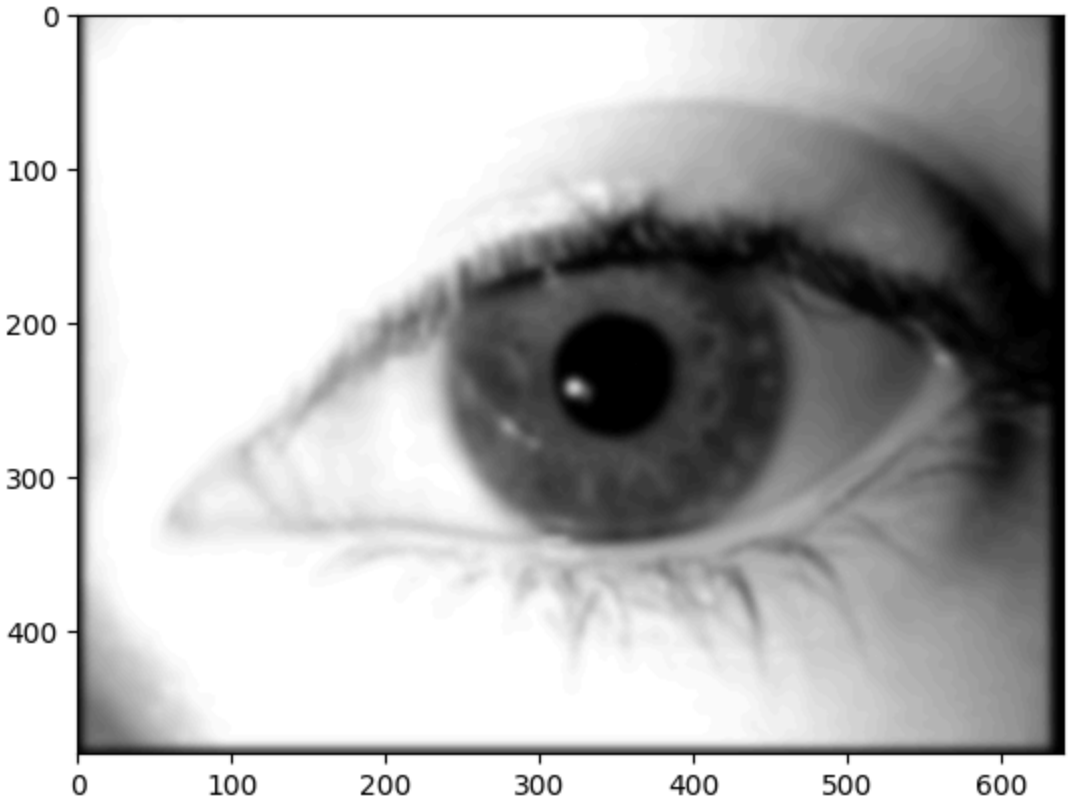
In [ ]:
```
fig, axes = plt.subplots(1, 6)

num = 0
for image in images:
    axes[num].imshow(image, cmap='gray')
    num += 1

plt.show()
```



In [ ]:
```
def imagePreProcessing(image, smooth_size):
    imageCopy = np.copy(image)
    plt.imshow(imageCopy)

    return ndimage.gaussian_filter(imageCopy, sigma=smooth_size)
```

In [ ]:
```
smooth_eye_imgs = list(imagePreProcessing(image, 3) for image in images)
plt.imshow(smooth_eye_imgs[0], cmap=plt.cm.gray)
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x137a74ff410>
```



In [ ]:
```
for s_img in smooth_eye_imgs:
    print(s_img.shape)
```

```
(480, 640)
(280, 320)
(480, 640)
(192, 334)
(480, 640)
(480, 640)
```

Find Edges with Convolution

Take the convolution of a Horizontal filter with the image to get the horizontal edges:

Horizontal Filter =

- [-1 -2 -1]
- [0 0 0]
- [1 2 1]

Take the convolution of a vertical filter with the image to get the vertical edges

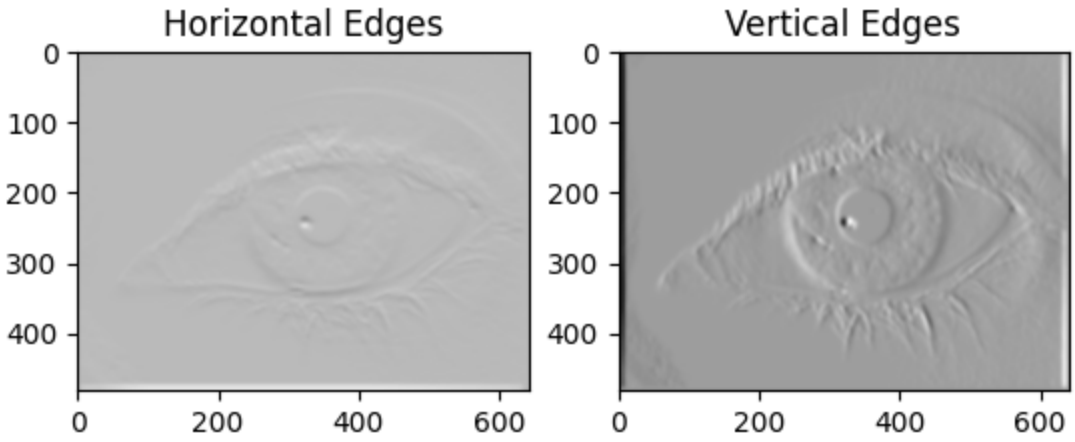
Vertical Filter =

- [-1 0 1]
- [-2 0 2]
- [-1 0 1]

In [ ]:
```
def show_horiz_and_vert_edges(horiz, vert):
    fig, ax = plt.subplots(1, 2)
    ax[0].set_title('Horizontal Edges')
    ax[0].imshow(horiz, cmap=plt.cm.gray)
    ax[1].set_title('Vertical Edges')
    ax[1].imshow(vert, cmap=plt.cm.gray)

    plt.show()
```

In [ ]:
```
horizontalFilter = np.array([[-1, -3, -1], [0, 0, 0], [1, 3, 1]])
verticalFilter = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])

horizontal_edges = list(convolve2d(smooth_eye_img, horizontalFilter, mode='full') f
vertical_edges = list(convolve2d(smooth_eye_img, verticalFilter, mode='full') for s

show_horiz_and_vert_edges(horizontal_edges[0], vertical_edges[0])
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x137a4c13bc0>
```



In [ ]:
```
show_horiz_and_vert_edges(horizontal_edges[1], vertical_edges[1])
```

In [ ]:
```
show_horiz_and_vert_edges(horizontal_edges[2], vertical_edges[2])
```

Get the Edge Map and Gradient Map

The edge map is:

E(img) = square root of (Horizontal edges^2 + vertical edges^2)

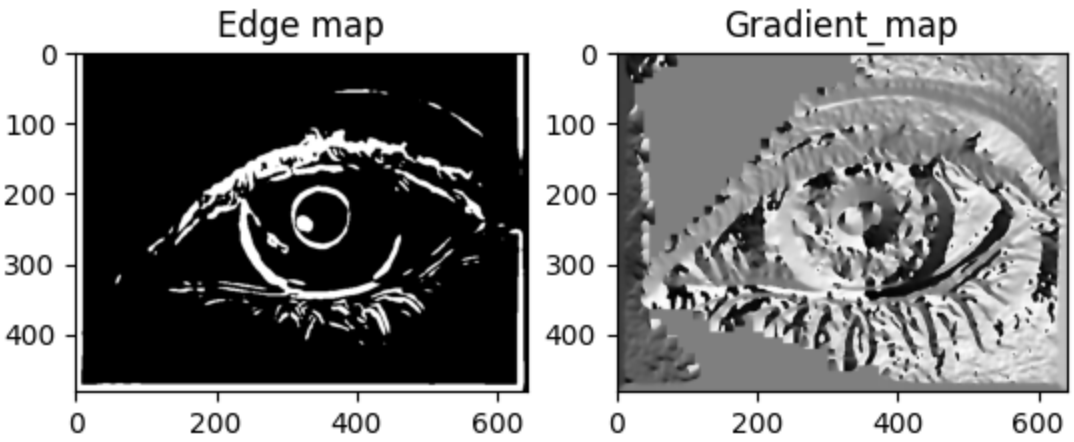
The gradient map shows the direction in which the edge is facing using arctangent

In [ ]:
```
# funciton is used to brighten the edge image for display
# takes an edge map and a filter amount
# the filter amount makes all pixels greater than the filter amount
# the desired maximum
# all other pixels will be the minimum
def filterEdgeMap(edge_img, filter_amount, max, min = 0):
    filtered_edges_matrix = np.zeros((edge_img.shape[0], edge_img.shape[1]))
```

```
    for x in range(0, edge_img.shape[0]):
        for y in range(0, edge_img.shape[1]):
            if(edge_img[x][y] > filter_amount):
                filtered_edges_matrix[x][y] = max
            else:
                filtered_edges_matrix[x][y] = min

    return filtered_edges_matrix
```

In [ ]:
```
edge_images = list(np.sqrt(np.add(horizontal_edges[i] ** 2, vertical_edges[i] ** 2)
gradient_matrices = list(np.arctan2(vertical_edges[i], horizontal_edges[i]) for i i
```

In [ ]:
```
bright_edge_matrix = filterEdgeMap(edge_images[0], 42, 255)

fig, ax = plt.subplots(1, 2)

ax[0].set_title('Edge map')
ax[0].imshow(bright_edge_matrix, cmap=plt.cm.gray)
ax[1].set_title('Gradient_map')
ax[1].imshow(gradient_matrices[0], cmap=plt.cm.gray)

plt.show()
```



In [ ]:
```
def plot_edge_and_gradient(edge_image, gradient_matrix, fig, i):
    bright_edge_matrix = filterEdgeMap(edge_image, 42, 255)

    ax = plt.subplots(1, 2)

    ax.set_title('Edge map')
    ax.imshow(bright_edge_matrix, cmap=plt.cm.gray)
    # ax.set_title('Gradient_map')
    # ax.imshow(gradient_matrix, cmap=plt.cm.gray)

    return ax

# fig, axes = plt.subplots(6, 1)

# for ax in axes:
#     ax.add_subplot(plot_edge_and_gradient(edge_images[i], gradient_matrices[i], f
```

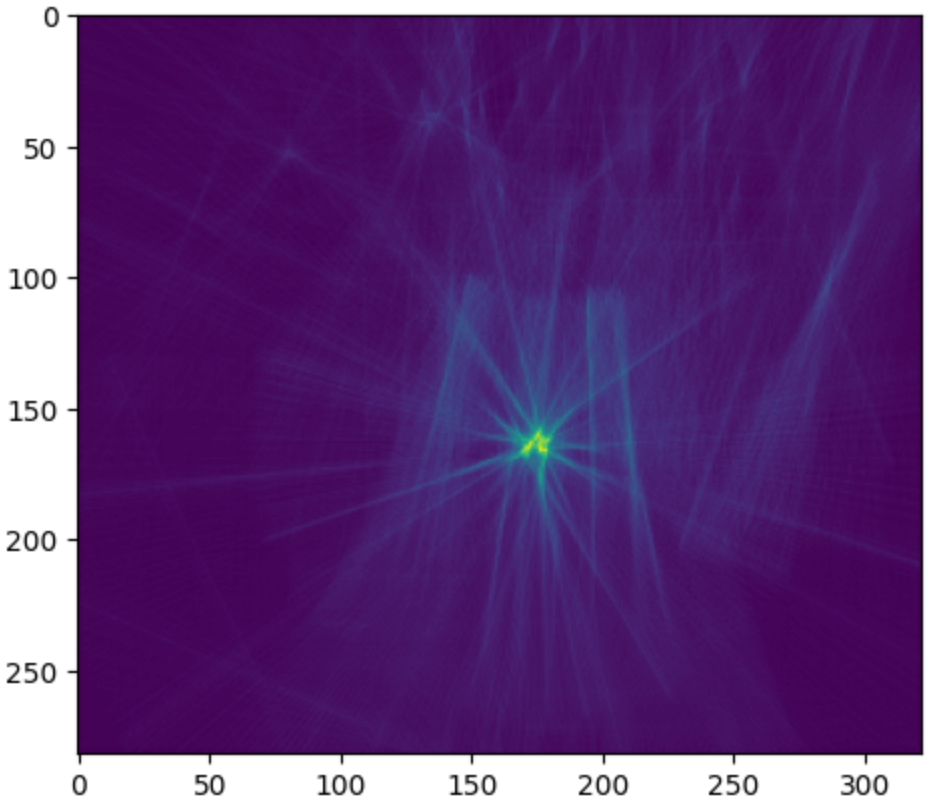
```
# plt.show()
```

Build Accumulation matrix with Hough Transform

In [ ]:
```
def buildAccumulativeMatrix(bin_matrix, grad_matrix):
    acc_matrix = np.zeros((bin_matrix.shape[0], bin_matrix.shape[1]))
    lineSize = 150

    for x in range(bin_matrix.shape[0]):
        for y in range(bin_matrix.shape[1]):
            if bin_matrix[x][y] == 1:
                for i in range( -1 * lineSize, lineSize):
                    vote_x = x + round( i * np.cos(grad_matrix[x][y]))
                    vote_y = y + round( i * np.sin(grad_matrix[x][y]))

                    if vote_x < 0 or vote_y < 0:
                        continue

                    if vote_x >= bin_matrix.shape[0] or vote_y >= bin_matrix.shape[
                        continue

                    acc_matrix[vote_x][vote_y] += 1

    return acc_matrix
```

In [ ]:
```
idx = 1
binary_edge_matrix = filterEdgeMap(edge_images[idx], 42, 1)
accumulative_matrix = buildAccumulativeMatrix(binary_edge_matrix, gradient_matrices

indx = np.unravel_index(np.argmax(accumulative_matrix, axis=None), accumulative_mat
plt.imshow(accumulative_matrix)
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x137a77a7680>
```

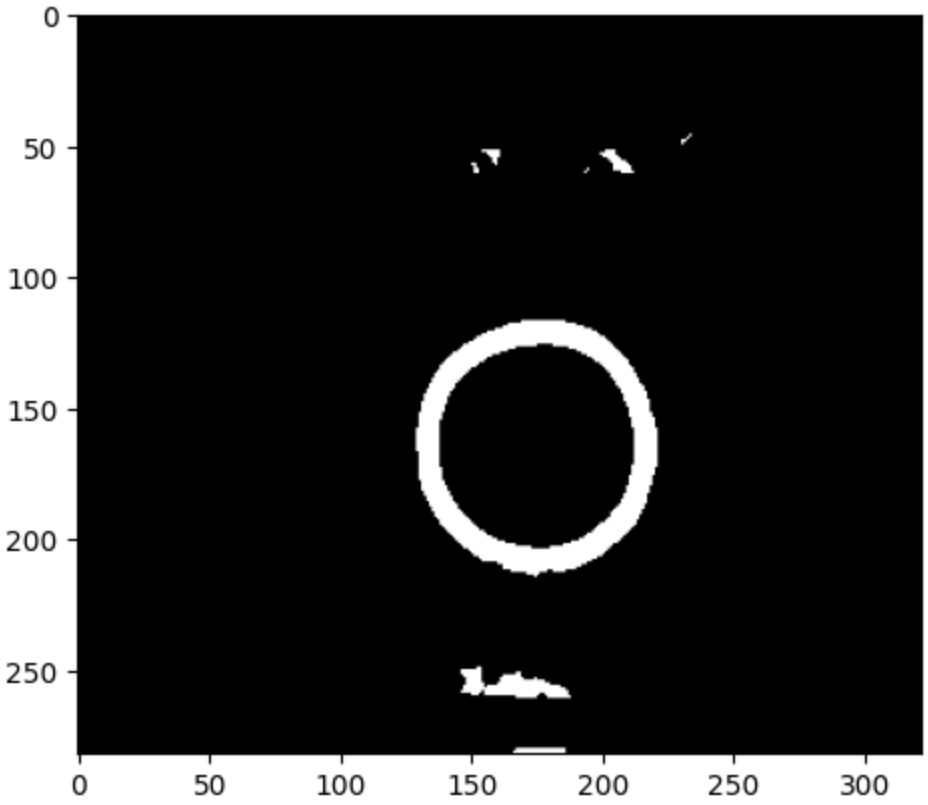


In [ ]:
```
def centerBox(x_value, y_value, percision):
    x_values = []
    y_values = []

    for x in range(x_value - percision, x_value + percision):
        x_values.append(x)

    for y in range(y_value - percision, y_value + percision):
        y_values.append(y)


    return x_values, y_values

def removeEdgesNotApartOfCenters(bin_matrix, grad_matrix, center_x_values, center_y
    lineSize = 125

    circle_matrix = np.zeros((bin_matrix.shape[0], bin_matrix.shape[1]))

    for x in range(bin_matrix.shape[0]):
        for y in range(bin_matrix.shape[1]):
            if bin_matrix[x][y] == 1:
                belongsToCircle = False

                for i in range( (-1) * lineSize, lineSize):
                    if i % 5 == 0:
                        vote_x = x + round( i * np.cos(grad_matrix[x][y]))
                        vote_y = y + round( i * np.sin(grad_matrix[x][y]))

                        if vote_x < 0 or vote_y < 0:
                            continue
```

```
                        if vote_x >= bin_matrix.shape[0] or vote_y >= bin_matrix.sh
                            continue

                        if vote_x in center_x_values and vote_y in center_y_values:
                            belongsToCircle = True
                            break

                if belongsToCircle:
                    circle_matrix[x][y] = 1


    return circle_matrix
```

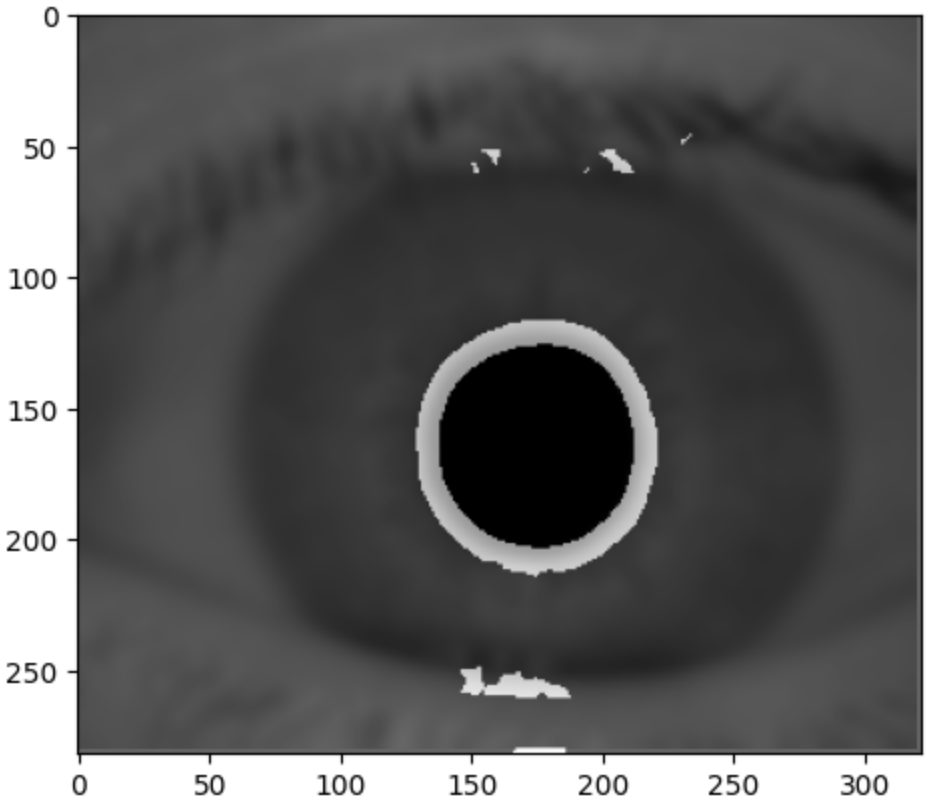
In [ ]:
```
center_x_values, center_y_values = centerBox(indx[0], indx[1], 10)
circles_matrix = removeEdgesNotApartOfCenters(binary_edge_matrix, gradient_matrices

plt.imshow(circles_matrix, cmap='gray')
```

Out[ ]: <matplotlib.image.AxesImage at 0x137ad745d60>



In [ ]:
```
plt.imshow(smooth_eye_imgs[idx], cmap='gray')
plt.imshow(circles_matrix, cmap='gray', alpha= 0.6)
```

Out[ ]: <matplotlib.image.AxesImage at 0x137a3907c20>



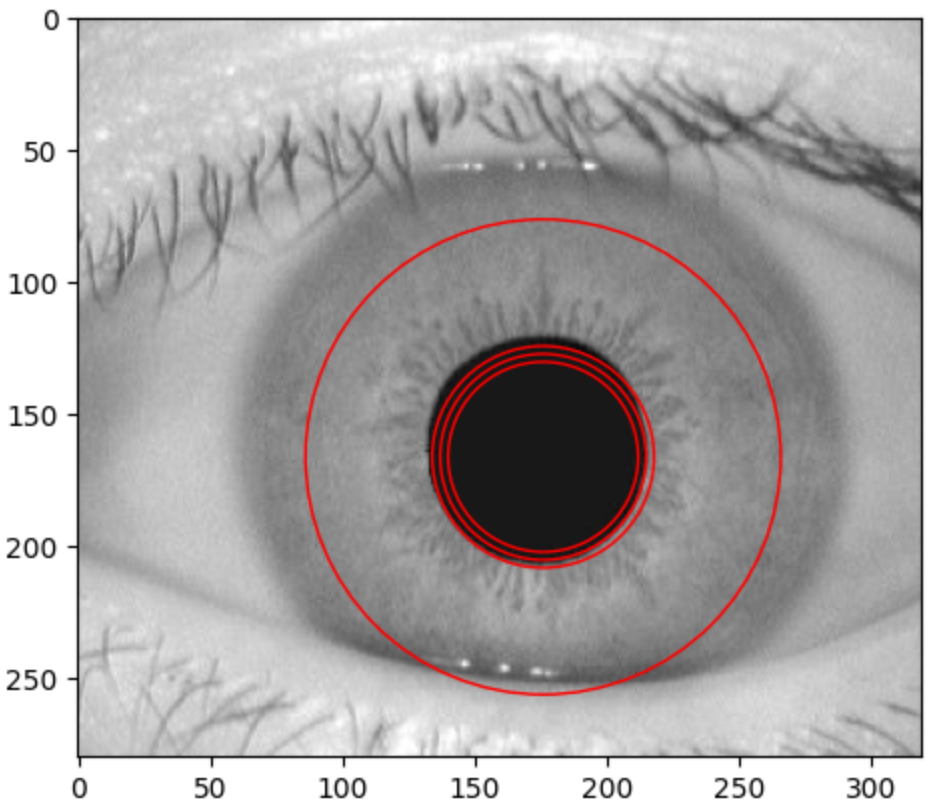
In [ ]:
```
def wrapIntoAMatrix(circles_matrix, center_x, center_y):
    percision = 100
    maxRadius = 100
    circle_array = np.zeros((maxRadius, percision))
    for r in range(0, maxRadius):
        if r % 3 == 0:
            for i in range(0, percision):
                circle_array[r][i] = circles_matrix[center_x + round(r * np.cos( (n
                    center_y + round(r * np.sin( (np.pi * i) / percision))]

                circle_array[r][i] += circles_matrix[center_x + round((r + 1) * np.
                    center_y + round( (r + 1) * np.sin( (np.pi * i) / percision))]

                circle_array[r][i] += circles_matrix[center_x + round((r + 2) * np.
                    center_y + round( (r + 2) * np.sin( (np.pi * i) / percision))]

    return circle_array

flattenedCircle = wrapIntoAMatrix(circles_matrix, indx[0], indx[1])

i = 0
radius = []
for r in flattenedCircle:
    total = np.sum(r)
    if total > 6:
        radius.append((i, total))
        # print(f'Radius {i} {np.sum(r)}')
    i += 1
```

```
print(len(radius))
print(radius)
```

```
6
[(36, 153.0), (39, 264.0), (42, 297.0), (45, 152.0), (48, 44.0), (90, 11.0)]
```

In [ ]:

```
fig, axes = plt.subplots()

circles = []
last_r = radius[0][0]
max_intensity = 0
for r in radius:
    if r[1] > 5 and r[1] > max_intensity:
        circles.append(plt.Circle((indx[1], indx[0]), r[0], color='red', fill=False
        max_intensity = r[1]
        last_r = r[0]
    elif r[0] - last_r > 20:
        circles.append(plt.Circle((indx[1], indx[0]), r[0], color='red', fill=False
        max_intensity = r[1]
        last_r = r[0]

axes.imshow(images[idx], cmap='gray')
for c in circles:
    axes.add_patch(c)

plt.show()
# fig.savefig('circles.png')
```



Copied Code

From: https://github.com/banderlog/daugman Writer: banderlog Date last commit: 2021

This code section directly below is an implementation of the daugman method in python. It was found on github at the mentioned address.

I modified the gaussianBlur to be (3, 3) instead of (1, 5)

In [ ]:
```
def daugman(gray_img: np.ndarray, center: Tuple[int, int],
            start_r: int, end_r: int, step: int = 1) -> Tuple[float, int]:
    """ The function will calculate pixel intensities for the circles
        in the ``range(start_r, end_r, step)`` for a given ``center``,
        and find a circle that precedes the biggest intensity drop

        :param gray_img: grayscale picture
        :param center:  center coordinates ``(x, y)``
        :param start_r: bottom value for iris radius in pixels
        :param end_r: top value for iris radius in pixels
        :param step: step value for iris radii range in pixels

        .. attention::
            Input grayscale image should be a square, not a rectangle

        :return: intensity_value, radius
    """
    x, y = center
    intensities = []
    mask = np.zeros_like(gray_img)


    # for every radius in range
    radii = list(range(start_r, end_r, step))  # type: List[int]
    for r in radii:
        # draw circle on mask
        cv2.circle(mask, center, r, 255, 1)
        # get pixel from original image, it is faster than np or cv2
        diff = gray_img & mask
        # normalize, np.add.reduce faster than .sum()
        #            diff[diff > 0] faster than .flatten()
        intensities.append(np.add.reduce(diff[diff > 0]) / (2 * math.pi * r))
        # refresh mask
        mask.fill(0)

    # calculate delta of radius intensitiveness
    #     mypy does not tolerate var type reload
    intensities_np = np.array(intensities, dtype=np.float32)
    del intensities

    # circles intensity differences, x5 faster than np.diff()
    intensities_np = intensities_np[:-1] - intensities_np[1:]
    # aply gaussian filter
    #     GaussianBlur() faster than filter2D() with custom kernel
    # original kernel:
    # > The Gaussian filter in our case is designedin MATLAB and
    # > is a 1 by 5 (rows by columns) vector with intensity values
    # > given by vector A = [0.0003 0.1065 0.7866 0.1065 0.0003]
```

```
    intensities_np = abs(cv2.GaussianBlur(intensities_np, (5, 5), 0))
    # get maximum value
    idx = np.argmax(intensities_np)  # type: int

    # return intensity value, radius
    return intensities_np[idx], radii[idx]


def find_iris(gray: np.ndarray, *,
              daugman_start: int, daugman_end: int,
              daugman_step: int = 1, points_step: int = 1,) -> Tuple[Tuple[int, int
    """ The function will apply :func:`daugman` on every pixel in the calculated im
        Basically, we are calculating where lies set of valid circle centers.
        It is assumed that iris center lies within central 1/3 of the image.

        :param gray: graysacale **square** image
        :param points_step: it will run daugman for each ``points_step``th point.
                            It has linear correlation with overall iris search spee
        :param daugman_start: bottom value for iris radius in pixels for :func:``da
        :param daugman_end: top value for iris radius in pixels for :func:``daugman
        :param daugman_step: step value for iris radii range in pixels for :func:``
                             It has linear correlation with overall iris search spe

        :return: radius with biggest intensiveness delta on image as ``((xc, yc), r
    """
    h, w = gray.shape
    if h != w:
        print('Your image is not a square!')

    # reduce step for better accuracy
    # we will look only on dots within central 1/3 of image
    single_axis_range = range(int(h / 3), h - int(h / 3), points_step)
    all_points = itertools.product(single_axis_range, single_axis_range)

    intensity_values = []
    coords = []  # List[Tuple[Tuple(int, int), int]]

    for point in all_points:
        val, r = daugman(gray, point, daugman_start, daugman_end, daugman_step)
        intensity_values.append(val)
        coords.append((point, r))

    # return the radius with biggest intensiveness delta on image
    # ((xc, yc), radius)
    # x10 faster than coords[np.argmax(values)]
    best_idx = intensity_values.index(max(intensity_values))
    return coords[best_idx]
```

End Referenced Code

In [ ]:
```
def square_image(img):
    h, w = img.shape
    copy = img.copy()
```

```
    if h != w:
        if h > w:
            diff = h - w

            if diff % 2 == 0:
                return copy[int(diff / 2) : h - int(diff / 2), :]
            else:
                return copy[int(diff / 2) : h - int(diff / 2) - 1, :]
        else:
            diff = w - h

            if diff % 2 == 0:

                return copy[:, int(diff / 2) : w - int(diff / 2 )]
            else:
                return copy[:, int(diff / 2) : w - int(diff / 2 ) - 1]
```

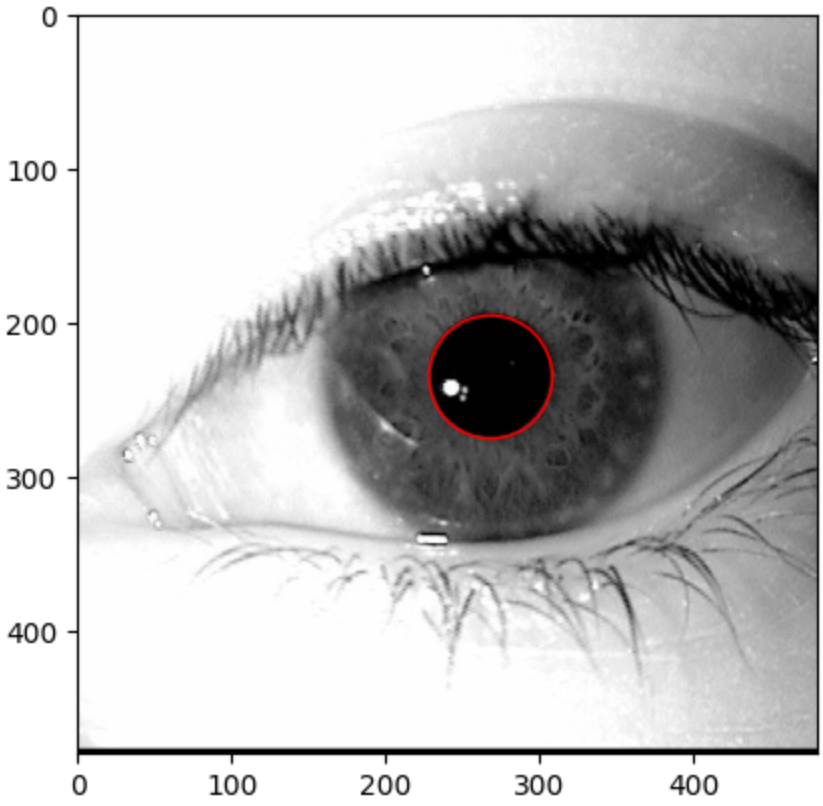
In [ ]:
```
square_images = list(np.asarray(square_image(gray_img)) for gray_img in gray_images
```

In [ ]:
```
answers = list(find_iris(square_img, daugman_start=35, daugman_end=80, daugman_step
```

In [ ]:
```
iris_centers = list(answer[0] for answer in answers)
iris_radii = list(answer[1] for answer in answers)

def plot_daugman_circle(img, center, radius):
    fig, axes = plt.subplots()

    print(center)
    print(radius)
    axes.imshow(img, cmap='gray')
    circle = plt.Circle(center, radius, color='red', fill= False)
    axes.add_patch(circle)
```

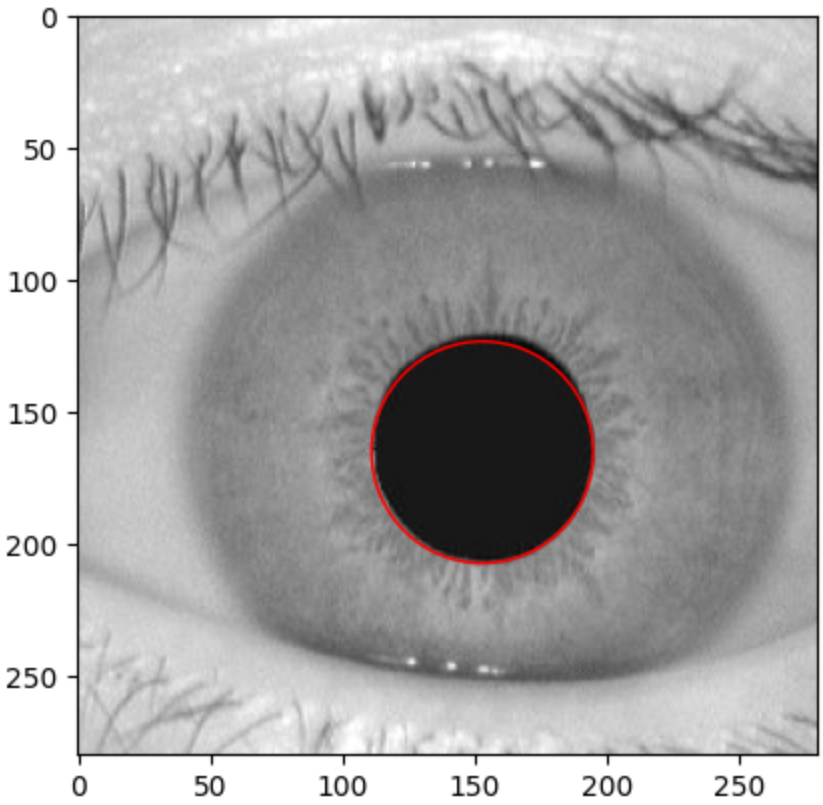
In [ ]:
```
idx = 0
plot_daugman_circle(square_images[idx], iris_centers[idx], iris_radii[idx])
```

```
(268, 235)
40
```



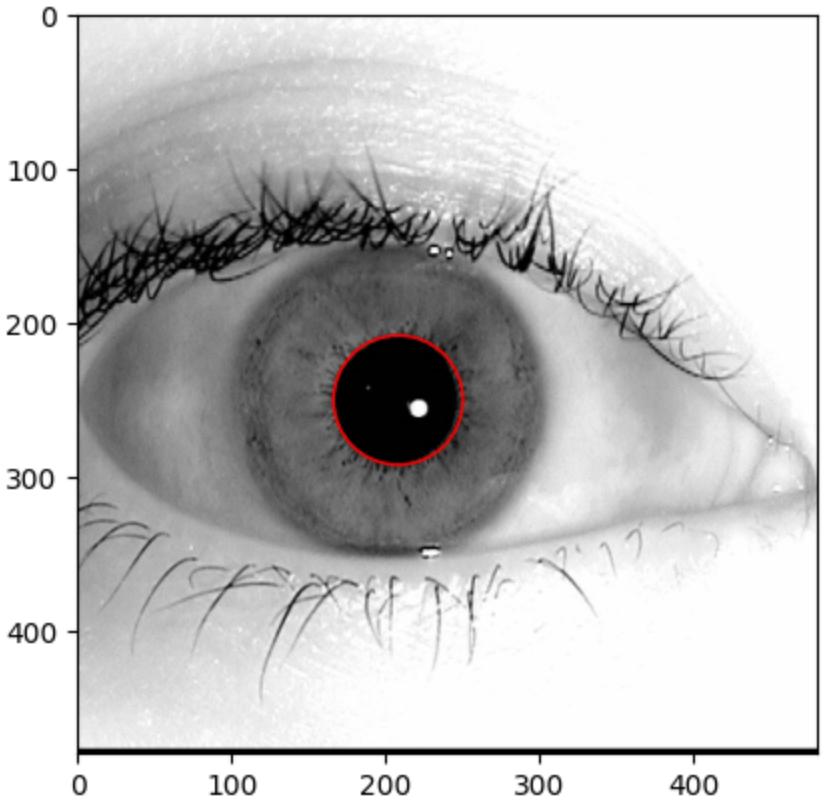
In [ ]:
```
idx = 1
plot_daugman_circle(square_images[idx], iris_centers[idx], iris_radii[idx])
```

```
(153, 165)
42
```



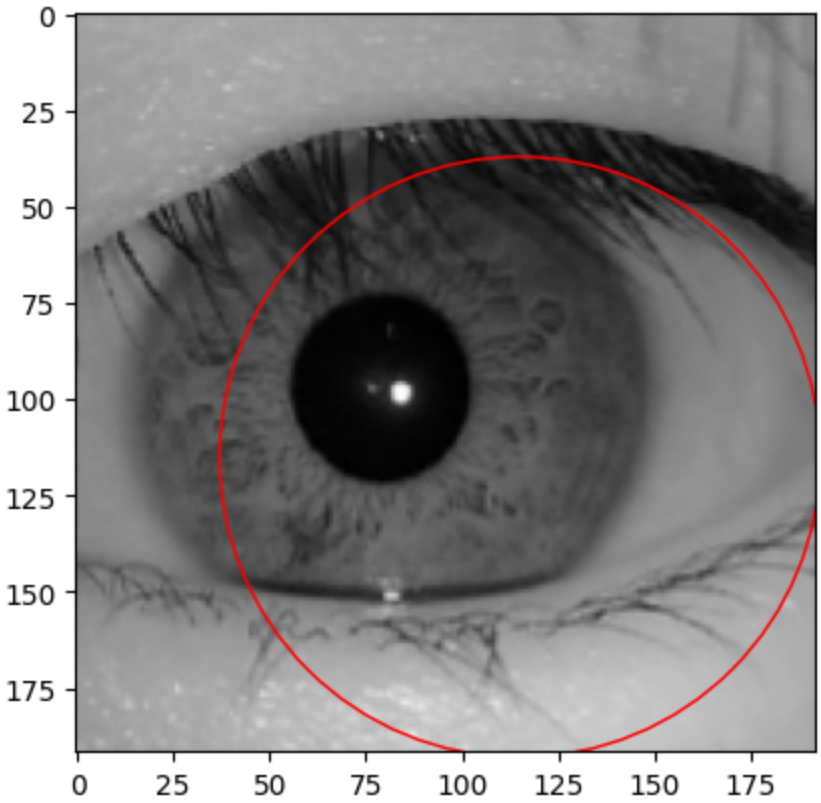
In [ ]:
```
idx = 2
plot_daugman_circle(square_images[idx], iris_centers[idx], iris_radii[idx])
```

```
(208, 250)
42
```



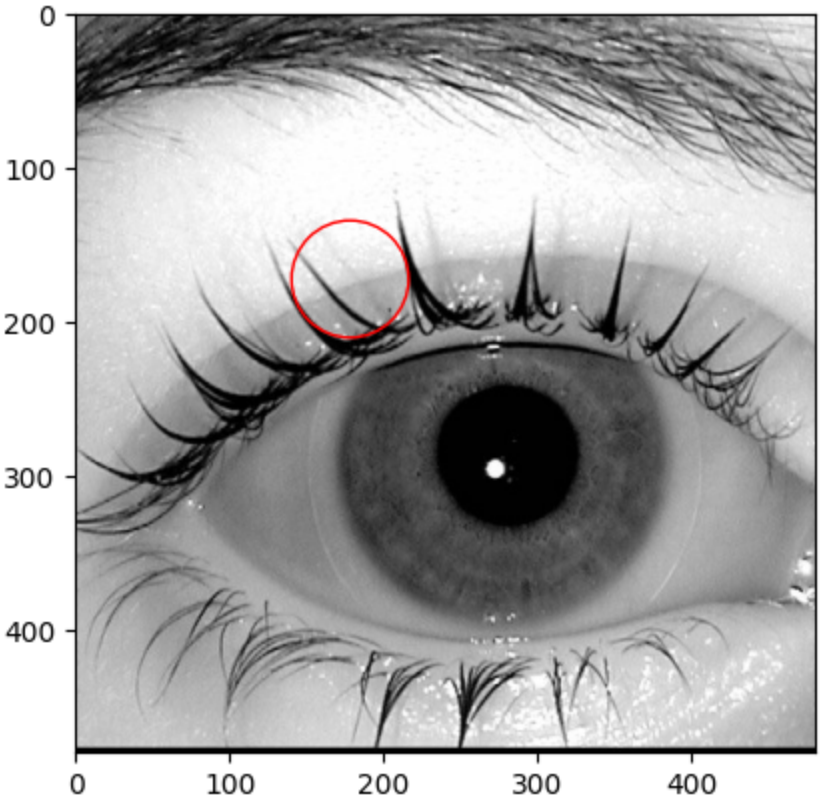
In [ ]:
```
idx = 3
plot_daugman_circle(square_images[idx], iris_centers[idx], iris_radii[idx])
```

```
(115, 115)
78
```



In [ ]:
```
idx = 4
plot_daugman_circle(square_images[idx], iris_centers[idx], iris_radii[idx])
```

```
(178, 172)
38
```



In [ ]:
```
idx = 5
plot_daugman_circle(square_images[idx], iris_centers[idx], iris_radii[idx])
```

```
(319, 289)
56
```