CSCI E-25

Working with Image Data

Steve Elston

Introduction

This lesson will familarize you with the basic concepts of working with image data and some statistical properties of images. Some key points are of this lesson are:

1. Discrete pixel structure of digital images.
2. Representation of color and gray scale images.
3. Intensity distribution of image data.
4. Equalizing intensity distributions and improving contrast.
5. Resizing images.

Some test text here.

> **Scikit-Learn Image:** In this lesson we will be using the Scikit-Learn Image package. This package provides many commonly used computer vision algorithms using a consistent Python API. This package follows the API conventions of Scikit-Learn, enabling the application of a rich library of machine learning algorithms. There is excellent documentation available for the Scikit-Learn Image package. You may wish to start by reading through the User Guide. Examples of what you can do with Scikit-Learn Image package can be seen in the Gallery.

> **Installation:** To run the code in this notebook you will need to install Scikit-Learn Image. Follow the installation directions for your operating system.

> **Scikit-Learn Image and Numpy:** Like all Scikit packages this one is built on Numpy. If you are not very familiar with Numpy you can find Tutorials here. A tutorial on using Numpy with Scikit-learn image data objects can be found here.

Loading Data

To get started with this lesson, execute the code in the cell below to import the packages you will need.

In [ ]:
```
%pip install scikit-image
```

```
Requirement already satisfied: scikit-image in /Library/Frameworks/Python.f
ramework/Versions/3.11/lib/python3.11/site-packages (0.19.3)
Requirement already satisfied: numpy>=1.17.0 in /Library/Frameworks/Python.
framework/Versions/3.11/lib/python3.11/site-packages (from scikit-image)
(1.24.2)
Requirement already satisfied: scipy>=1.4.1 in /Library/Frameworks/Python.f
ramework/Versions/3.11/lib/python3.11/site-packages (from scikit-image) (1.
10.0)
Requirement already satisfied: networkx>=2.2 in /Library/Frameworks/Python.
framework/Versions/3.11/lib/python3.11/site-packages (from scikit-image)
(3.0)
Requirement already satisfied: pillow!=7.1.0,!=7.1.1,!=8.3.0,>=6.1.0 in /Li
brary/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-package
s (from scikit-image) (9.4.0)
Requirement already satisfied: imageio>=2.4.1 in /Library/Frameworks/Pytho
n.framework/Versions/3.11/lib/python3.11/site-packages (from scikit-image)
(2.25.0)
Requirement already satisfied: tifffile>=2019.7.26 in /Library/Frameworks/P
ython.framework/Versions/3.11/lib/python3.11/site-packages (from scikit-ima
ge) (2023.2.3)
Requirement already satisfied: PyWavelets>=1.1.1 in /Library/Frameworks/Pyt
hon.framework/Versions/3.11/lib/python3.11/site-packages (from scikit-imag
e) (1.4.1)
Requirement already satisfied: packaging>=20.0 in /Library/Frameworks/Pytho
n.framework/Versions/3.11/lib/python3.11/site-packages (from scikit-image)
(23.0)
Note: you may need to restart the kernel to use updated packages.
```

In [ ]:
```
%pip install -U numpy scipy scikit-learn
```

```
Requirement already satisfied: numpy in /Library/Frameworks/Python.framewor
k/Versions/3.11/lib/python3.11/site-packages (1.24.2)
Requirement already satisfied: scipy in /Library/Frameworks/Python.framewor
k/Versions/3.11/lib/python3.11/site-packages (1.10.0)
Requirement already satisfied: scikit-learn in /Library/Frameworks/Python.f
ramework/Versions/3.11/lib/python3.11/site-packages (1.2.1)
Requirement already satisfied: joblib>=1.1.1 in /Library/Frameworks/Python.
framework/Versions/3.11/lib/python3.11/site-packages (from scikit-learn)
(1.2.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in /Library/Frameworks/
Python.framework/Versions/3.11/lib/python3.11/site-packages (from scikit-le
arn) (3.1.0)
Note: you may need to restart the kernel to use updated packages.
```

In [ ]:
```
%pip install matplotlib
```

```
Requirement already satisfied: matplotlib in /Library/Frameworks/Python.fra
mework/Versions/3.11/lib/python3.11/site-packages (3.6.3)
Requirement already satisfied: contourpy>=1.0.1 in /Library/Frameworks/Pyth
on.framework/Versions/3.11/lib/python3.11/site-packages (from matplotlib)
(1.0.7)
Requirement already satisfied: cycler>=0.10 in /Library/Frameworks/Python.f
ramework/Versions/3.11/lib/python3.11/site-packages (from matplotlib) (0.1
1.0)
Requirement already satisfied: fonttools>=4.22.0 in /Library/Frameworks/Pyt
hon.framework/Versions/3.11/lib/python3.11/site-packages (from matplotlib)
(4.38.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /Library/Frameworks/Pyt
hon.framework/Versions/3.11/lib/python3.11/site-packages (from matplotlib)
(1.4.4)
Requirement already satisfied: numpy>=1.19 in /Library/Frameworks/Python.fr
amework/Versions/3.11/lib/python3.11/site-packages (from matplotlib) (1.24.
2)
Requirement already satisfied: packaging>=20.0 in /Library/Frameworks/Pytho
n.framework/Versions/3.11/lib/python3.11/site-packages (from matplotlib) (2
3.0)
Requirement already satisfied: pillow>=6.2.0 in /Library/Frameworks/Python.
framework/Versions/3.11/lib/python3.11/site-packages (from matplotlib) (9.
4.0)
Requirement already satisfied: pyparsing>=2.2.1 in /Library/Frameworks/Pyth
on.framework/Versions/3.11/lib/python3.11/site-packages (from matplotlib)
(3.0.9)
Requirement already satisfied: python-dateutil>=2.7 in /Library/Frameworks/
Python.framework/Versions/3.11/lib/python3.11/site-packages (from matplotli
b) (2.8.2)
Requirement already satisfied: six>=1.5 in /Library/Frameworks/Python.frame
work/Versions/3.11/lib/python3.11/site-packages (from python-dateutil>=2.7-
>matplotlib) (1.16.0)
Note: you may need to restart the kernel to use updated packages.
```

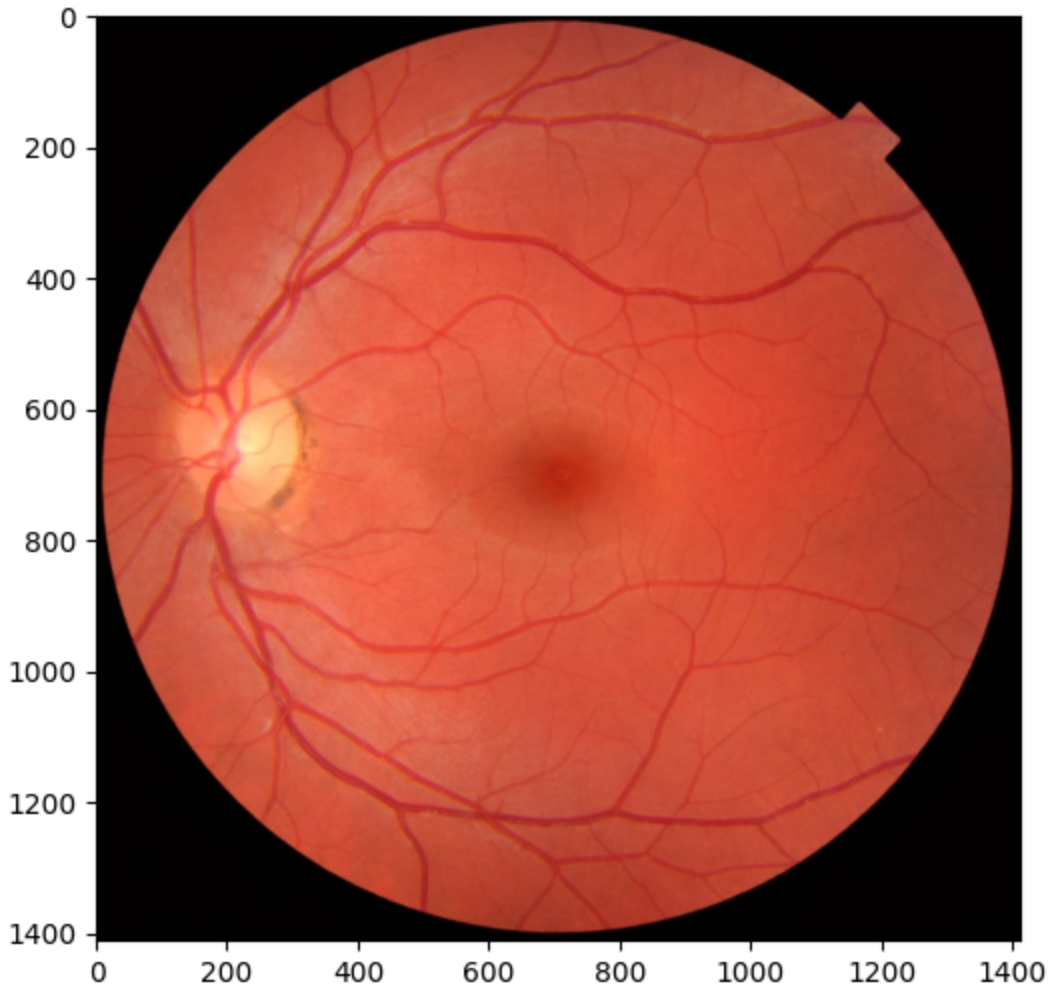
In [ ]:
```
import skimage
from skimage import data
from skimage.filters.rank import equalize, threshold
import skimage.filters as skfilters
from skimage import exposure
from skimage.morphology import disk, square
from skimage.color import rgb2gray, rgb2ycbcr, ycbcr2rgb, rgb2xyz, xyz2rgb,
from skimage.measure import block_reduce
from skimage.transform import resize
import numpy as np
import numpy.random as nr
from PIL import Image
from scipy import signal
from sklearn.preprocessing import MinMaxScaler
import math
import matplotlib.pyplot as plt
%matplotlib inline
```

Structure of a Color Image Object

The code in the cell below loads a color image of a human retina, prints the data types and dimensions of the image object, and displays the image. The image is displayed by matplotlib.pyplot.imshow. Execute the code and examine the result.

In [ ]:
```
retina_image = data.retina()
print('The image object is ' + str(type(retina_image)))
print('The pixel values are of type ' + str(type(retina_image[0,0,0])))
print('Shape of image object = ' + str(retina_image.shape))
fig, ax = plt.subplots( figsize=(6, 6))
_=ax.imshow(retina_image)
```

```
The image object is <class 'numpy.ndarray'>
The pixel values are of type <class 'numpy.uint8'>
Shape of image object = (1411, 1411, 3)
```

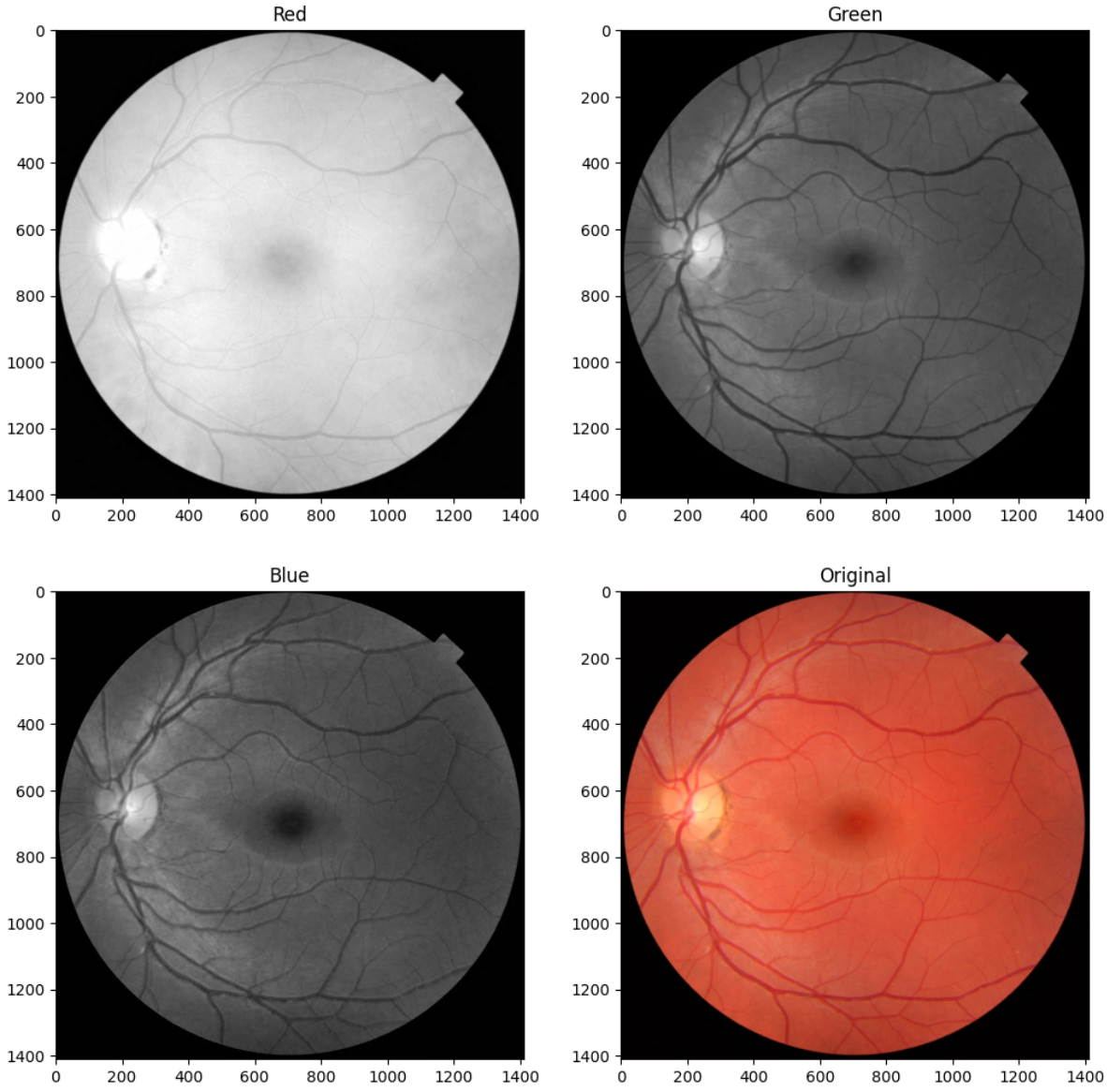


The image object has 3-dimensions, the two spatial dimensions and the 3 color channels. Examine this image noticing the wide variation in color and intensity. Notice also that the illumination of the retina does not appear uniform, resulting in a bright spot on the left and a darker region on the right.

> **Exercise 1-1:** Complete the code for the function in the cell below to display the 3 color channels of the image and the original image in a 2x2 array using the matplotlib.pyplot.imshow function. The color channels are

> in red, green, blue order and should be displayed as gray scale using the `cmap=plt.get_cmap('gray')` argument. Your function should label the channels and the original image. Execute your function and examine the results.

In [ ]:
```
def plot_3_color_channels(img):
    '''Function plots the three color channels of the image along with the c
    fig, ax = plt.subplots(2, 2, figsize=(12, 12))
    ax = ax.flatten()

    ## Complete the code below

    # Graphs titles
    titles = ['Red', 'Green', 'Blue', 'Original']

    # Color channels in grayscale
    for i in range(3):
        ax[i].set_title(titles[i])
        ax[i].imshow(img[:, :, i], cmap = 'gray')

    # Color image
    ax[3].set_title(titles[3])
    ax[3].imshow(img)

# Plot color channels in greyscale and original color image
plot_3_color_channels(retina_image)
```



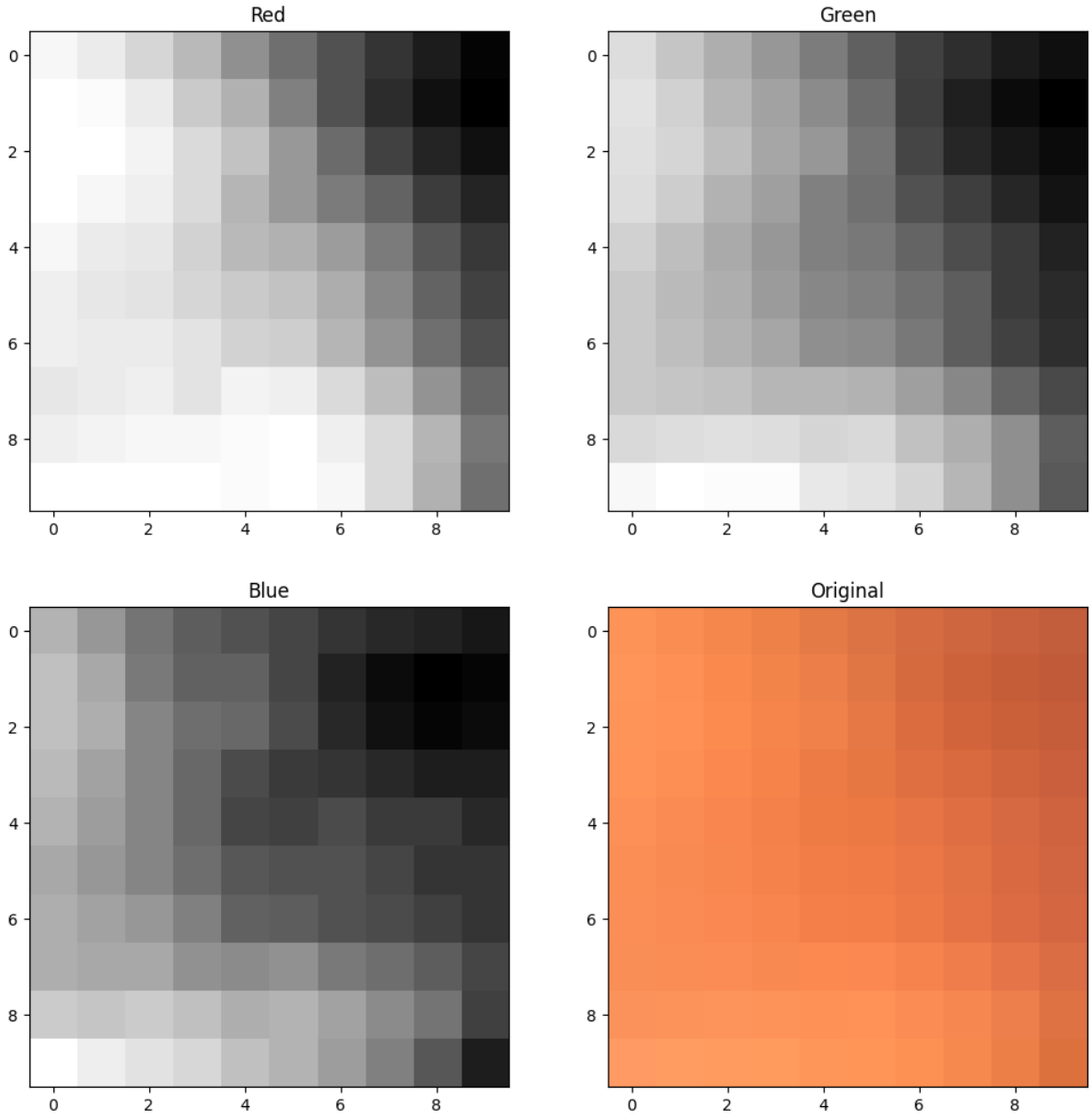
> Examine the intensity (brightness) of the color channels and answer these questions:
>
> 1. Which channel has the greatest intensity, and does this make sense given the image?
> 2. Is it likely that the saturation of the red channel arises as an artifact of the illumination spot on the left of the retina image?
>
> **End of exercise.**

> **Answers:**
>
> 1. The red channel. It does make sense because the redish color of the original image has an important component of red.

> 2. Illuminstion certanly affects the saturation. The color is lighter where there is intense light and darker where there is dimmer light.

When working with digital images it is always important to keep in mind the discrete nature of the samples. To demonstrate the discrete nature of a digital image you can visualize a 100 pixel, or 10×10, sample from the larger image by executing the code in the cell below.

In [ ]:
```
plot_3_color_channels(retina_image[600:610,300:310,:])
```

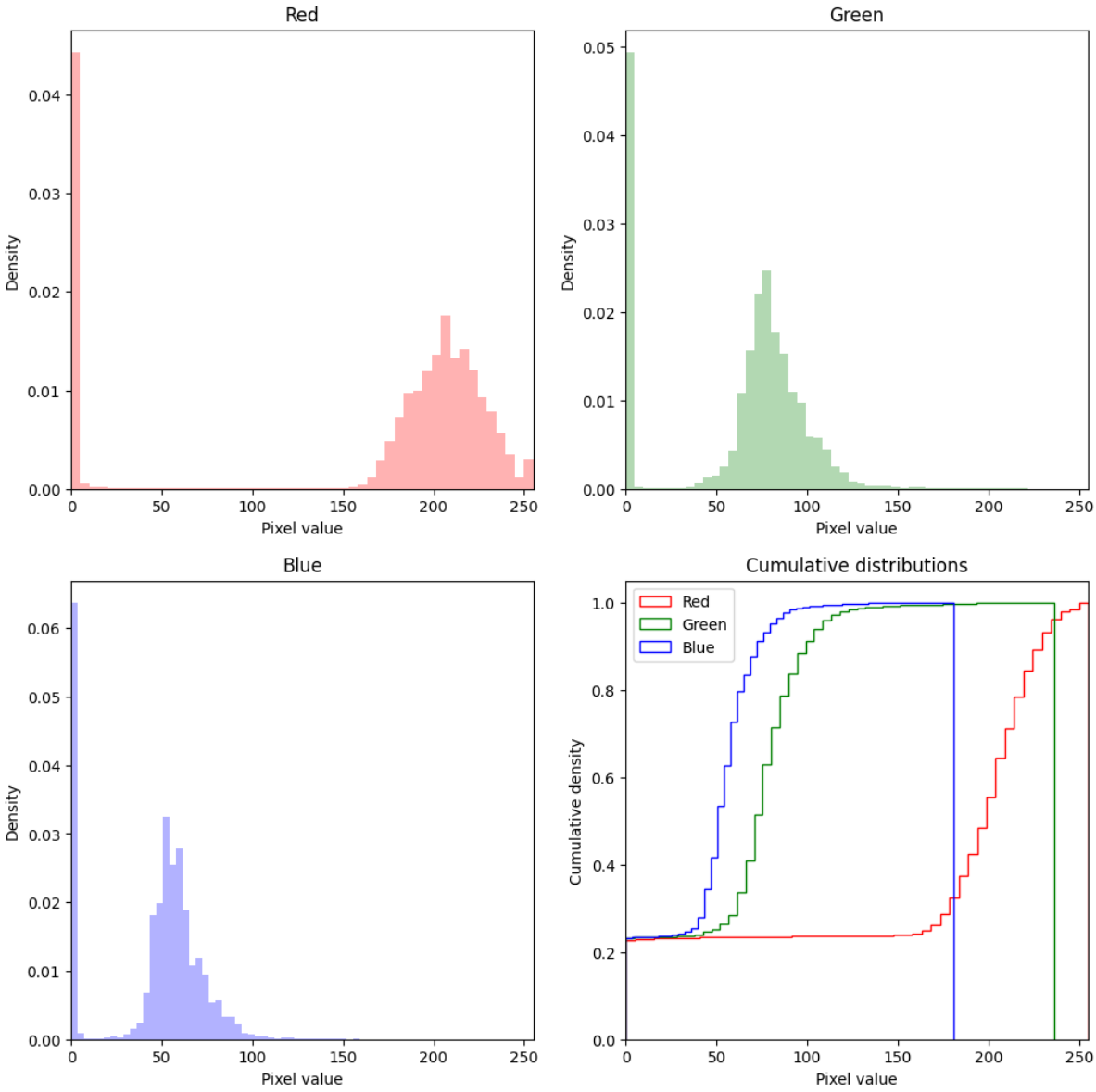


Notice the discrete nature in each of the three color channels and the color image. The sum of these discrete color-channel pixel intensities yields the color image.

Statistical Properties of an Image

The next question is, what is the distribution of pixel intensities in the 3 color channels of the image? Histograms and cumulative density functions are used to analyze these distributions. The code in the cell below plots the histograms of the 3 color channels along with their cumulative distributions. Execute this code and examine the results.

In [ ]:
```
def plot_image_distributions(img, xlim=(0,255)):
    '''Function plots histograms of the three color channels of the image al
    with the cumulative distributions'''
    fig, ax = plt.subplots(2,2, figsize=(12, 12))
    ax = ax.flatten()
    titles=['Red','Green','Blue']
    for i in range(3):
        ax[i].hist(img[:,:,i].flatten(), bins=50, density=True, color=titles
        ax[i].set_title(titles[i])
        ax[i].set_xlabel('Pixel value')
        ax[i].set_ylabel('Density')
        ax[i].set_xlim(xlim)
        ax[3].hist(img[:,:,i].flatten(), bins=50, density=True, cumulative=T
        ax[3].set_xlim(xlim)
    ax[3].set_title('Cumulative distributions')
    ax[3].set_xlabel('Pixel value')
    ax[3].set_ylabel('Cumulative density')
    plt.legend(loc='upper left')

plot_image_distributions(retina_image)
```



There are several properties of the distribution of the pixel values which are important:

1. The distribution of the intensity for of the red channel has clearly higher values than the other channels.
2. A significant fraction of pixel values have 0 intensity for all 3 color channels. These pixels are primarily black background around the retina, but may also represent the dark pupil spot in the center of the retina.
3. A few red channel pixels have the maximum value of 255. The red intensity of these pixels is said to be **saturated**.

Improving Contrast

Our next question to address is what is the ideal distribution of the intensity values of an image? A useful, and obviously answer, is that we want the pixel values over the full range of possible values. For unsigned integer values, {0,255}. Further, the distribution

of pixel values should be uniform. For the $n = 256$ unsigned integer values the **probability mass function**, or **PMF**, of the ith value is:

$$p(i) = \frac{1}{n}$$

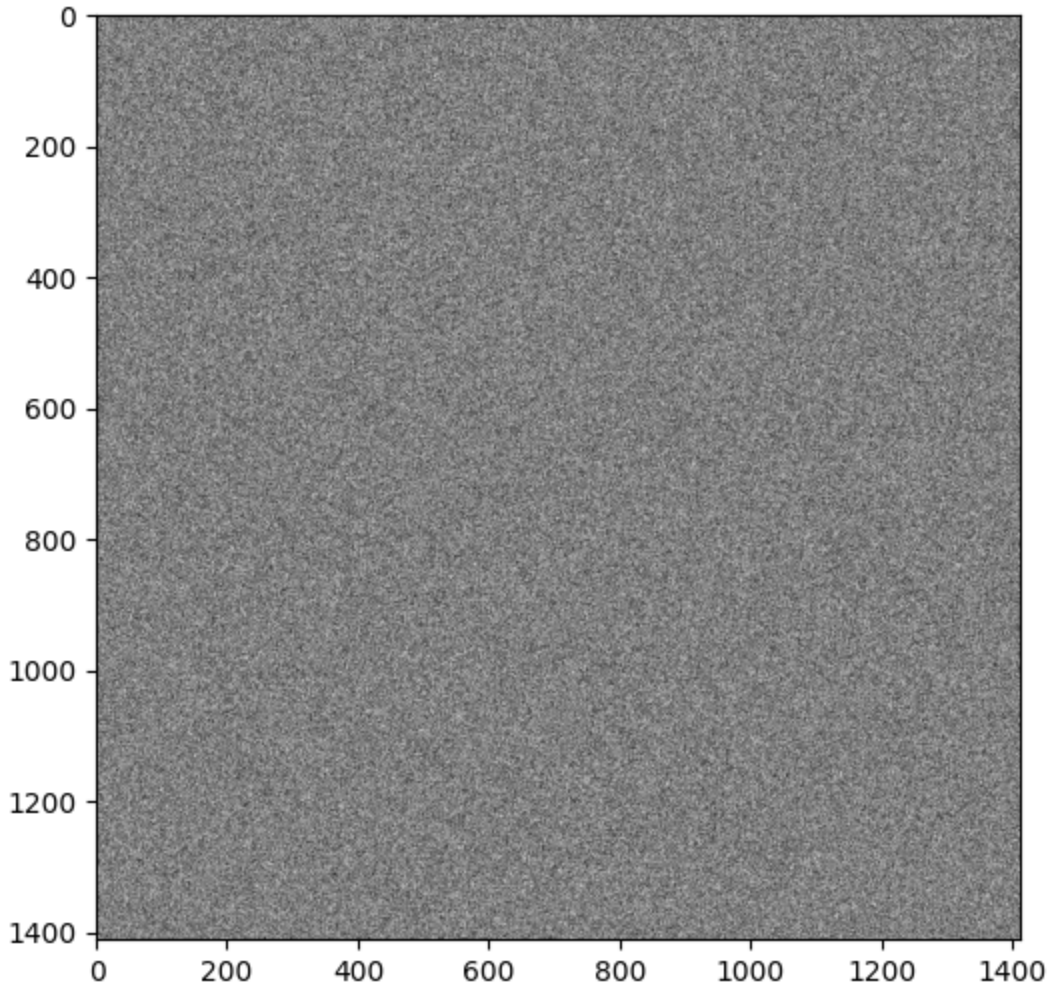
And the **cumulative density function**, or **CDF**, of the uniform distribution at the ith value is, x_i:

$$CDF(i) = \sum_{i=0}^{n-1} \frac{1}{x_i}$$

We can visualize an example of a gray-scale image of unsigned integers on the range {0,255} with random uniform distributed pixel values. The code in the cell below forms a gray-scale image randomly sampled uniform distributed pixel values and displays the result.

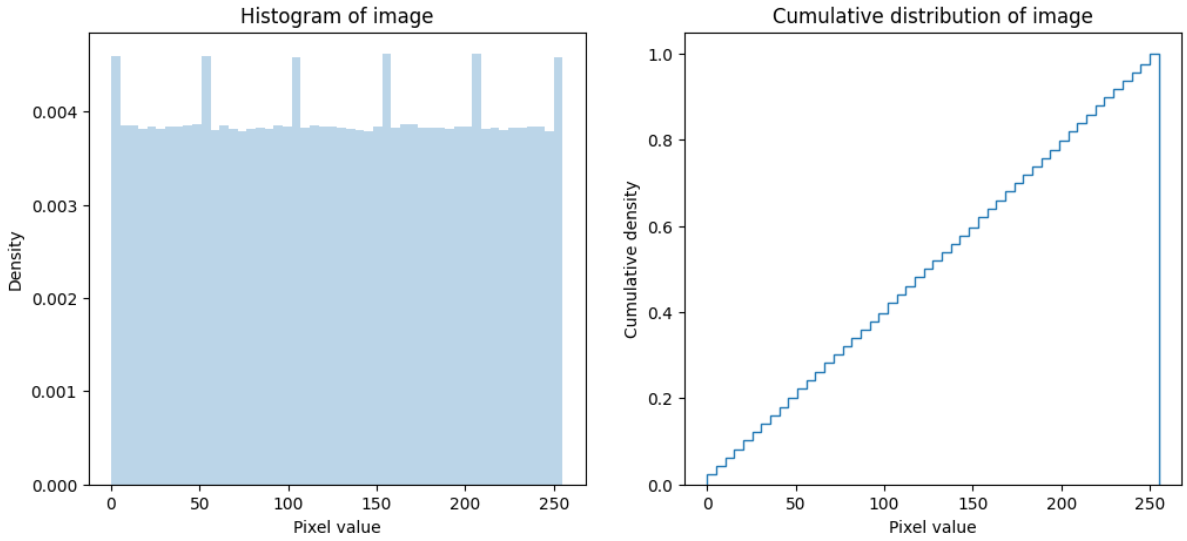
In [ ]:
```
random_image = np.multiply((nr.uniform(low=0.0, high=255.0, size=retina_imag
random_image = random_image.astype(np.uint8)
print(random_image.shape)
fig, ax = plt.subplots( figsize=(6, 6))
_=ax.imshow(random_image, cmap=plt.get_cmap('gray'))
```

```
(1411, 1411)
```



To view the distribution of the pixel values of this image execute the code in the cell below.

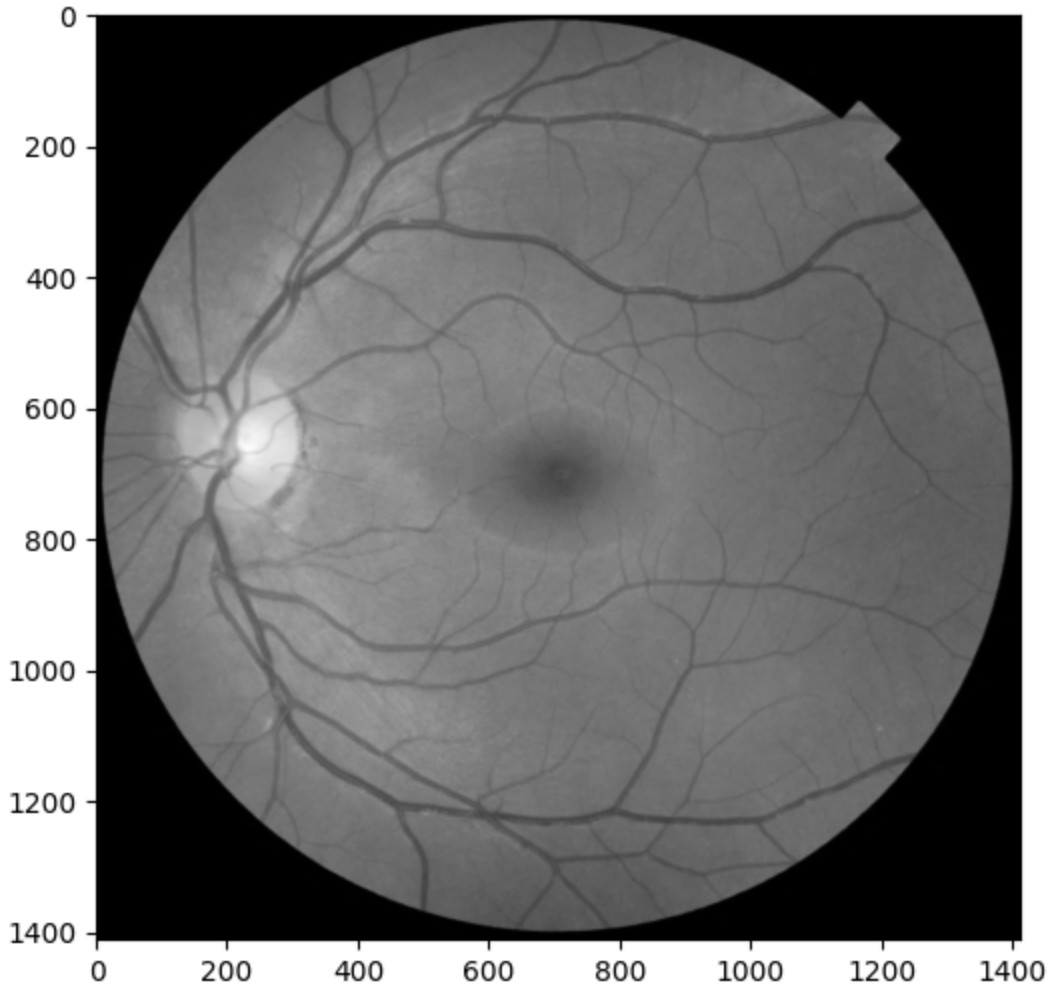
In [ ]:
```
def plot_gray_scale_distribution(img):
    '''Function plots histograms a gray scale image along
    with the cumulative distribution'''
    fig, ax = plt.subplots(1,2, figsize=(12, 5))
    ax[0].hist(img.flatten(), bins=50, density=True, alpha=0.3)
    ax[0].set_title('Histogram of image')
    ax[0].set_xlabel('Pixel value')
    ax[0].set_ylabel('Density')
    ax[1].hist(img.flatten(), bins=50, density=True, cumulative=True, histty
    ax[1].set_title('Cumulative distribution of image')
    ax[1].set_xlabel('Pixel value')
    ax[1].set_ylabel('Cumulative density')
    plt.show()

plot_gray_scale_distribution(random_image)
```

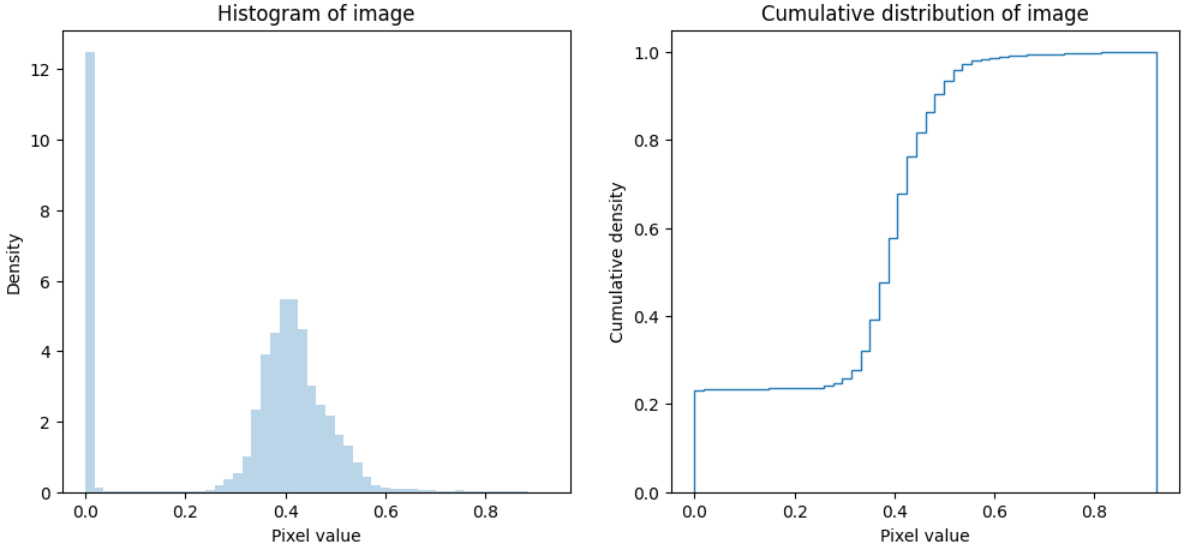


> **Exercise 1-2:** To compare the pixel value distribution of the retina image to the ideal values do the following:
>
> 1. Create a gray-scale image object named `retina_gray_scale` using the skimage.color.rgb2gray function.
> 2. Print the dimensions of the image object.
> 3. Display the gray-scale image. Make sure the image is large enough to see the details.
> 4. Plot the distribution of the pixel values of the gray-scale image.

In [ ]:
```
def plot_grayscale(img):
    fig, ax = plt.subplots( figsize=(6, 6))
    _=ax.imshow(img, cmap=plt.get_cmap('gray'))
    plt.show()

## Put you code below

# 1. Create a gray-scale image object named `retina_gray_scale` using the sk
retina_gray_scale = rgb2gray(retina_image)

# 2. Print the dimensions of the image object.
print('Shape of image object = ' + str(retina_gray_scale.shape))

# 3. Display the gray-scale image. Make sure the image is large enough to se
plot_grayscale(retina_gray_scale)

# 4. Plot the distribution of the pixel values of the gray-scale image.
plot_gray_scale_distribution(retina_gray_scale)
```

```
Shape of image object = (1411, 1411)
```





In [ ]:
```
retina_image.dtype
```

Out[ ]:
```
dtype('uint8')
```

> Examine the distribution plots and answer these questions:

> 1. How would you describe these results with respect to the ideal distribution?
> 2. How do you think the range of pixel values limit the contrast of the image?
>
> **End of exercise.**

> **Answer:**
>
> 1. The distribution is normal. This way, it is more precise to show the different tones between the minimum grey value and the maximum grey value.
>
> 2. The range of pixel values limits the contrast of the image, where we only have 256 possibilities compared with a color image where we have three different numbers (each of them with a 256 pixel value range).

Histogram Equalization

Contrast of an image is range between the minimum and maximum pixel values of an image. The larger the range of values the more distinctive the differences in the image will be. To improve the contrast in an image we need to **equalize** the pixel values over the maximum of the range. The goal is to find a transformation that stretches the pixel values into a uniform distribution. This process is know as **histogram equalization**.

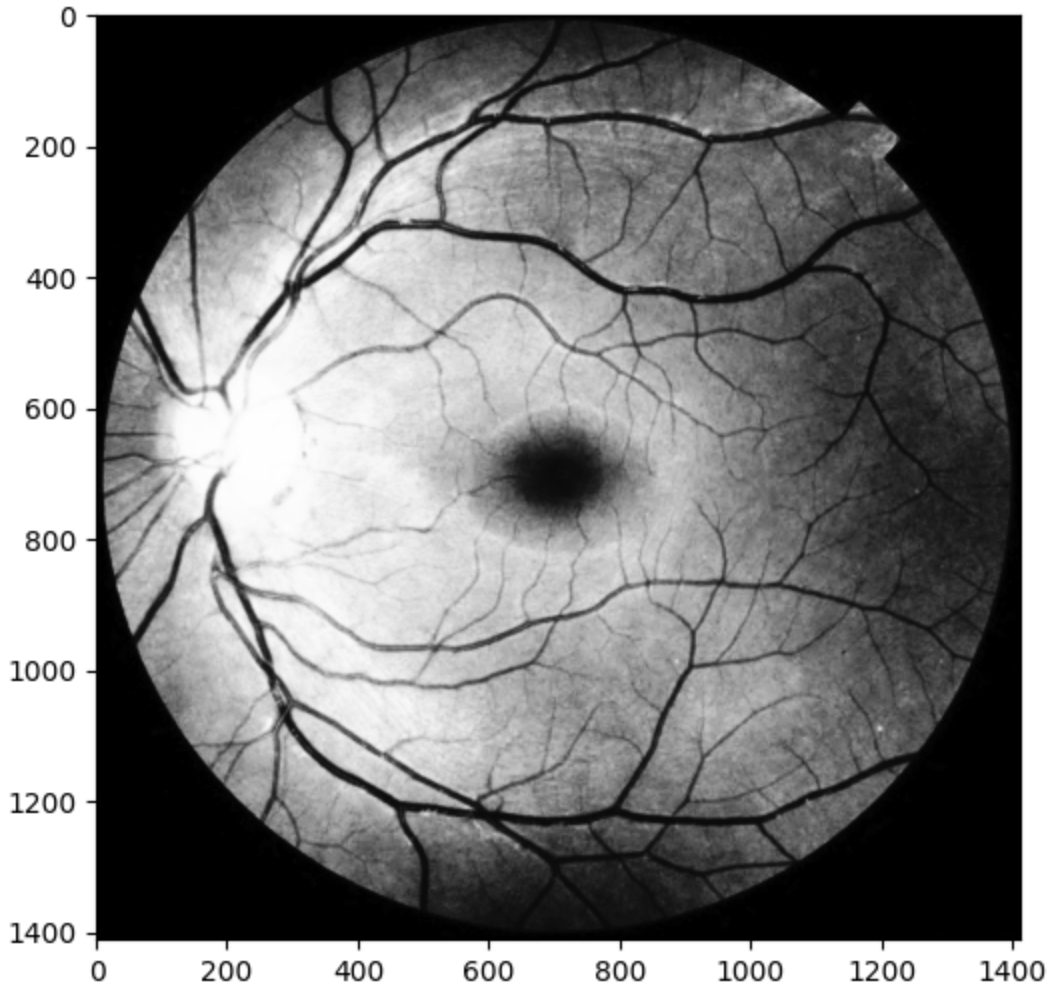
Histogram equalization can be performed in a number of ways. The obvious algorithm is global histogram equalization. The pixel values are transformed to equalize the histogram across the entire image. However, if illumination is inconsistent across the image, global equalization will not be optimal. An alternative is to perform local histogram equalization over small areas of the image. This method is known as **adaptive histogram equalization**. Adaptive equalization can compensate for uneven illumination across the image.

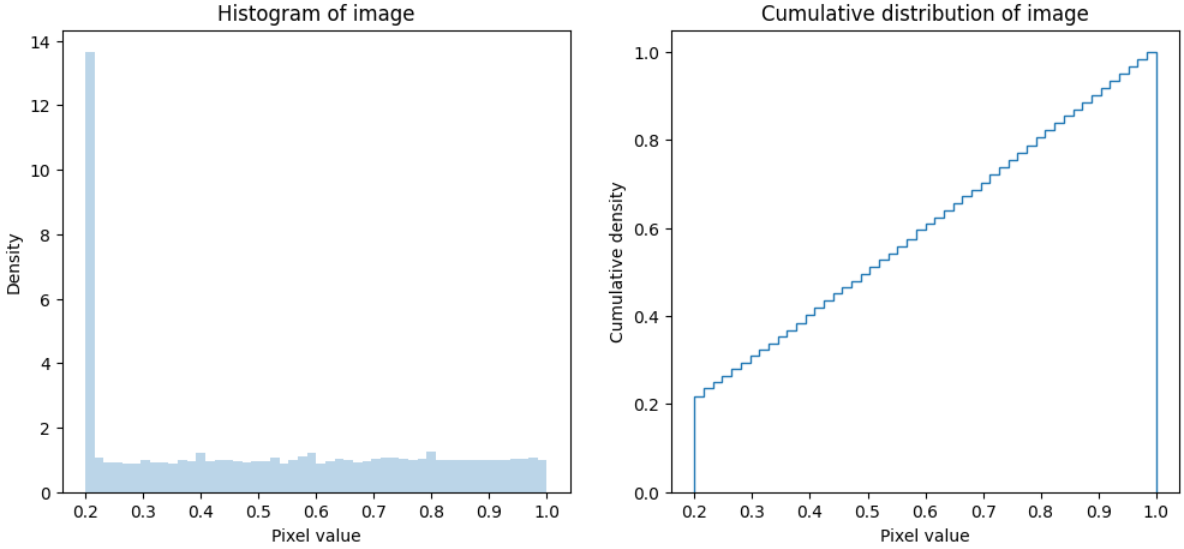
> **Exercise 1-3:** You will now apply both common types of histogram equalization to the gray-scale retina image. Use both the sklearn.exposure.equalize_hist function and the sklearn.exposure.equalize_adapthist function. Create and execute a function which does the following:
>
> 1. Executes the equalization function passed as an argument. Pass the function name as a string using the Python eval function.
> 2. Display the equalized gray-scale image using the `plot_grayscale()` function.

> 3. Plot the distribution of the pixel values of the equalized gray-scale image using the `plot_gray_scale_distribution()` function.
>
> Execute your function for the two equalization algorithms. You can do this by iterating over a list of the function names. Print a line indicating which function is being executed. Save the results in a Numpy object named `retina_gray_scale_equalized`.

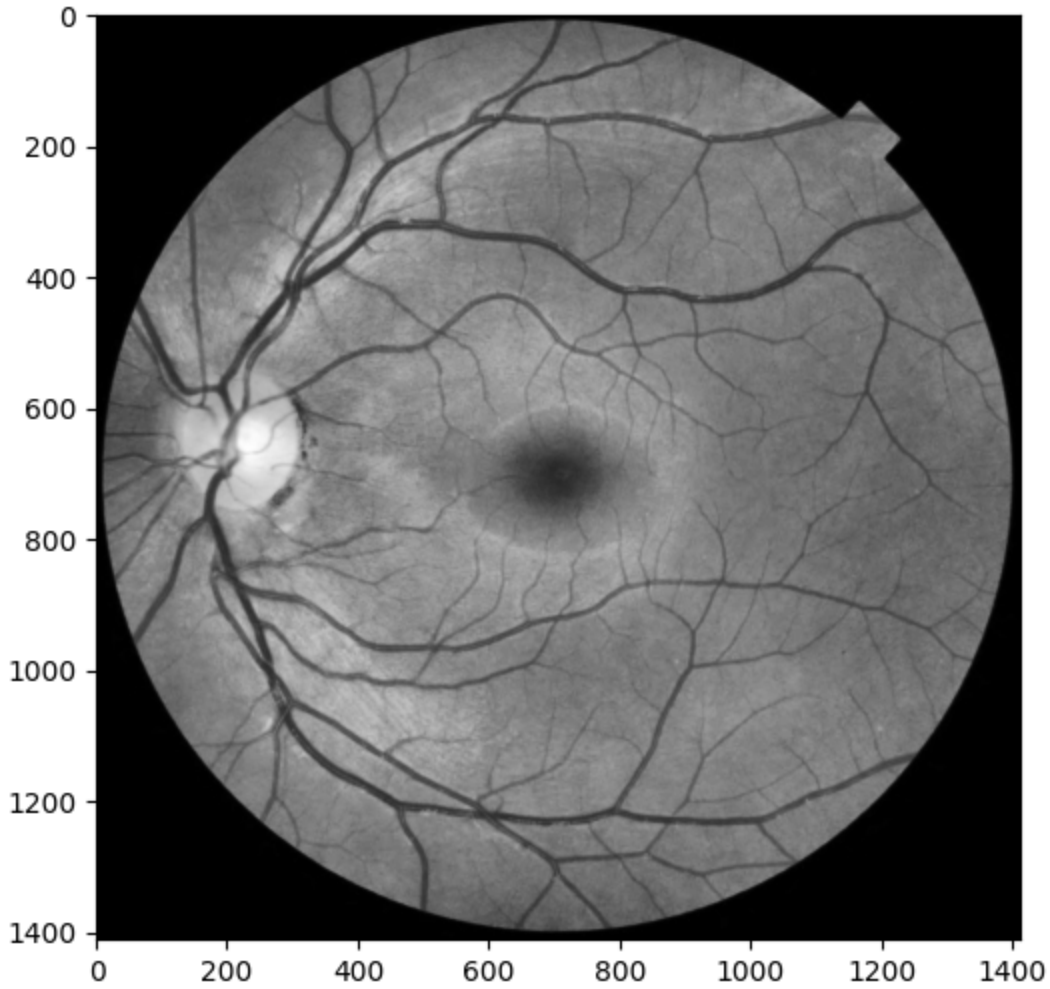
In [ ]:
```
def test_equalize(img, func):
    img_equalized = np.multiply(eval(func)(img), 255).astype(np.uint8)
    plot_grayscale(np.divide(img_equalized, 255.0))
    plot_gray_scale_distribution(np.divide(img_equalized, 255.0))
    return img_equalized


## Put you code below
def exec_equalization():
    # Functions to be executed
    sk_functions = ['exposure.equalize_hist',
                    'exposure.equalize_adapthist']

    # Execute functions iterating over a the list of function names
    images = []

    for i in sk_functions:
      # Print name of the function being executed
      print(f'\n\n\nFunction: {i}\n')

      # Call test_equalize()
      images.append(test_equalize(retina_gray_scale, i))

    # Save the results in a Numpy object named retina_gray_scale_equalized.
    retina_gray_scale_equalized = np.array(images)

    return retina_gray_scale_equalized



if __name__ == "__main__":
    # Return the equalized images in a 3-dimensional Numpy array
    retina_gray_scale_equalized = exec_equalization()
```

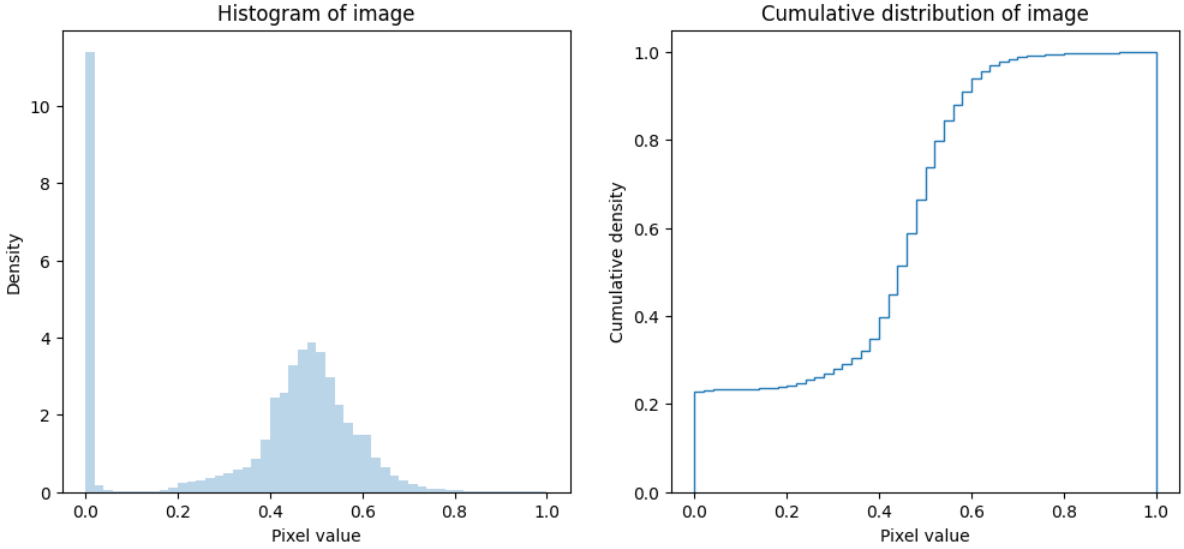
```
Function: exposure.equalize_hist
```






```
Function: exposure.equalize_adapthist
```





> Answer the following questions:
>
> 1. Compare the unequalized and equalized images. What aspects of the of the images are more apparent with the improved contrasted.
> 2. Compare the distributions of pixel values between the unequalized image, the random uniformly distributed image, and equalized images

> (2). Which of these histograms look the most similar and what does this tell you about the contrast of the image.
> 3. Does the difference in the distribution between the locally equalized image and the globally equalized image make sense and why?
>
> **End of exercise.**

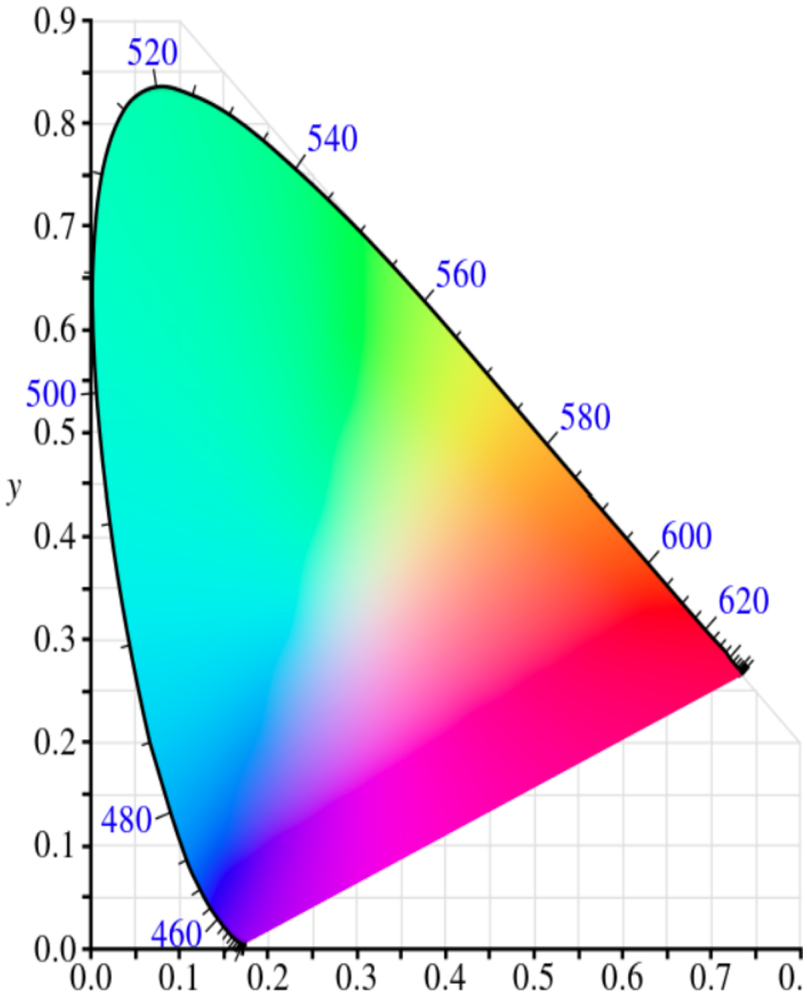
> **Answers:**
>
> 1. Equalized images have better contrast making it easier to identify the features that are same color in the original image. In this case, it is much easier to identify the veins and the dark spot in the center of the retina.
>
> 2. The **histogram** and **cumulative distribution** obtained with `exposure.equalize_adapthist` function are closer to the ones obtained from the unequalized image producing lower contrast. The **histogram** and **cumulative. distribution** obtained with `exposure.equalize_hist` function are closer to those obtained from the grey noise image, but produced higher contrast.
>
> 3. Yes, he difference in the distribution between the locally equalized image and the globally equalized image make sense. This is because the process to generate the locally equalized image takes small portions of the image to improve the contrast, while the process to generate the globally equalized image takes the range found in the whole image and then applies the contrast funtion in an even way for to the whole image.

Equalization for Multi-Channel Images

Contrast improvement, including histogram equalization, cannot be directly applied to the individual color channels of an RGB image. For RGB images the intensity of each color channel is mathematically unconstrained by the other two. However in reality, the brightness or intensity of each pixel depends on the value of all three channels. Therefore, independently applying 2-dimensional equalization to an RGB image causes normalization problems.

A common approach is to transform an RGB image into one of several possible formats that use a 2-dimensional color space map or **chromatisity** map of image intensity. There are a great many such choices, a number of which are supported in the skimage.color package.

As an example, the CIE 1931 or XYZ is a map of luminance, *Y*, over a two dimensional space of chromatiity. *Z* is the quasi-equal to blue, and *X* is a mix of the RGB values. The **XYZ** color space is shown in the figure below.

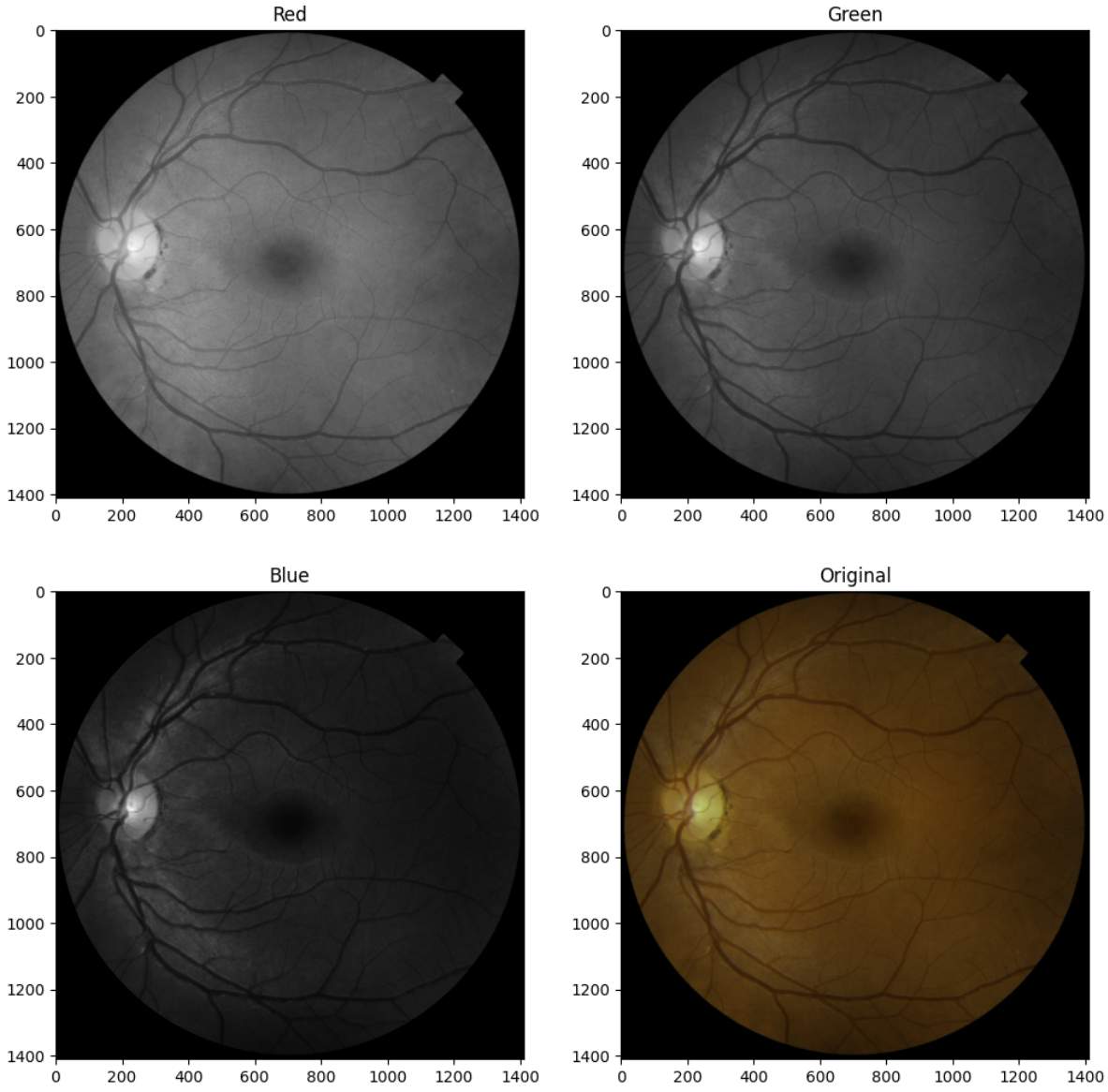


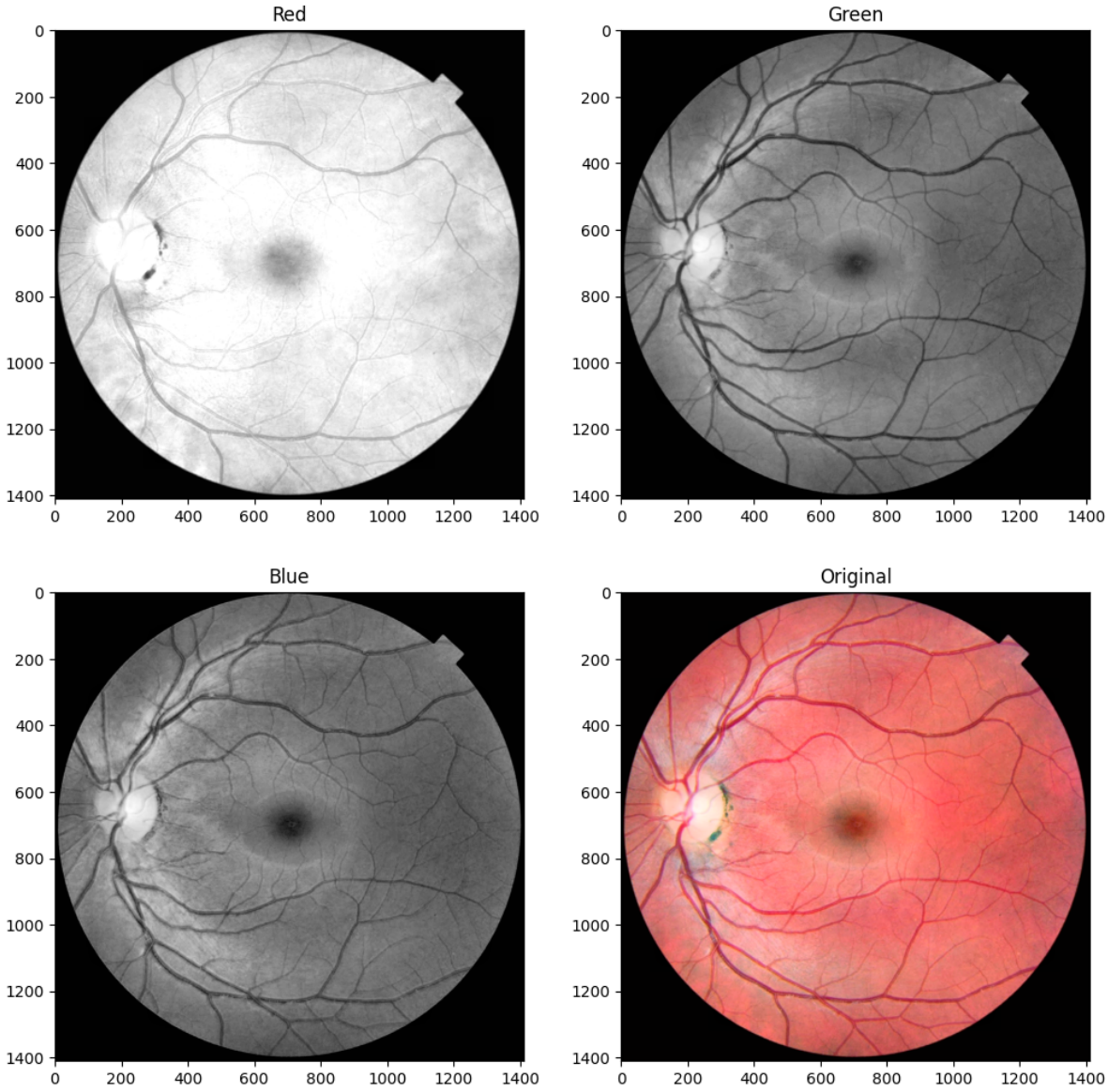
XYZ Color Space

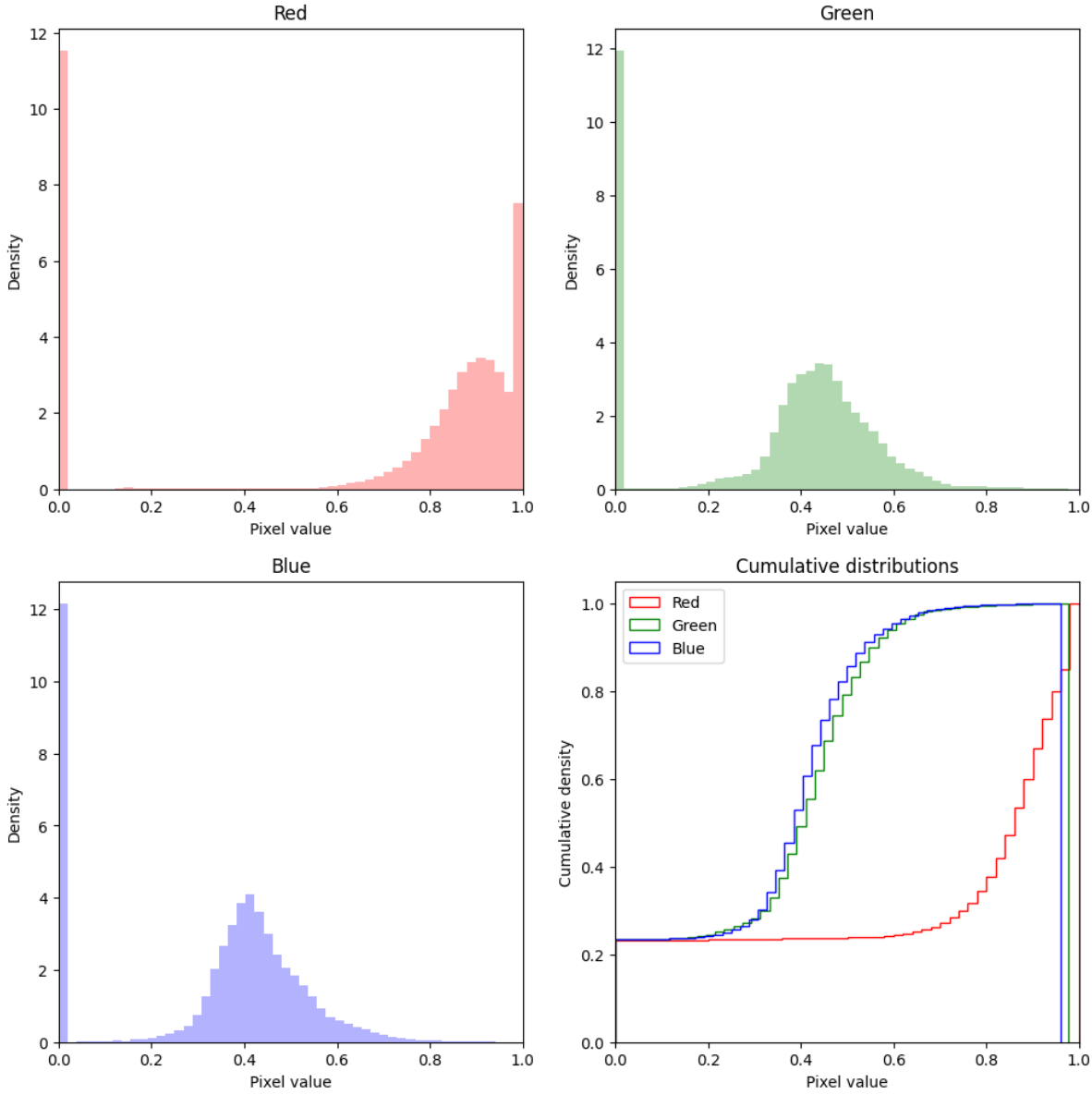
> **Exercise 1-4:** To apply the sklearn.exposure.equalize_adapthist function to a color image the following steps are used:
>
> 1. The skimage.color.rgb2xyz function is used to convert the *RGB* image to *XYZ* format.
> 2. The color channels of the transformed image are displayed.
> 3. The equalized *XYZ* image is converted to *RGB* and the color channels and densities are displayed.
>
> Execute the code and examine the results.

In [ ]:
```
retina_xyz = rgb2xyz(retina_image)
plot_3_color_channels(retina_xyz)

for i in range(3):
```

```
    retina_xyz[:,:,i] = exposure.equalize_adapthist(retina_xyz[:,:,i])

retina_rgb_equalized = exposure.rescale_intensity(xyz2rgb(retina_xyz), out_r
plot_3_color_channels(retina_rgb_equalized)
plot_image_distributions(retina_rgb_equalized, xlim=(0.0,1.0))
```




Red
Green
Blue
Original
0
200
400
600
800
1000
1200
1400



> Compare the equalized RGB images and pixel value densities to the images and densities of the original image and answer the answer the following questions: t
>
> 1. Did the histogram equalization achieve the goal of improving the contrast of the image both in the color channels and for the 3-channel color image, and why?
> 2. Given the use of the locally adapted histogram equalization algorithm, does the distribution of the pixel values in the 3 channels of the equalized image make sense, and why?
> 3. What is the evidence of saturation of the red color channel after equalization?
> 4. Does the change in color of the 3-channel color image make sense given the histogram equalization, and why?

> **End of exercise.**

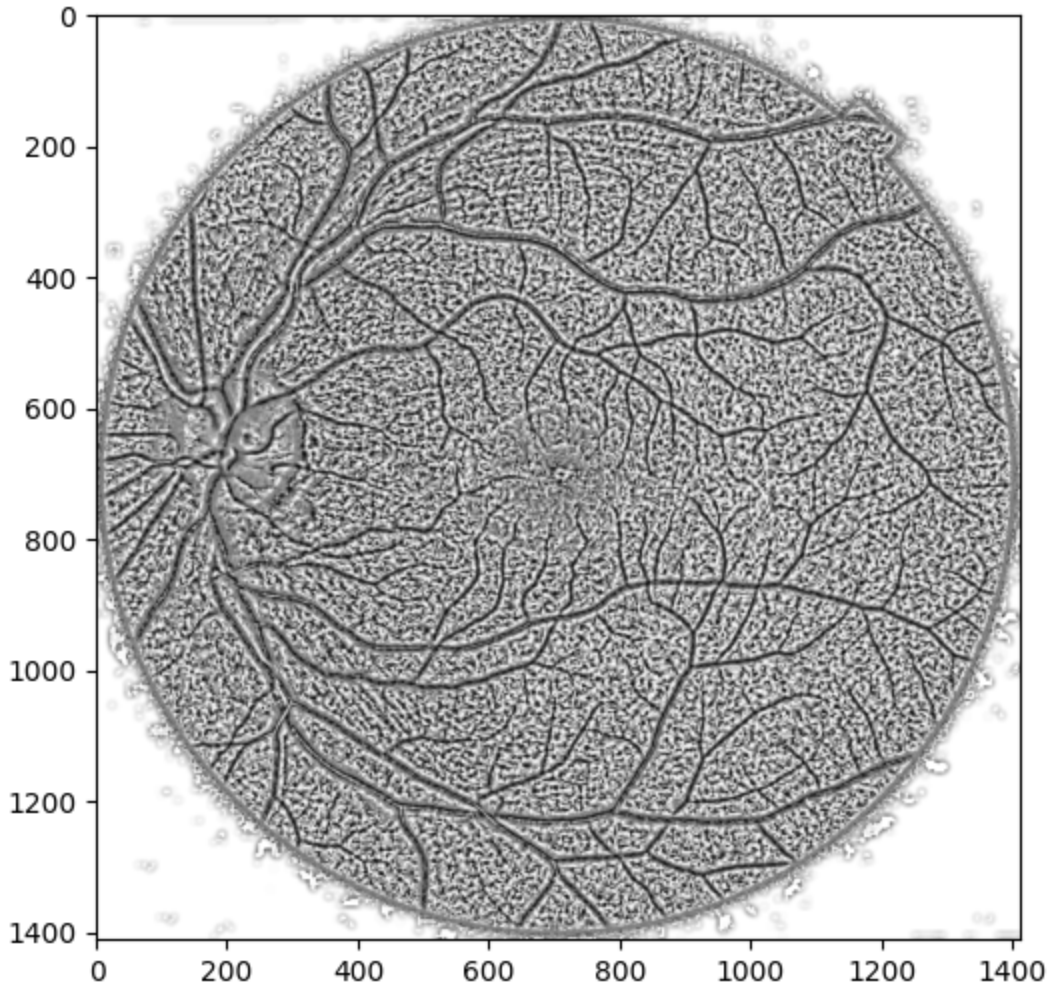
> **Answers:**

1. Yes, it did. The `exposure.equalize_adapthist` function was able to generate better contrast than the originally found by applying CLAHE. CLAHE improves contrast taking small regions of the graphic while clipping the histogram to avoid noise.
2. Yes, it does. It is easy to observe that the pixel values are now distributed more evenly and the cumulative distributions are smoother.
3. Saturation of the red color can be observed in the histogram of that color. It is represented by most of the pixel values being located at the right end of the graphic.
4. Yes, it does. The cumulative distributions show how the saturation of the red color is not that high compared with the image before being equalized, while green and blue have now higher pixel values.

Rank equalization

Contrast improvement is such an important data preparation step for computer vision that many algorithms have been proposed. One approach is to use rank statistics over a small region or local region of the image. The skimage.filters.rank.equalize function implements just such an algorithm. Execute the code in the cell below to see the effect this algorithm has on the gray-scale retina image. |

In [ ]:
```
retina_rank_equalized = equalize(np.multiply(retina_gray_scale, 255).astype(
plot_grayscale(retina_rank_equalized)
```



This locally equalized image shows considerably more detail than the global histogram equalization or adaptive histogram equalization methods. But, is this what we really want? In some cases yes. If fine details like texture are important to the computer vision solution, this equalization would be preferred. However, too much detail might lead to unnecessary complexity if the goal was to identify major structural elements of the image. In summary, the correct preprocessing for an image depends on the other algorithms one intends to apply.

Other Contrast Adjustments

Besides histogram equalization, numerous mathematical transformations for improving contrast have been developed. These methods seek to improve contrast by a nonlinear transformation of the pie values. We will examine just of few of the many possibilities:

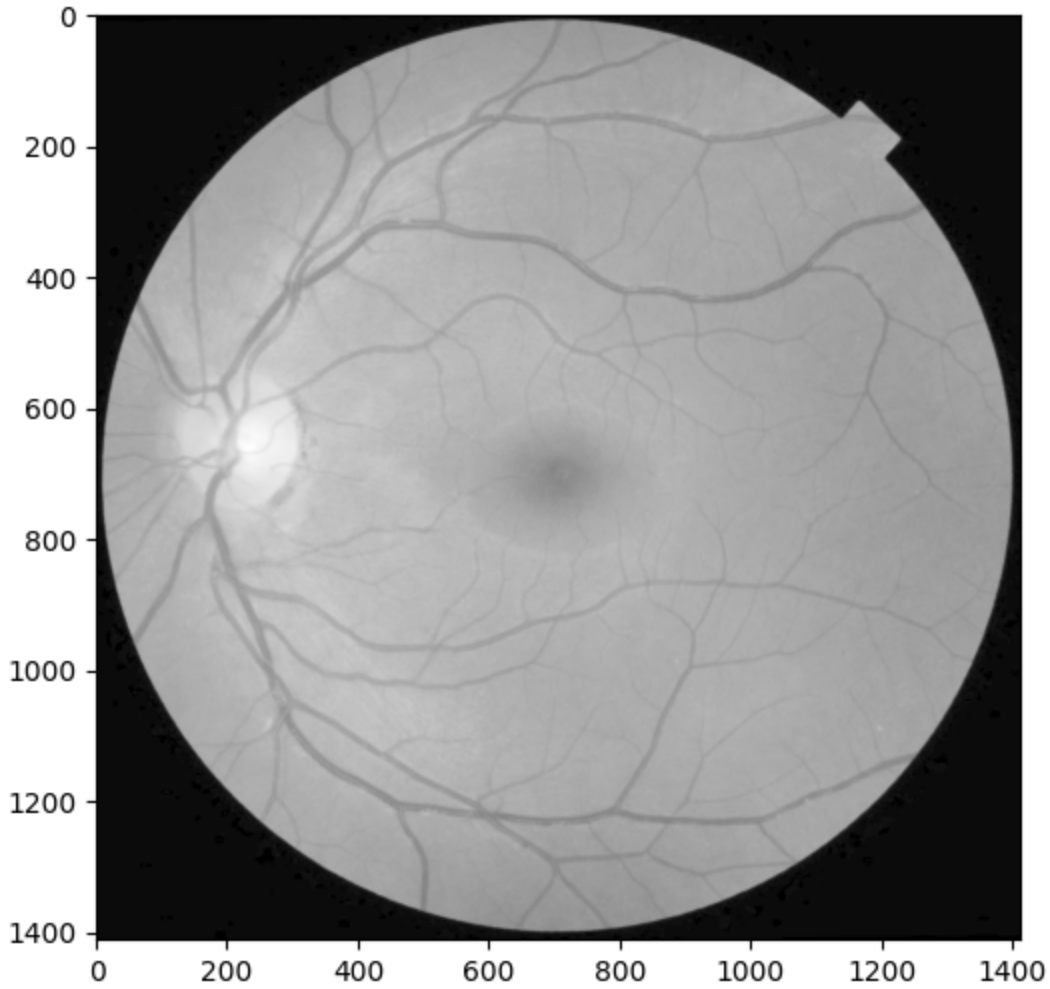
- **Gamma adjustment** is a power law transformation that shifts the histogram of the pixel values. For input pixel values x_i, and power, γ, the output pixel values are computed $x_i' = gain * x_i^{\gamma}$, were gain is an optional scale adjustment. If $\gamma < 1$ the histogram shifts to the right and for $\gamma > 1$ the histogram shifts to the left.

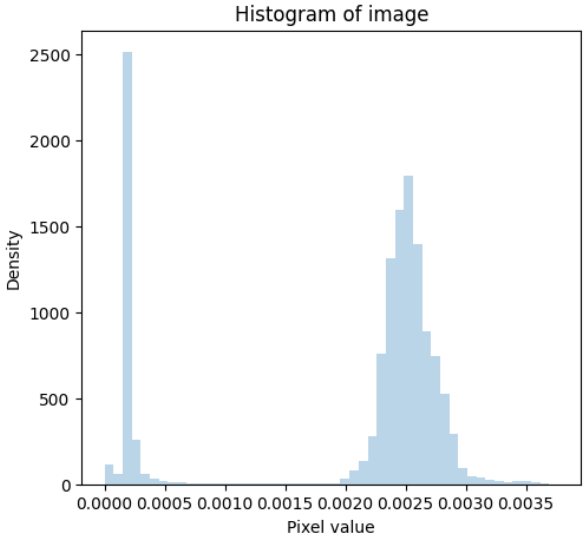
- **Logarithmic adjustment** computes a logarithmic compression of the pixel values, x_i, $x_i' = gain * log(x_i + 1)$, where gain is an optional scale adjustment.
- **Sigmodal adjustment** is a nonlinear transformation of the pixel values, x_i, with a cutoff value, $x_i' = \frac{1}{1+exp(gain*(cutoff-x_i))}$, and an optional gain adjustment.

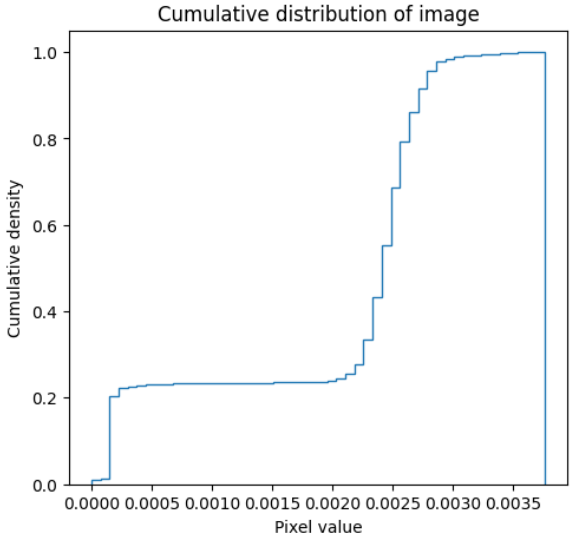
> **Exercise 1-5:** To get a feel for the gamma adjustment method you will now do the following:
>
> 1. Iterate over gamma values of $[0.5, 2.0]$.
> 2. Apply the gamma adjustment skimage.exposure.adjust_gamma function to the gray scale retina image.
> 3. Display the adjusted image and the pixel value density. Make sure you include a printed indication of gamma for each case.
> 4. Execute your code.

In [ ]:
```
## Place your code below

gammas = [0.5, 1.0, 1.5, 2.0]
for gamma in gammas:
    print(f'\n\n\nGamma correction with gamma = {gamma}\n')
    retina_gamma_corrected = exposure.adjust_gamma(retina_gray_scale, gamma)
    plot_grayscale(retina_gamma_corrected)
    plot_gray_scale_distribution(np.divide(retina_gamma_corrected, 255.0))
```

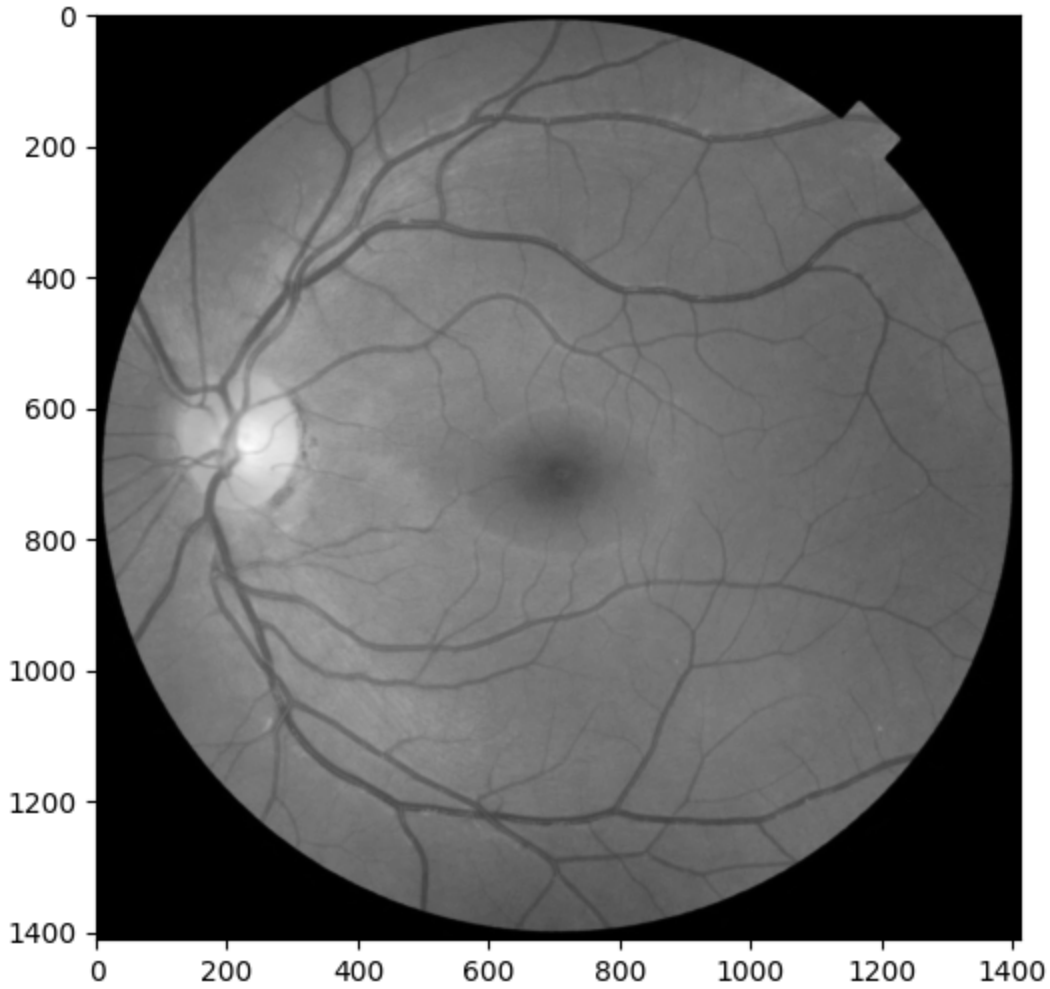
```
Gamma correction with gamma = 0.5
```

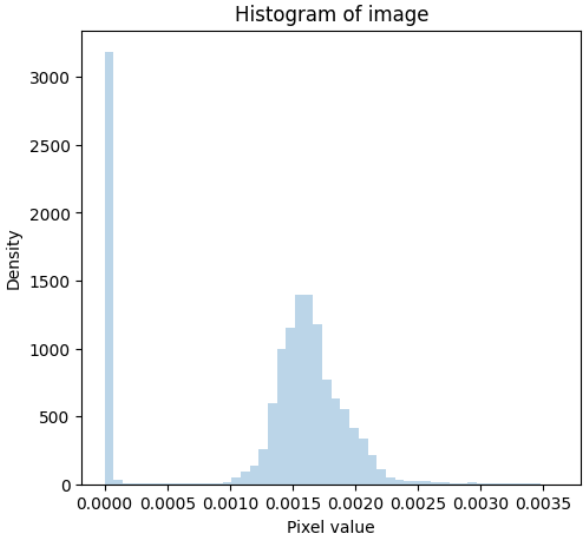


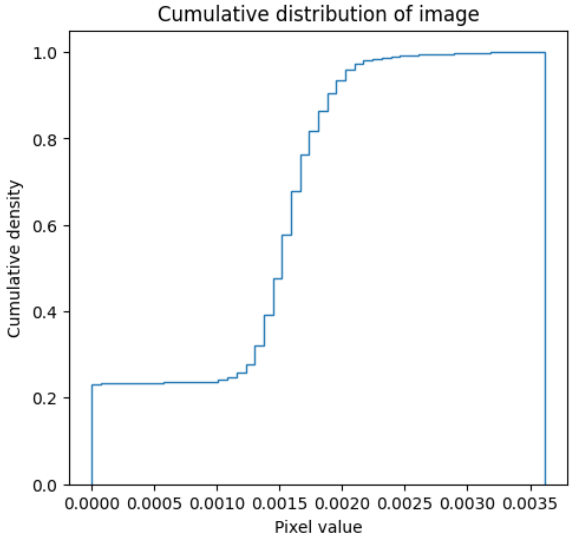




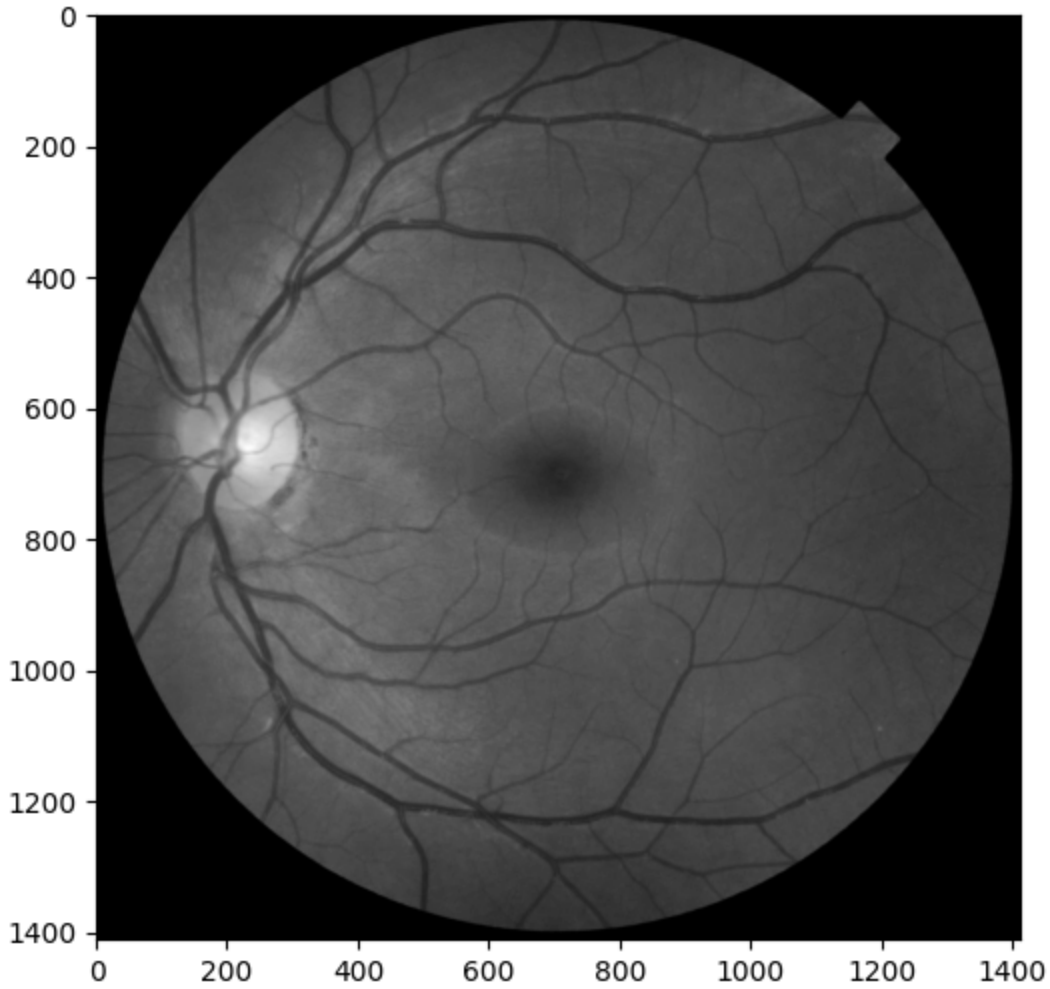

```
Gamma correction with gamma = 1.0
```

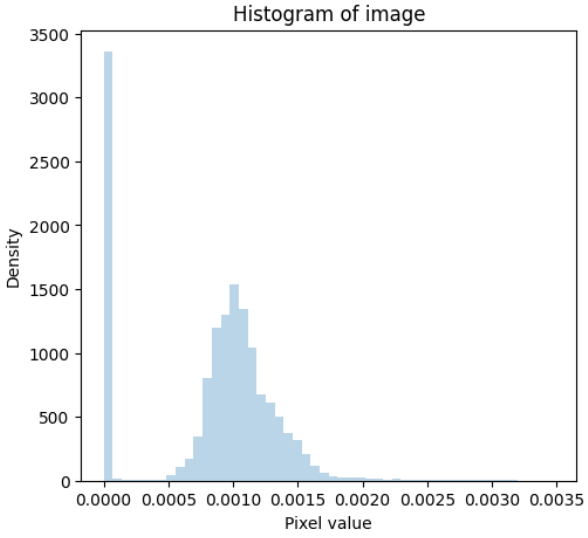


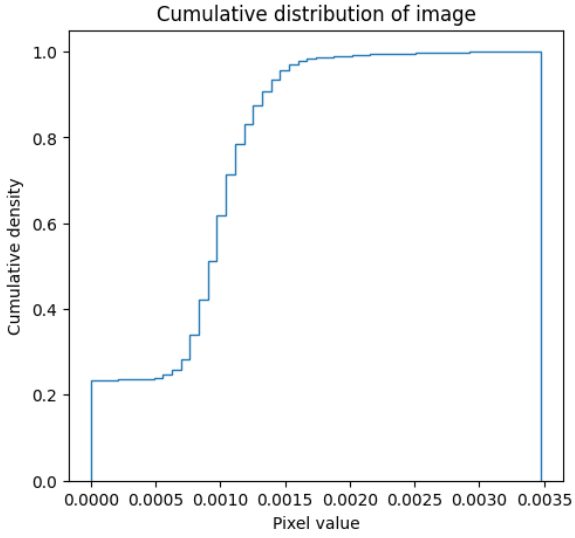




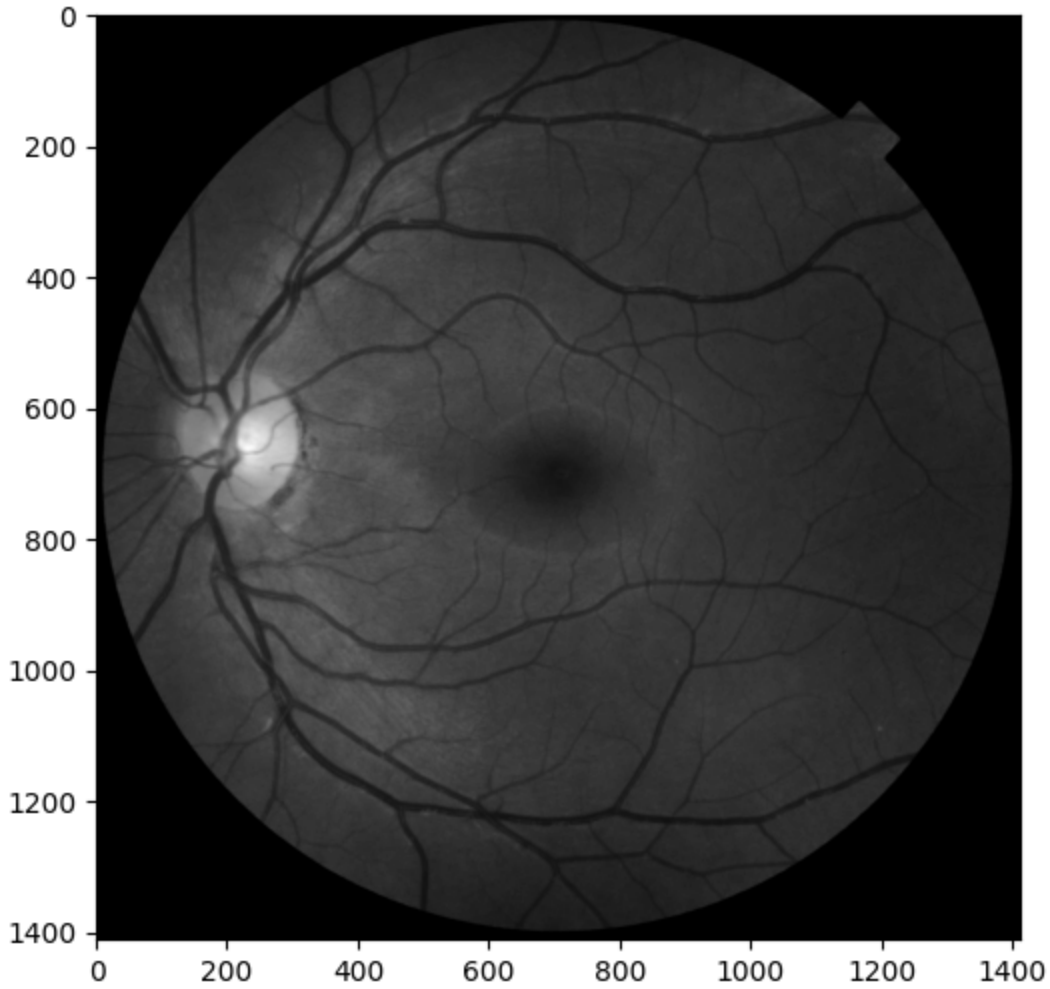

```
Gamma correction with gamma = 1.5
```

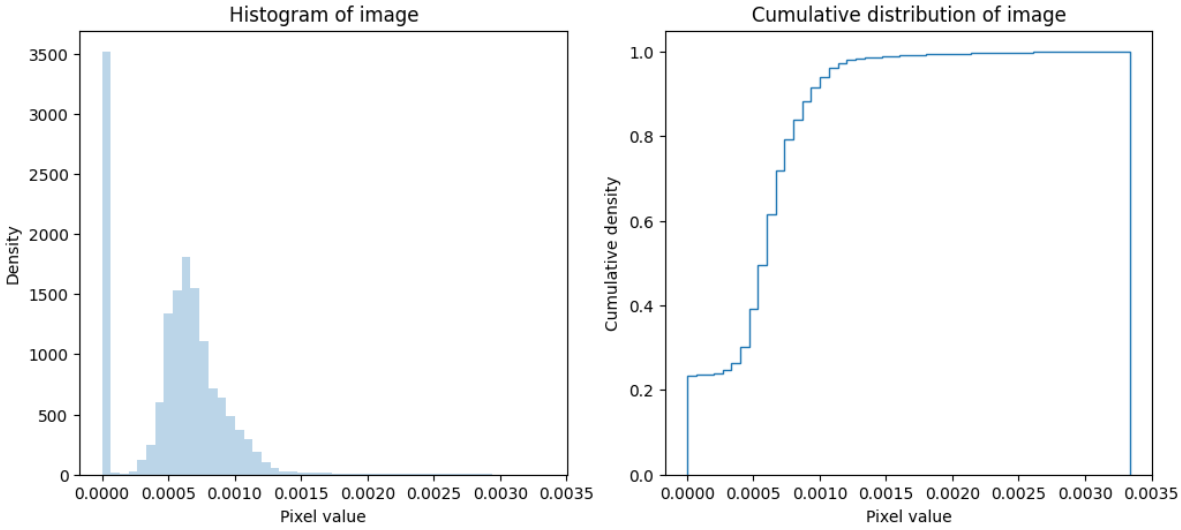






Gamma correction with gamma = 2.0





> Examine your results for the values of gamma, comparing them to the original gray-scale image. How does the brightness of the image and the distribution of pixel values change with gamma?
> **End of exercise.**

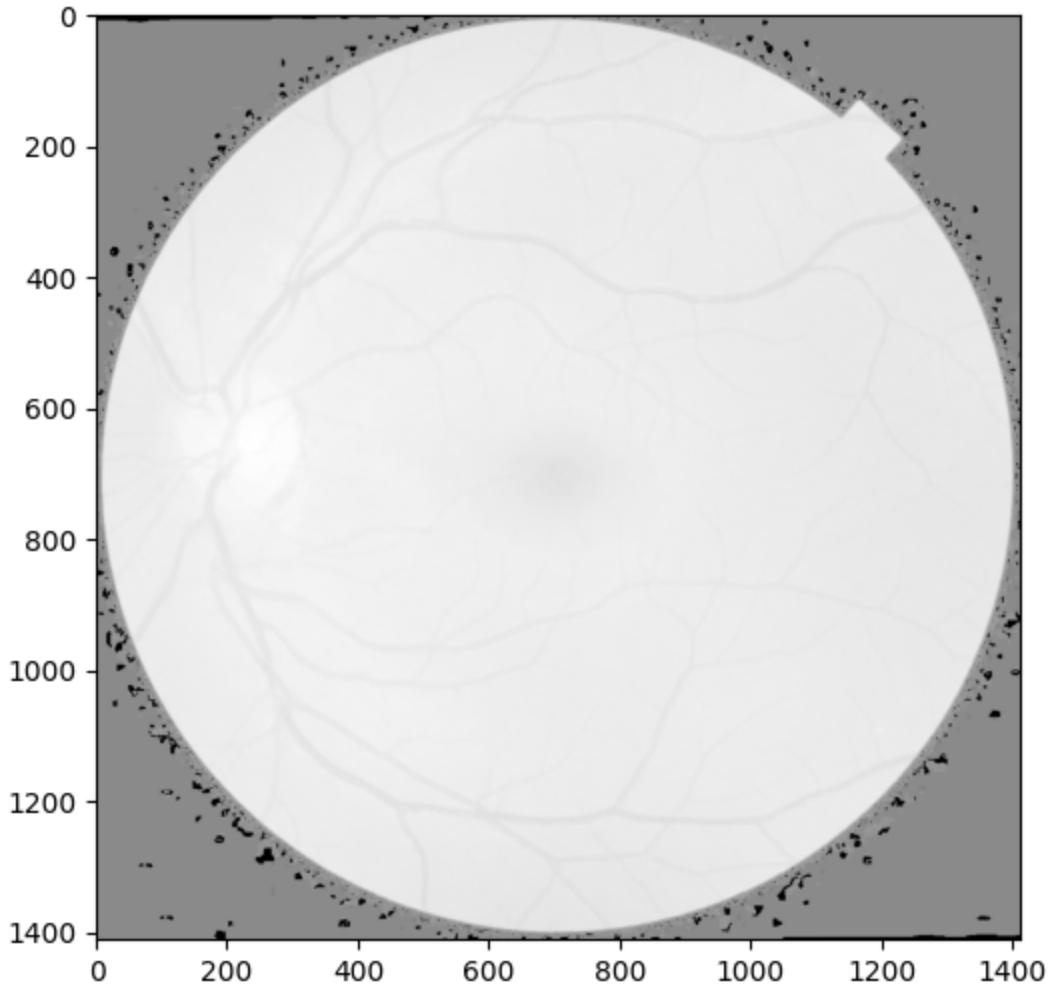
> **Answer:**

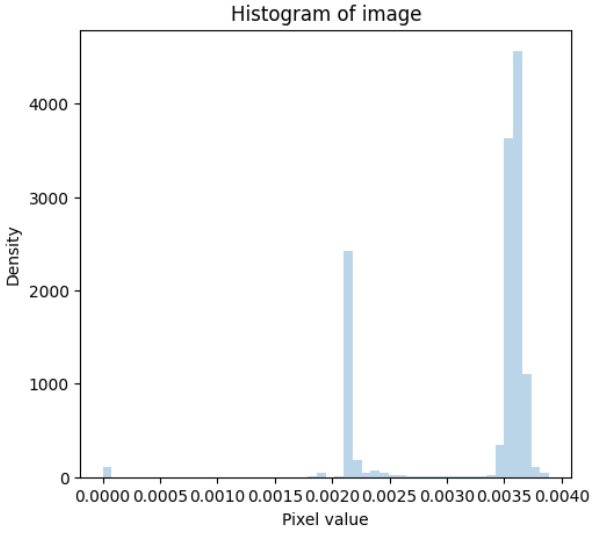
> For gamma values greater than 1, the image becomes darker and the histogram shifts toward left, while for gamma values less than 1 the image becomes lighter and the histogram shifts towards right. Gamma equals 1 does not alter the input image's brightness.

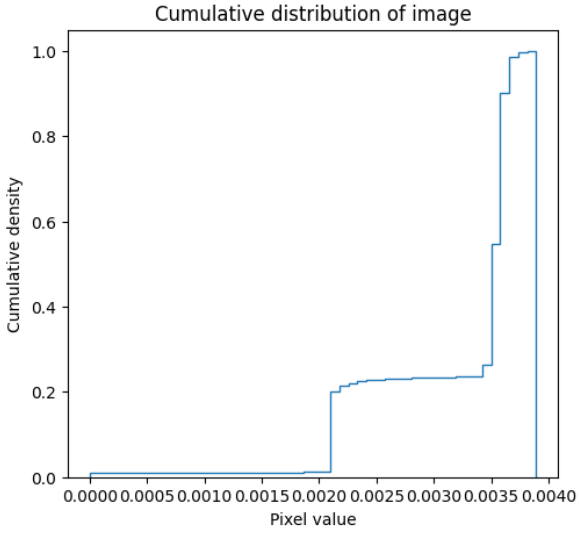
> **Exercise 1-6:** To get a feel for the sigmodial adjustment method you will now do the following:
>
> 1. Iterate over cutoff values of $[0.3, 0.4, 0.5]$.
> 2. Apply the gamma adjustment skimage.exposure.adjust_sigmoid function to the gray scale retina image.
> 3. Display the adjusted image and the pixel value density. Make sure you include a printed indication of gamma for each case.
> 4. Execute your code.

In [ ]:
```
## Place your code below

# I added values to the list of cutoffs to better see the effect of the gamm
cutoffs = [0.1, 0.3, 0.4, 0.5, 1.0, 2.0]
for cutoff in cutoffs:
    print(f'\n\n\nGamma correction with gamma = {cutoff}\n')
    retina_sigmoid_corrected = exposure.adjust_gamma(retina_gray_scale, cutc
    plot_grayscale(retina_sigmoid_corrected)
    plot_gray_scale_distribution(np.divide(retina_sigmoid_corrected, 255.0))
```

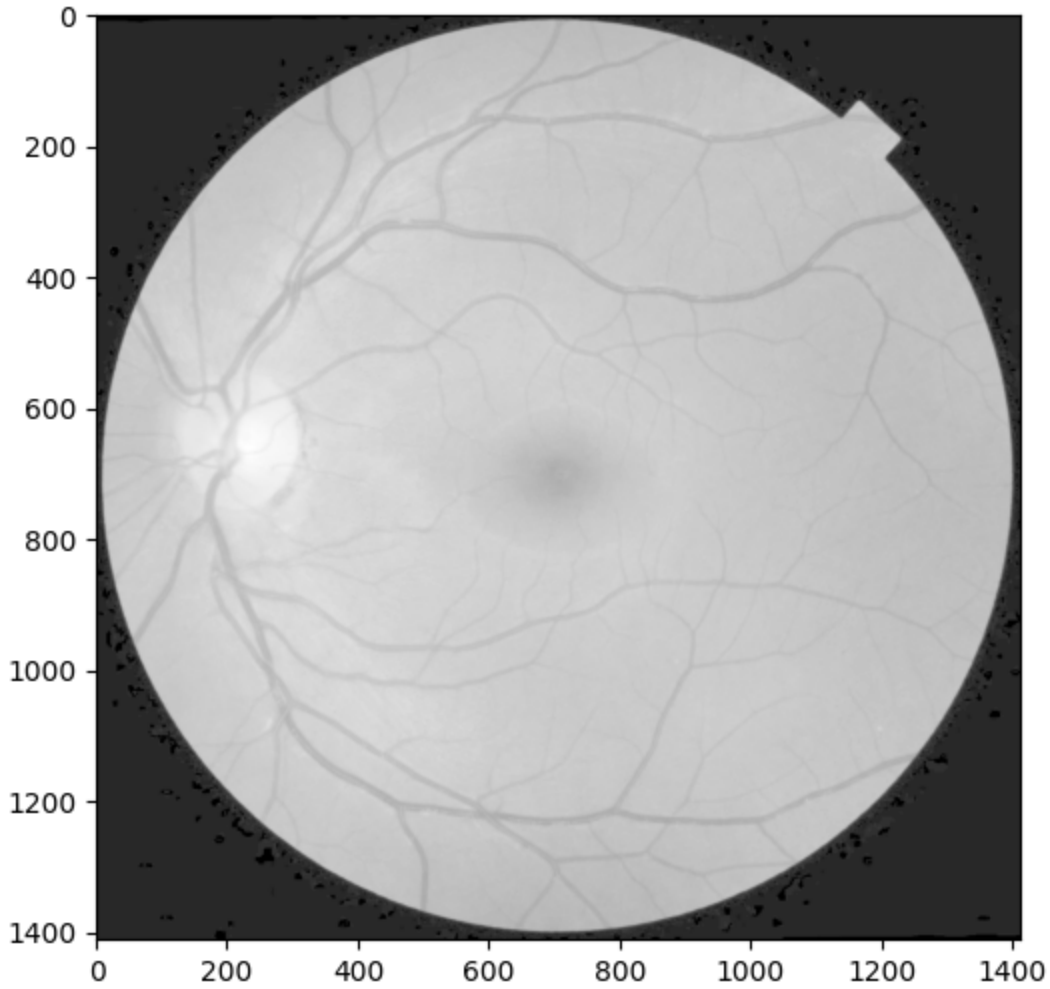
```
Gamma correction with gamma = 0.1
```

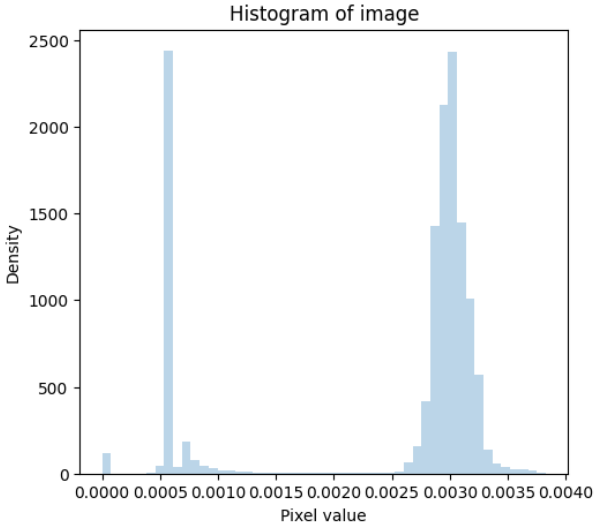


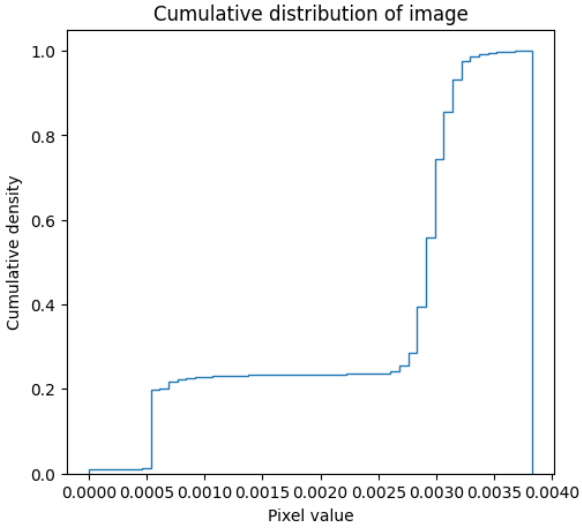




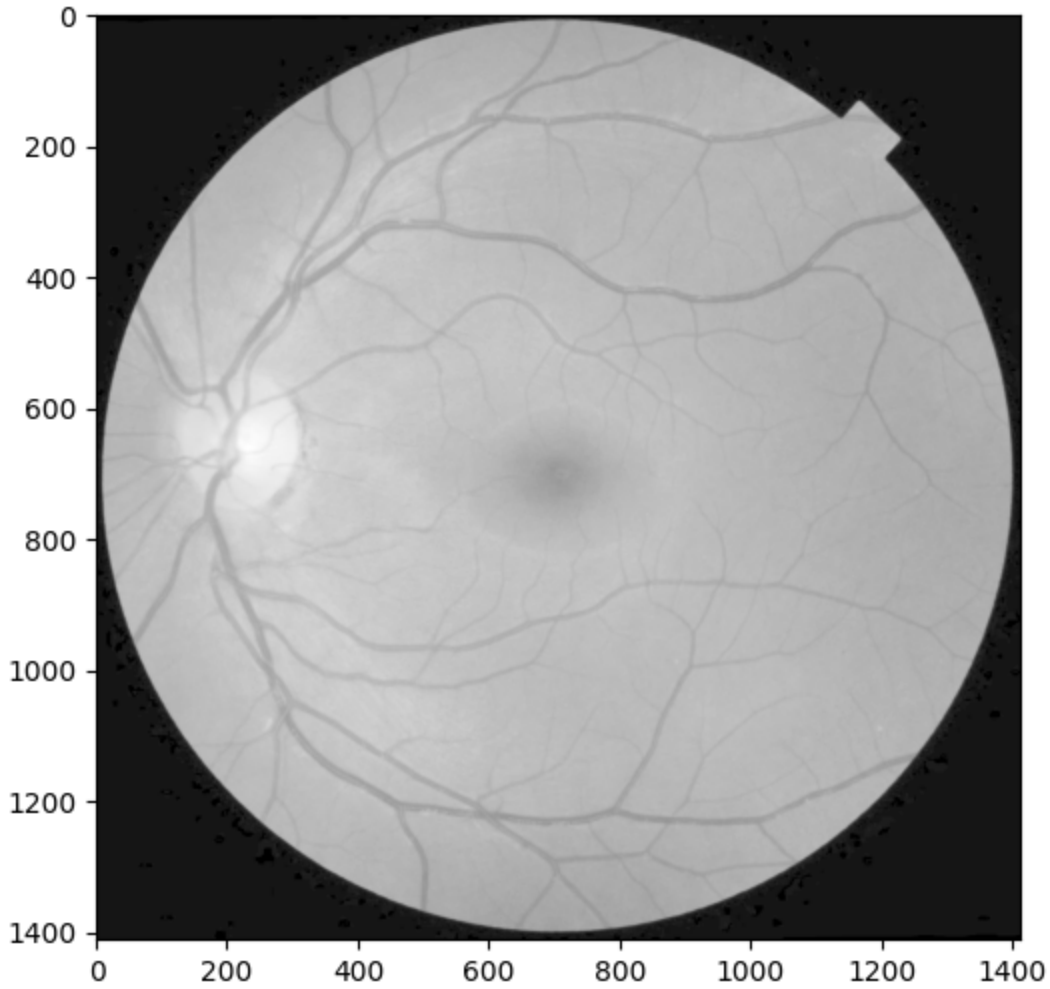
Gamma correction with gamma = 0.3

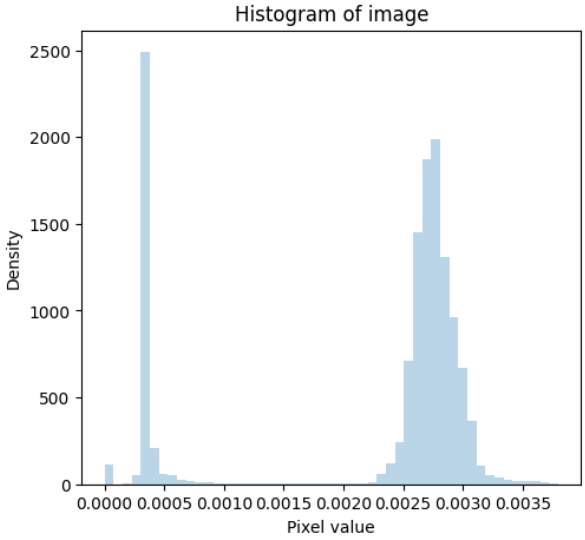


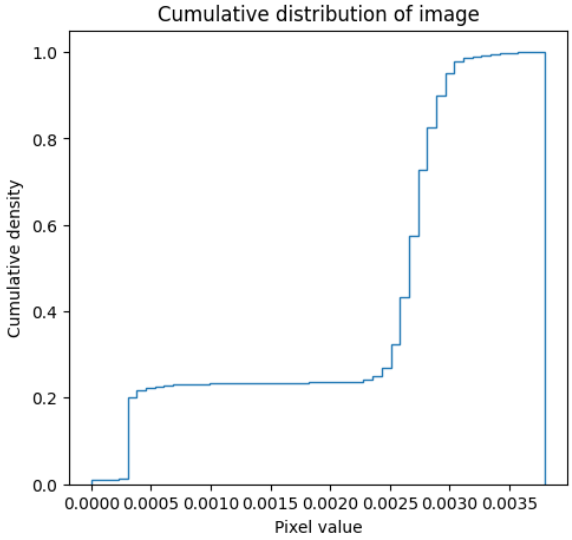




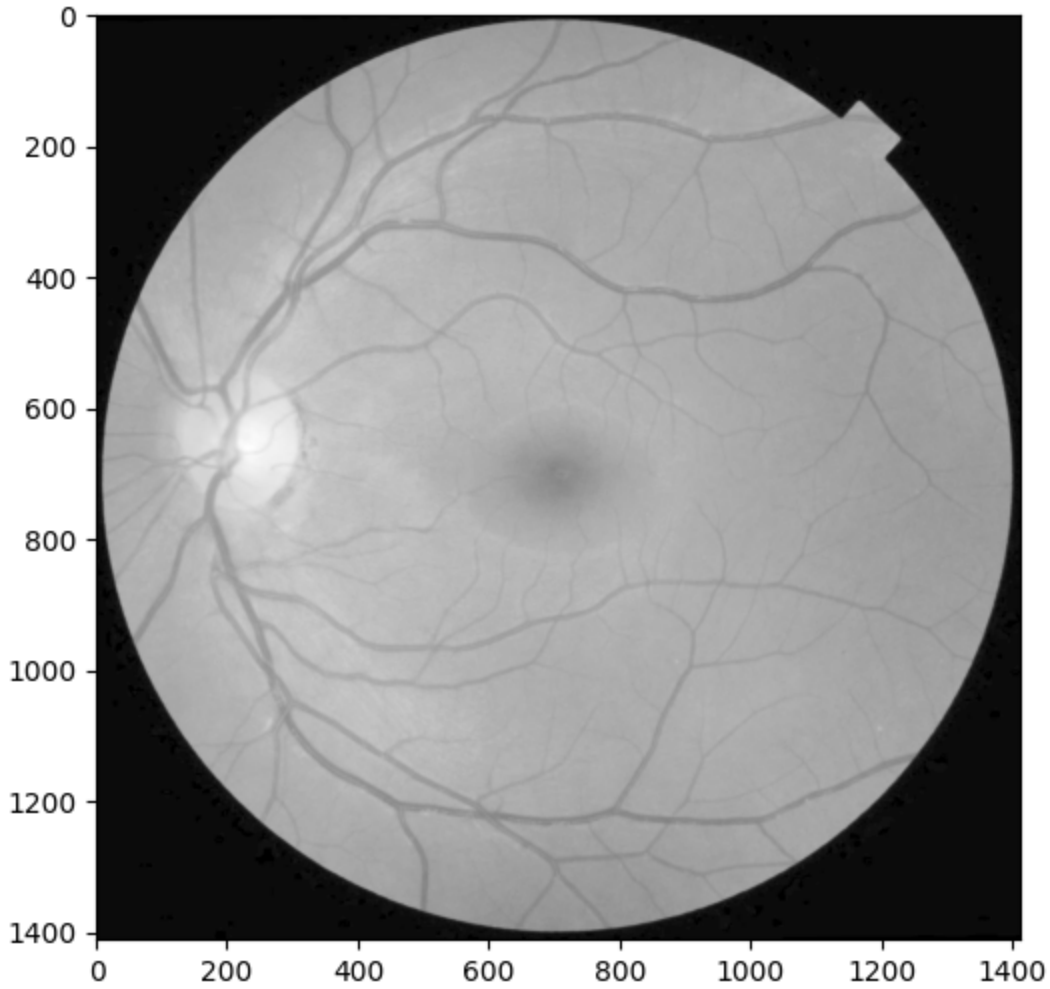
Gamma correction with gamma = 0.4

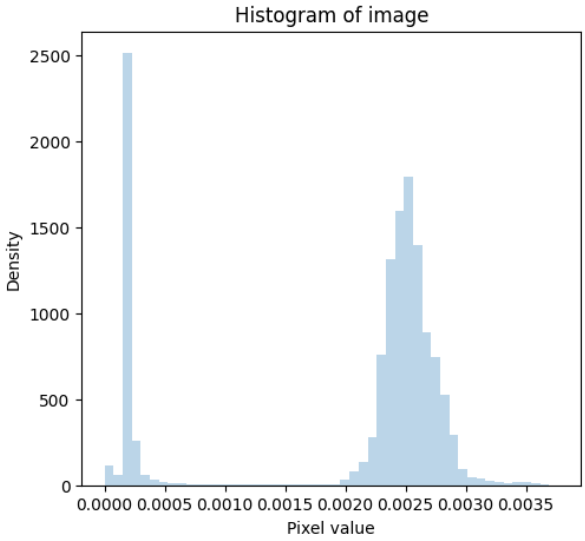


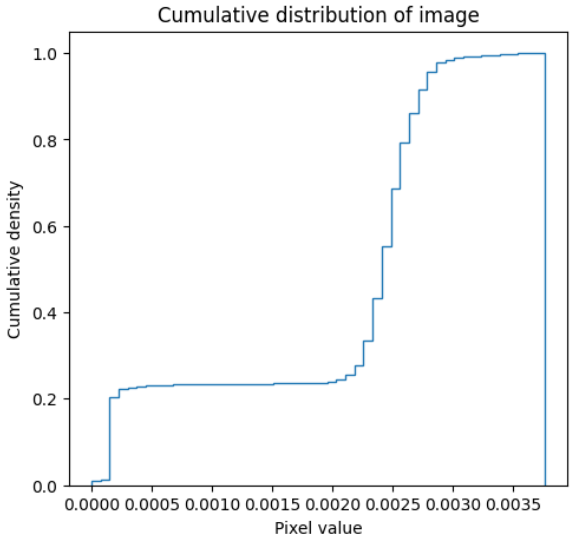




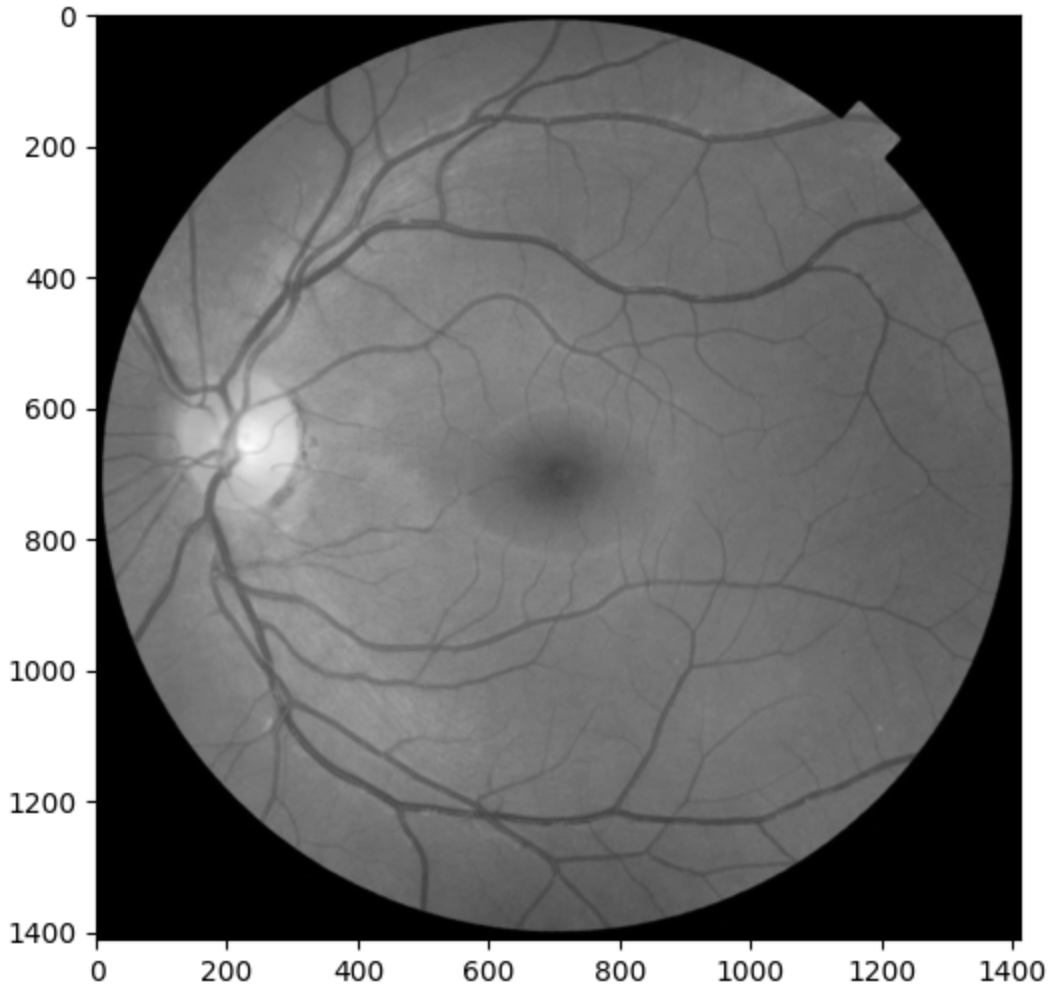
Gamma correction with gamma = 0.5

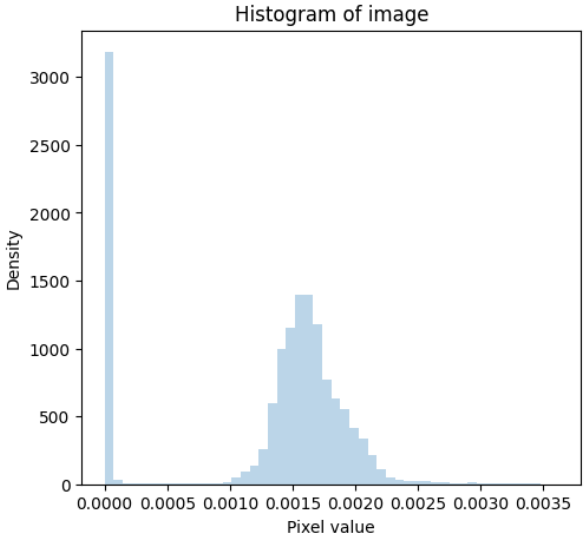


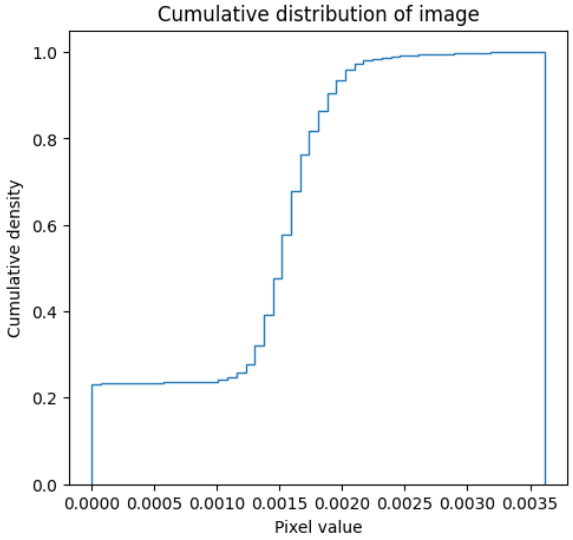




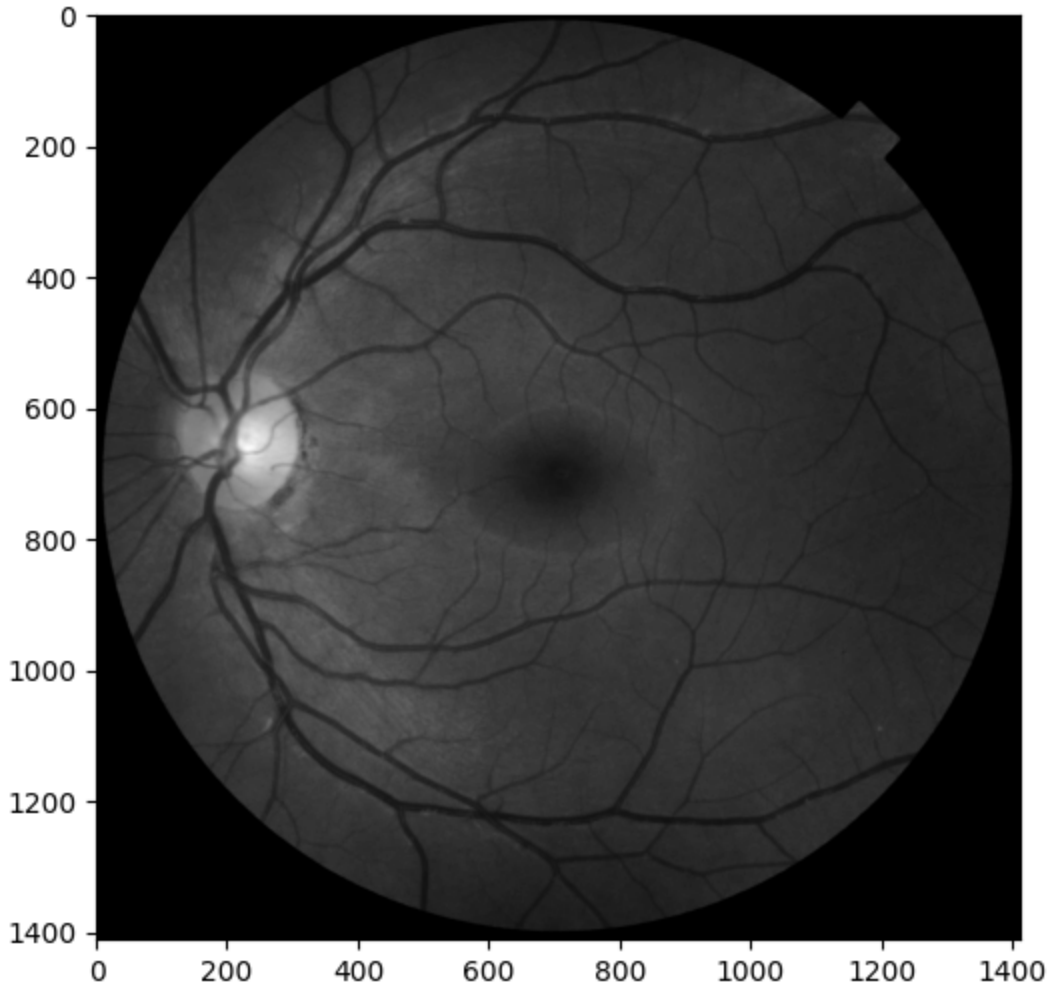

```
Gamma correction with gamma = 1.0
```

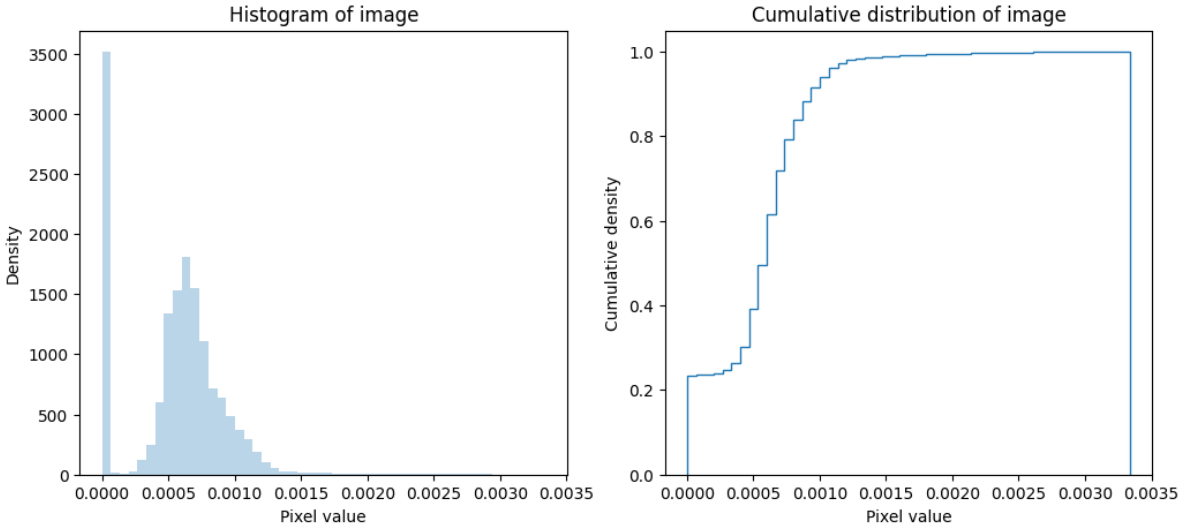







```
Gamma correction with gamma = 2.0
```





> Examine the images and the pixel value densities for the resulting images and compare these to the original gray-scale image. How does the brightness and densities change with the cutoff value? Pay attention to expansion or compression of the range of pixel values.

> **Answer:**

> For cutoff values lower than 0.5, contrast becomes lighter and the histogram shifts toward right, while for cutoff values higher than 0.5 the image becomes darker and the histogram shifts towards left. Cutoff equals 0.5 does not alter the input image's contrast.

Binary Images

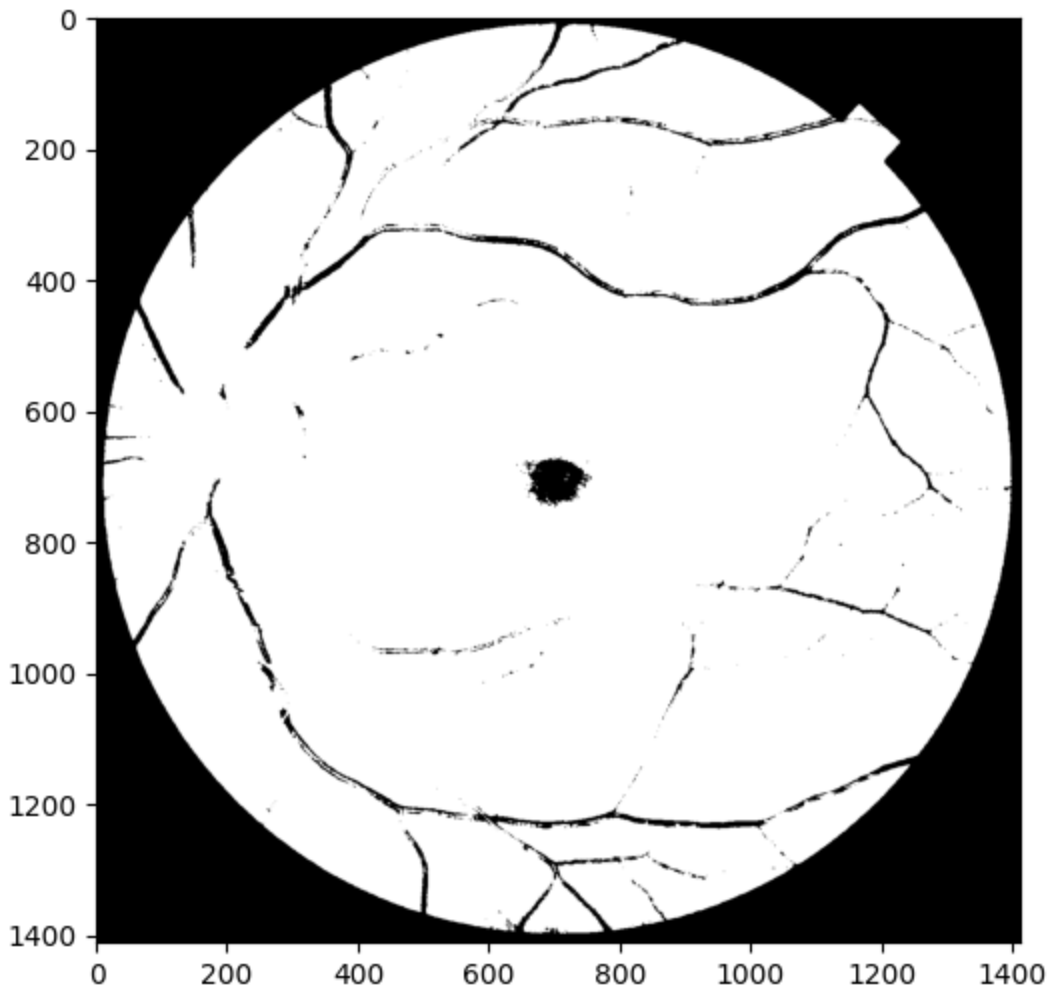
Many computer vision algorithms operate on binary images. Primarily these methods are in the category of **morphology**, which we will explore later. A binary image has only two values, $\{positive, negative\}$ or $\{1, 0\}$.

> **Exercise 1-7:** You will complete a function named `transform2binry()` to convert either a 3-channel color image or gray scale image to a integer binary image, $\{1, 0\}$, given a threshold value in the range $0 \leq threshold \leq 1$ as an argument. The function must do the following:
>
> 1. If the image is multi-channel, convert it to gray-scale.
> 2. Transform the threshold value to the fraction of the range of the pixel values. Print the transformed threshold value.
> 3. Apply the threshold to the gray-scale pixel values and return the binary images.
> 4. Execute your function on the **locally equalized color** retina image, print the dimensions of the binary image, and display the image, using a threshold value of 0.37.
> 5. Execute your function on the **locally equalized gray scale** retina image, print the dimensions of the binary image, and display the image, using a threshold value of 0.37.

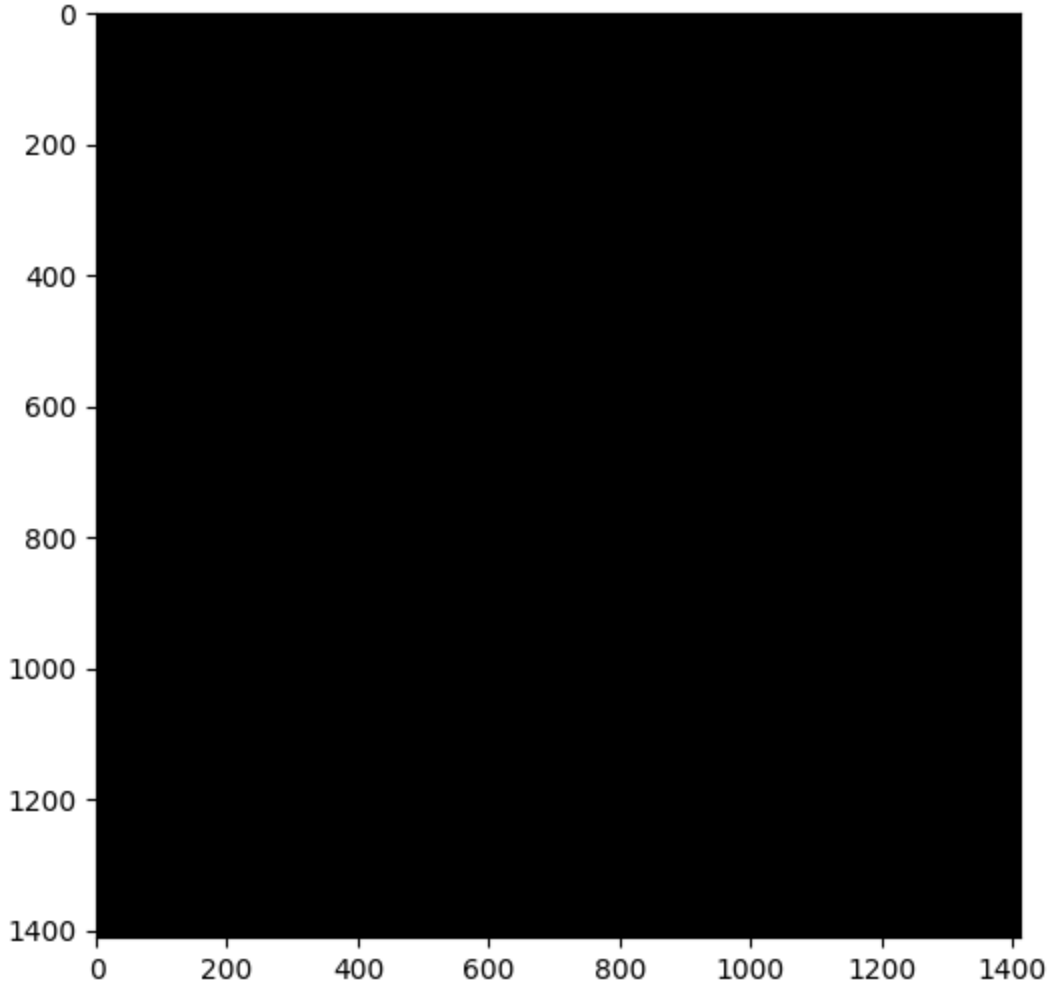
In [ ]:
```
## Put you code below
def transform2binary(img, threshold = 0.5):
    '''
    Function converts a gray scale or color image to binary values.
        Args:
            img - a color or gray scale image file
            threshold = the threshold value on a 0-1 scale. Pixel values >=
        Returns:
            Binary 2d image as a numpy array
    '''
    ## Make sure to use a copy to prevent weird bugs that are nearly impossi
    img = np.copy(img)


    ## Complete the code below
```

```
    ## Convert to gray scale if needed

    # Check if the image is color or gray scale
    if(img.ndim == 3):
        img = rgb2gray(img)
        print("Color image converted to grey")

    else:
        print("Grey image detected")

    print(f"Threshold value = {threshold}")
    img[img >= threshold] = 1
    img[img < threshold] = 0

    return img


for i in [retina_rgb_equalized, retina_gray_scale_equalized[0], retina_gray_
    retina_binary = transform2binary(i, threshold=0.37)
    print(retina_binary.shape)
    plot_grayscale(retina_binary)
```

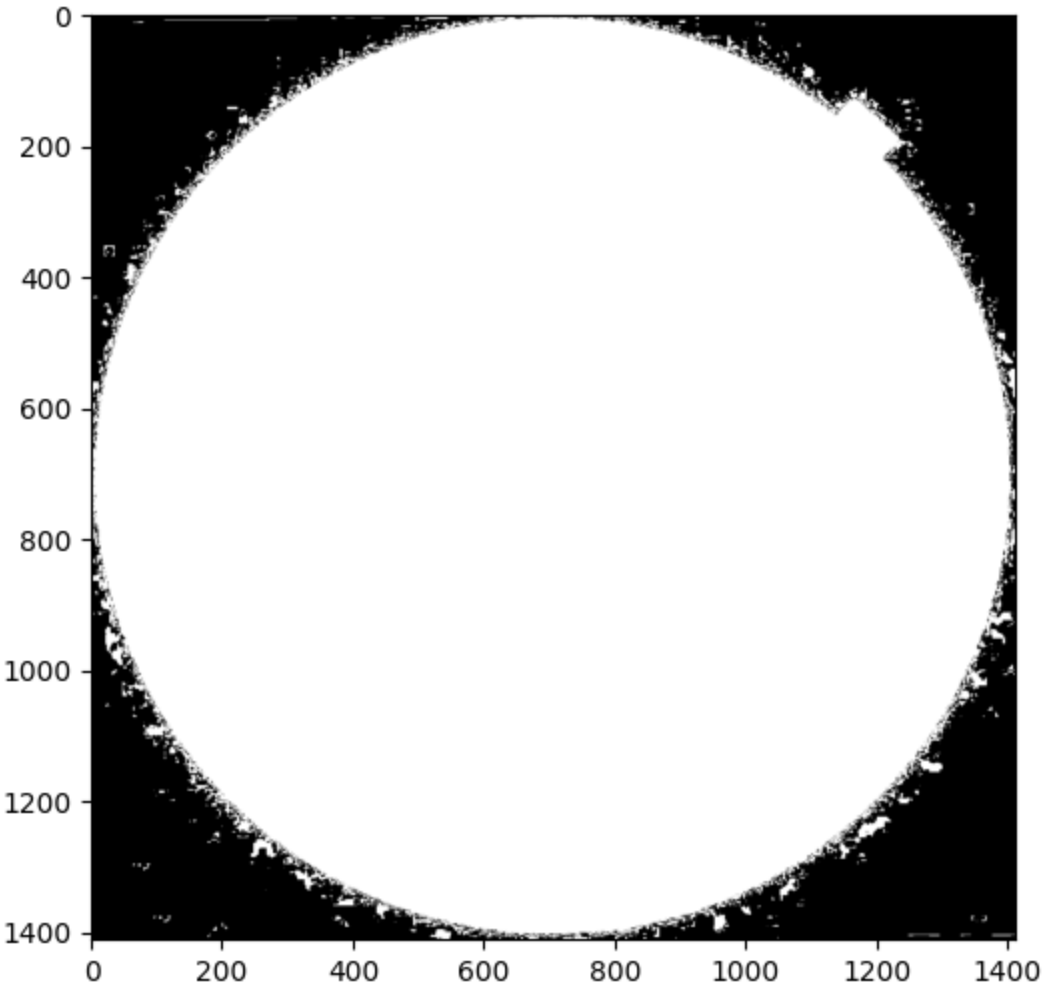
```
Color image converted to grey
Threshold value = 0.37
(1411, 1411)
```



```
Grey image detected
Threshold value = 0.37
(1411, 1411)
```




```
Grey image detected
Threshold value = 0.37
(1411, 1411)
```



> Examine the image and answer the following questions.
>
> 1. Does the binary image created from the equalized color image capture key aspects of the retina and why?
> 2. Compare the binary images created from the equalized color image and the equalized gray scale image. Is there any difference, and is this the result you would expect?
>
> **End of exercise.**

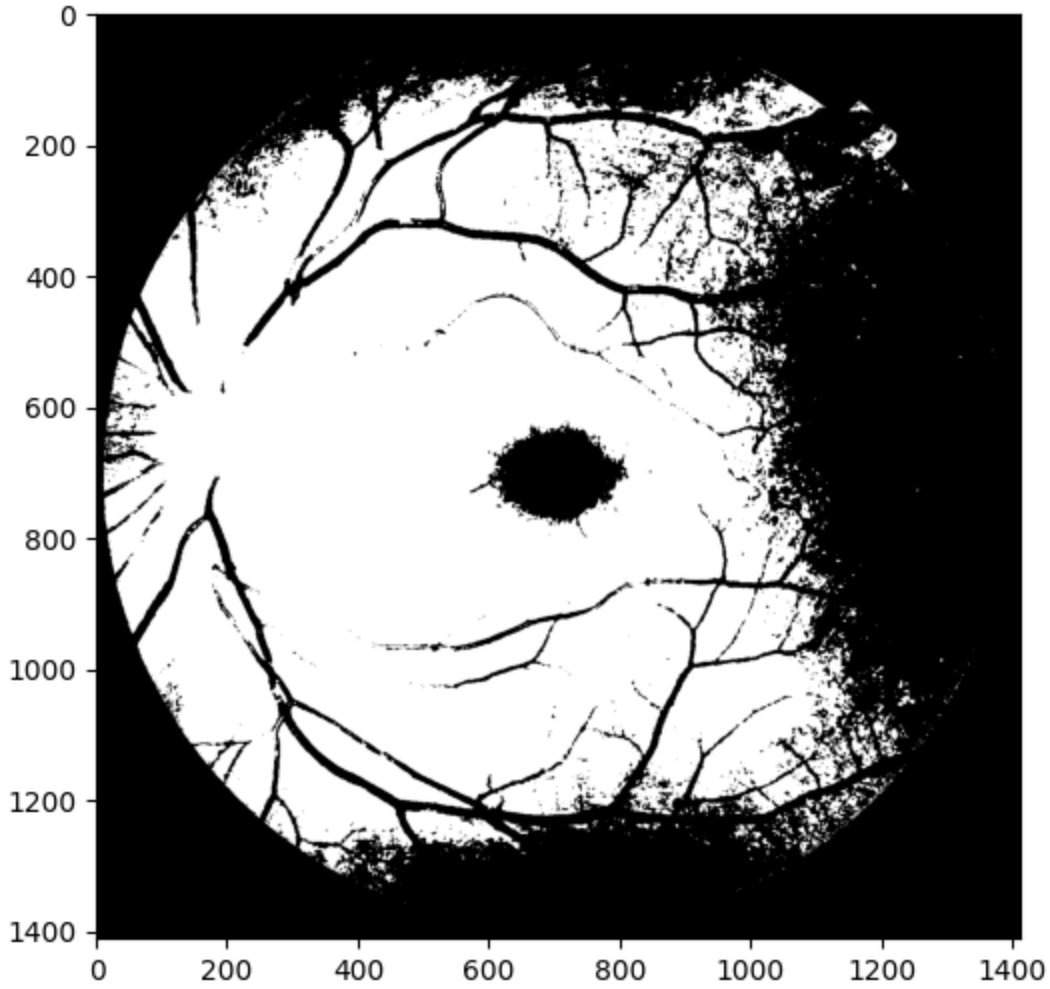
> **Answers:**
>
> 1. Yes, but just a few. The differences in color provide a pixel value range open enough to show those characteristics that are more prominent in the image.
> 2. I did not obtain useful images from the transformation of the requested equalized grey images (retina_gray_scale_equalized[0] and retina_gray_scale_equalized[1]).

In the foregoing exercise, the threshold for the decision classifying pixel values as true or false, $\{0, 1\}$ was set manually by trial and error. There are numerous algorithms which have been devised for finding thresholds. In general, these algorithms attempt to find an optimal threshold using various measures. Ideally, these algorithms search for a low frequency point in the pixel value histograms which can be used to divide the values.

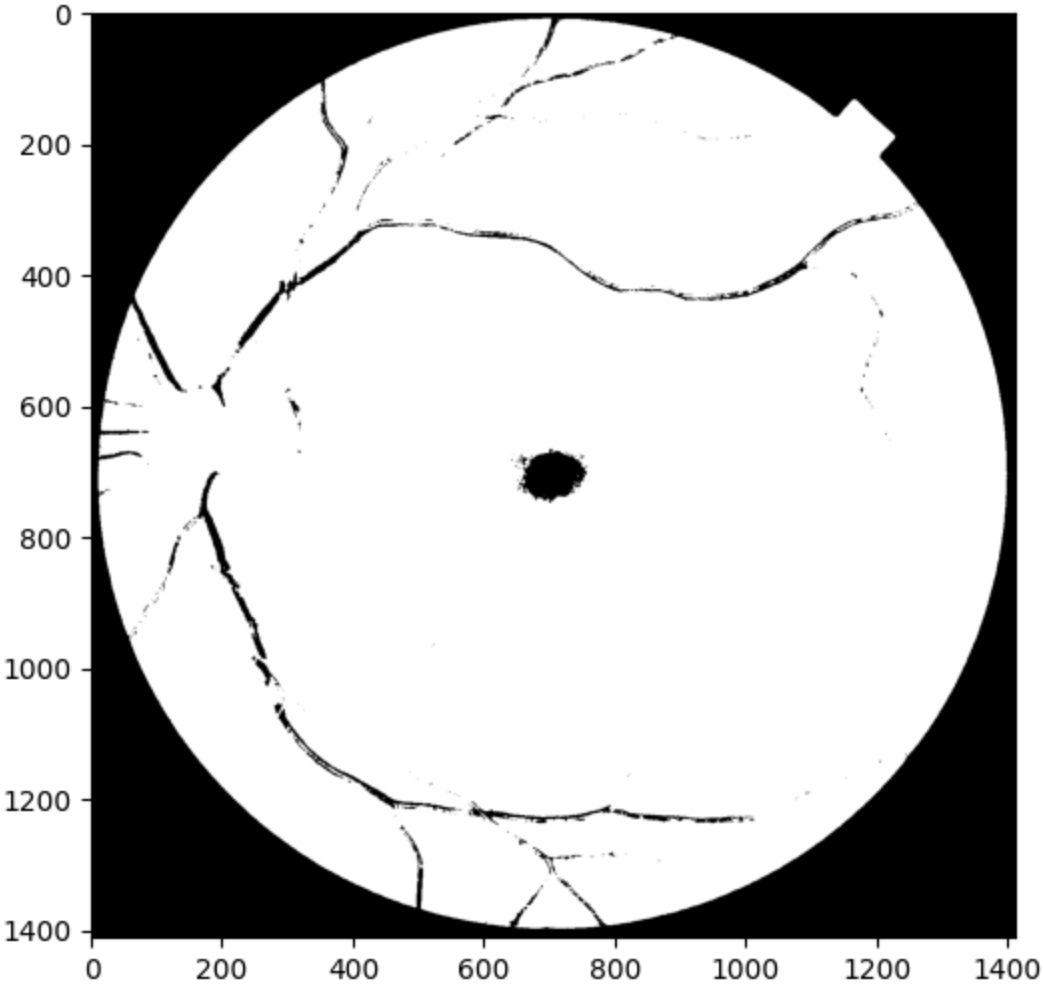
Exercise 1-8: We can create a binary image using one of the many established algorithms to compute a threshold. In this case Otsu's threshold algorithm. Use this function to find a threshold, apply the threshold to the equalized gray-scale image to compute a binary image, and plot the result.

In [ ]:
```
## Put you code below

for i in [retina_gray_scale_equalized[0], retina_gray_scale_equalized[1]]:
    threshold_value = skimage.filters.threshold_otsu(i)

    print('the threshold value = {:.3f}'.format(threshold_value))
    retina_binary_equalized = i >= threshold_value
    print('Image dimensions = ' + str(retina_binary_equalized.shape))
    plot_grayscale(retina_binary_equalized)
```

```
the threshold value = 136.000
Image dimensions = (1411, 1411)
```



```
the threshold value = 65.000
Image dimensions = (1411, 1411)
```



> How does this binary image compare to the ones computed with the threshold found by trail-and-error, and why?
> **End of exercise.**

> **Answer:**

> This method works much better because it takes into account the values of the image to be transformed.

Inversion of Images

For some machine vision algorithms it is easier or more effective to work with the **inverse image** or **negative** of the image. The concept is simple. Pixel values are generally restricted to a range like $\{0-255\}$ for unsigned integer representation or $\{0.0-1.0\}$ for a floating point image. The given an intensity $P_{i,j}$ of the ijth pixel, the inverted intensity, $I_{i,j}$, is then:

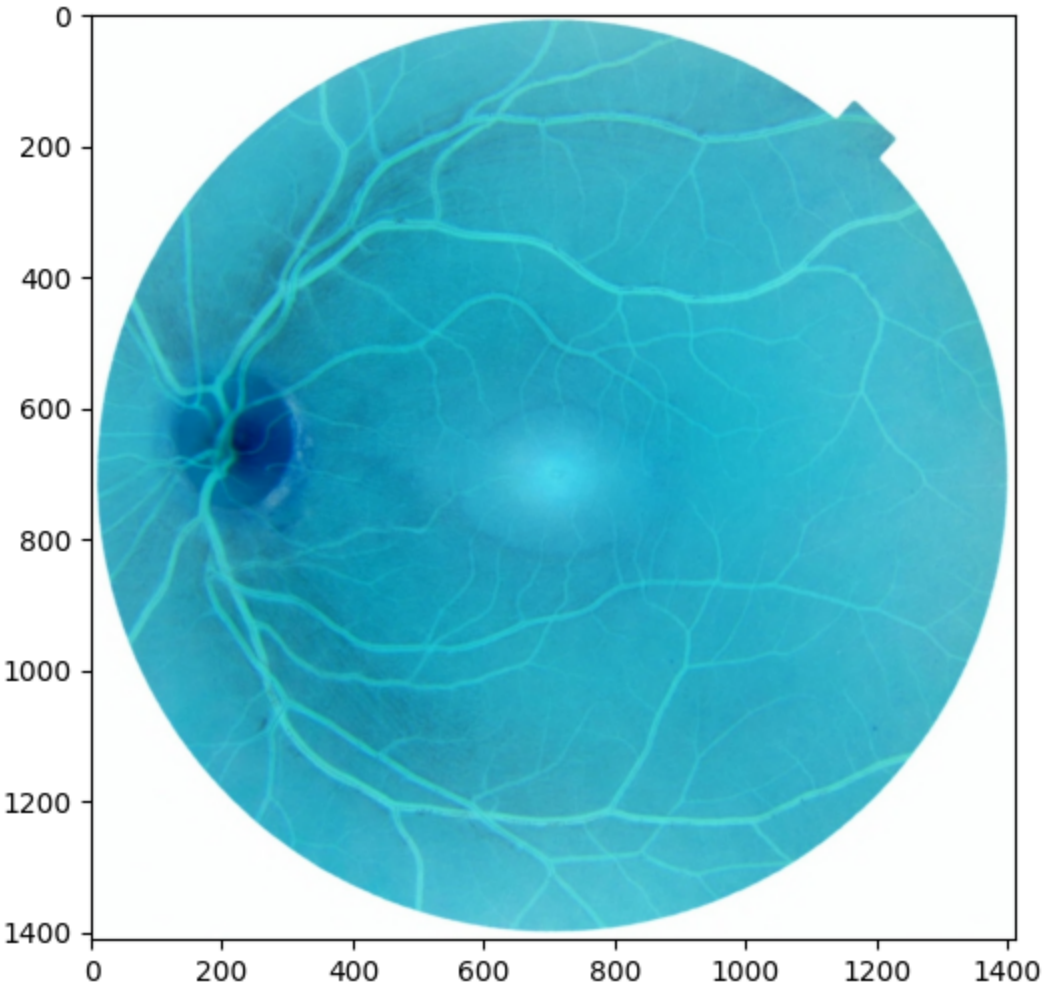
$$I_{i,j} = max\left[P\right] - P_{i,j}$$

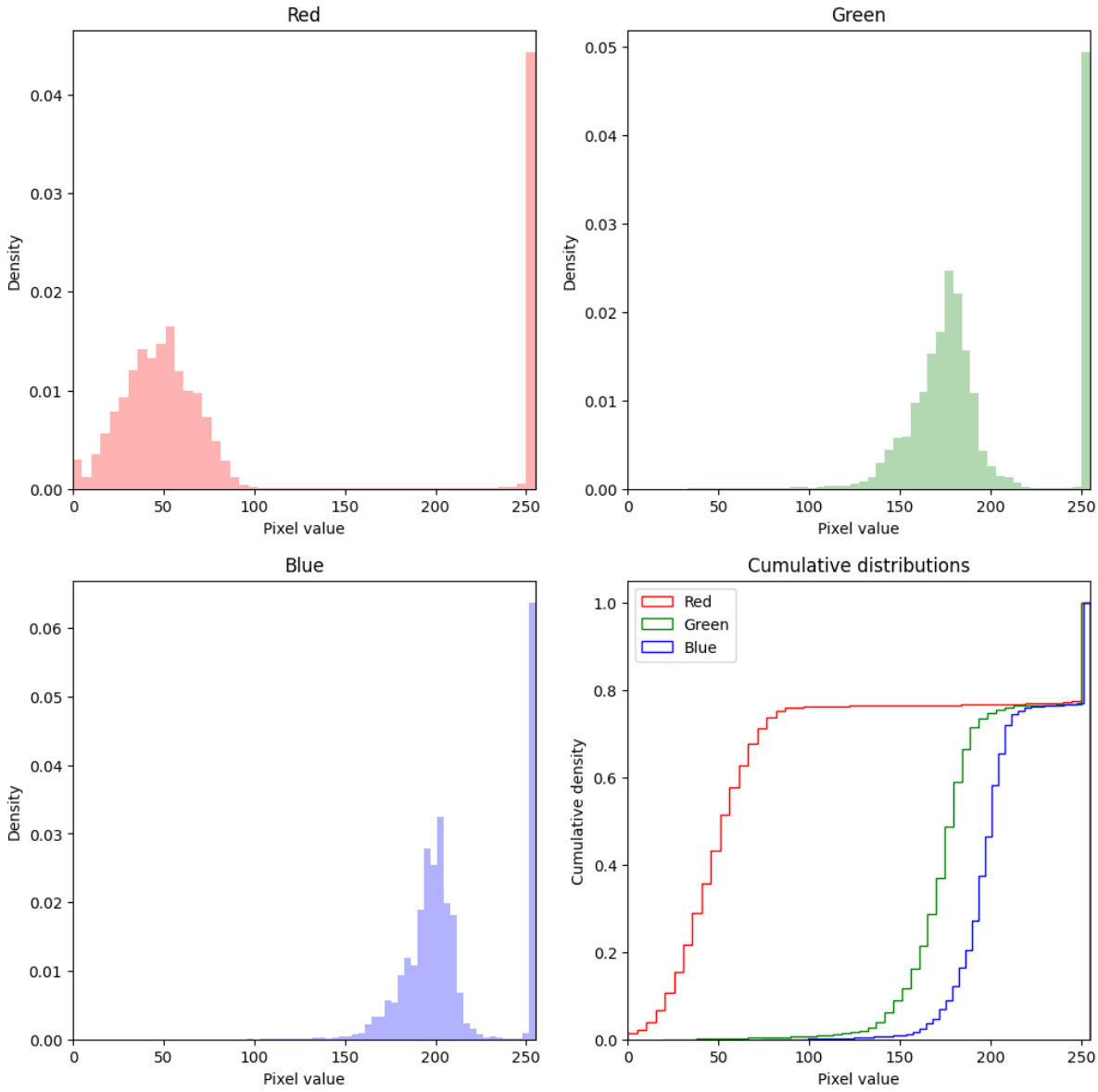
Where, $max\left[P\right]$ is the largest value the representation of the image allows, typically 255 or 1.0.

> **Exercise 1-9:** You will now write a function named `invert_image` that will perform image inversion on both 3-channel and gray-scale images. Make sure you find the correct maximum value for the data type of the image, 255 for `unit8` or 1.0 float.
>
> Now, apply your function to the original color retina image and display the image along with the density plot.

In [ ]:
```
def invert_image(img):
    '''
    Function to invert an image, or create the negative.
        Args:
            img - a 2d gray scale or color image
        Returns:
            Inverse of the image
    '''
    ## Make sure to use a copy to prevent weird bugs that
    ## that are nearly impossible to track down

    img = np.copy(img)

    ## Put your code below
    max_value = 255

    return max_value - img


inverted_retina = invert_image(retina_image)
print(inverted_retina.shape)

fig, ax = plt.subplots(figsize=(6, 6))
_=ax.imshow(inverted_retina)
plot_image_distributions(inverted_retina)
```

```
(1411, 1411, 3)
```


0
200
400
600
800
1000
1200
1400
0
200
400
600
800
1000
1200
1400



> Answer the following questions:
>
> 1. Compare the distribution of the pixel values for the three color channels of the inverted image with the distributions for the original image. Do the distributions for the inverted image make sense given the original values and why?
> 2. Do you think the color of the 3-channel inverted image is correct and why?

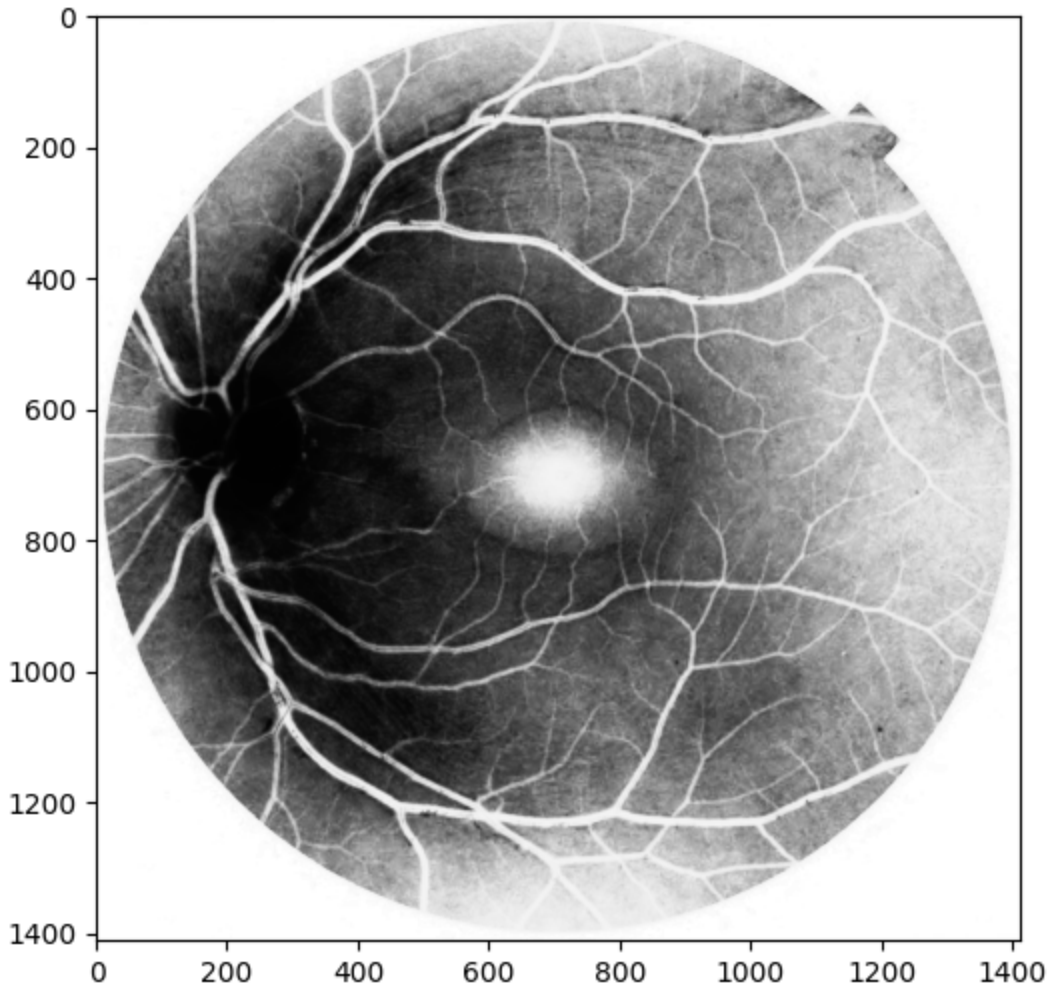
> Next, apply your function to the adaptive histogram equalized gray-scale retina image and display the image along with the distribution plot.

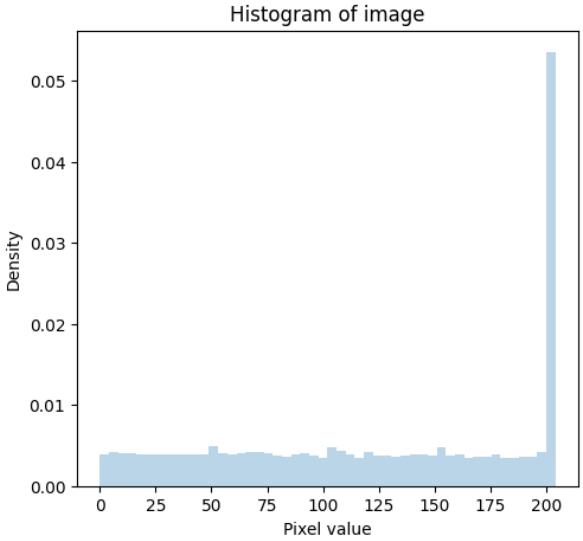
> **Answers:**

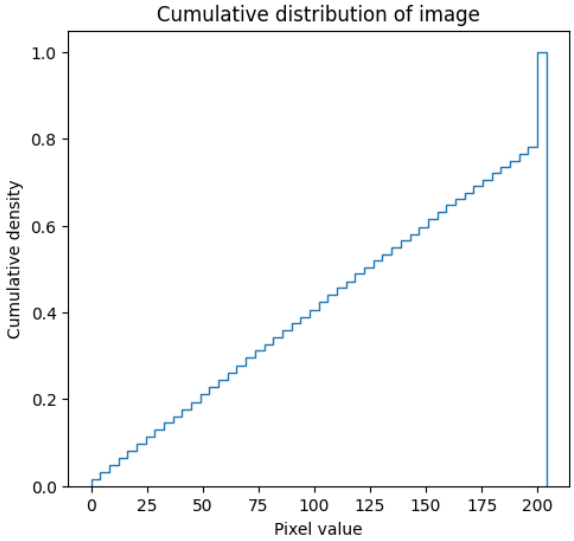
> 1. Yes, they do make sense. Inverting the pixel values within the range (0, 255) means making any original value equals to (255 - (original value)), which is what is represented in the pixel value distribution charts.
> 2. Yes, it is. The inverted image is colored with the RGB combination of the inverse of each of the original values.

In [ ]:
```
for i in [retina_gray_scale_equalized[0], retina_gray_scale_equalized[1]]:
    inverted_retina_grayscale = invert_image(i)
    print(inverted_retina_grayscale.shape)
    plot_grayscale(inverted_retina_grayscale)
    plot_gray_scale_distribution(inverted_retina_grayscale)
```

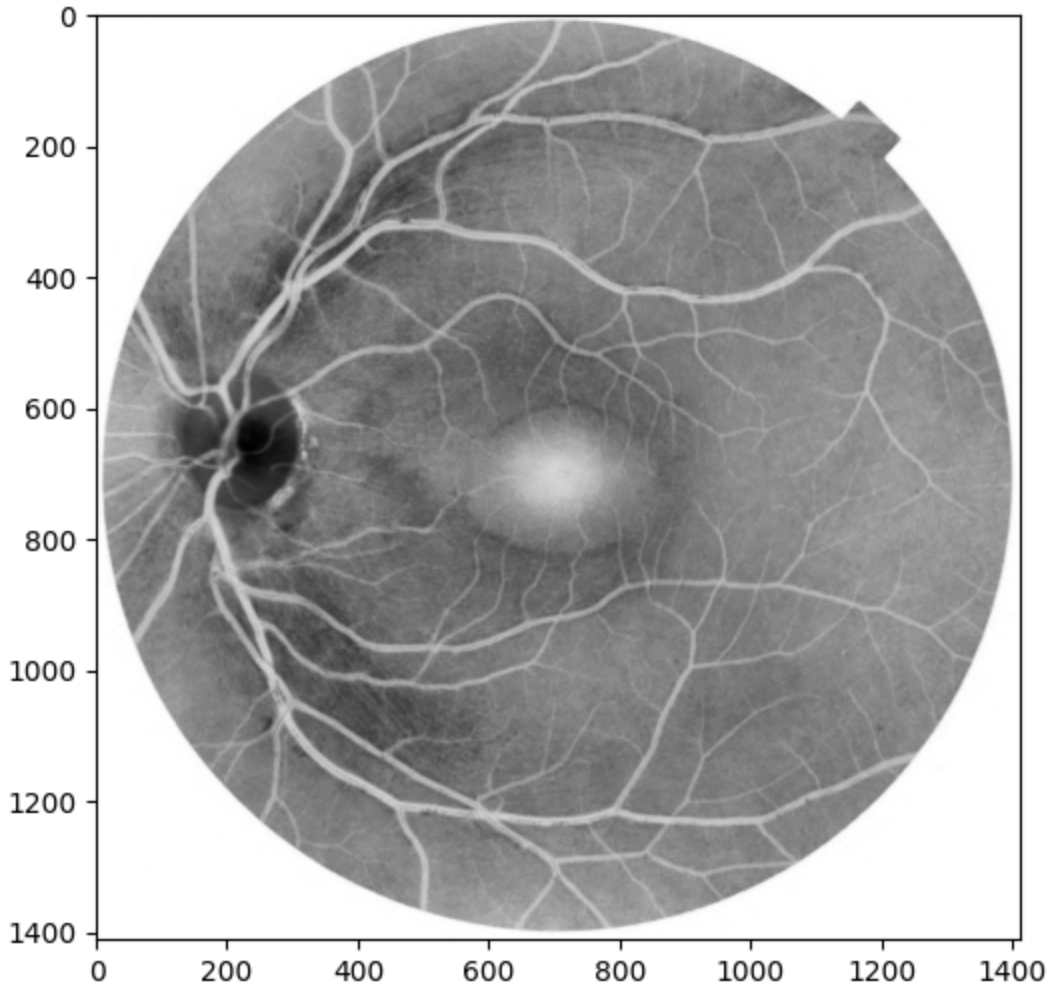
```
(1411, 1411)
```

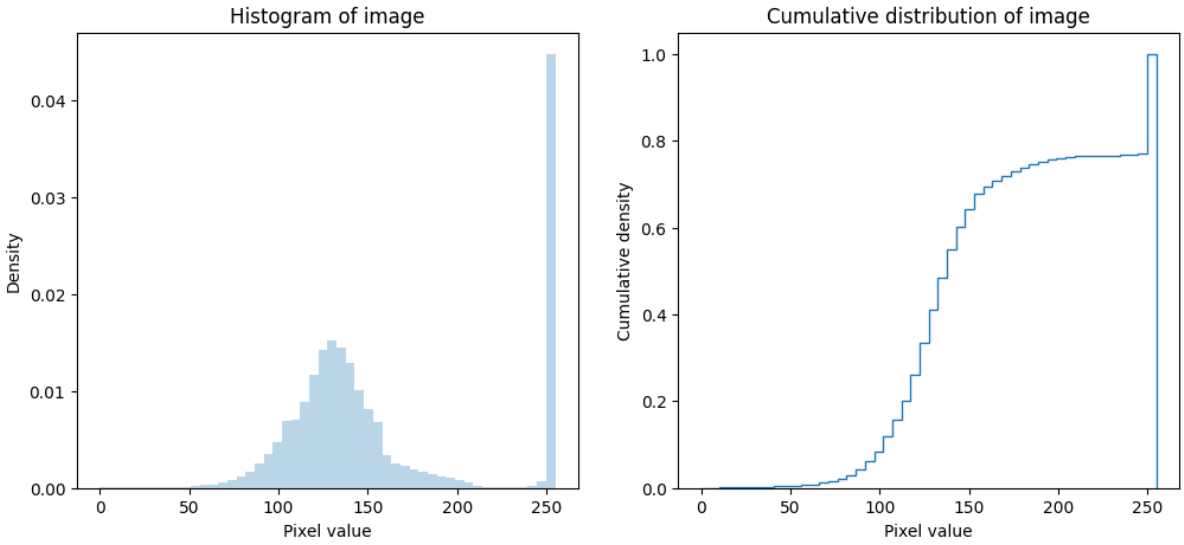



Histogram of image
Density
Pixel value
0.00
0.01
0.02
0.03
0.04
0.05
0 25 50 75 100 125 150 175 200


Cumulative distribution of image
Cumulative density
Pixel value
0.0
0.2
0.4
0.6
0.8
1.0
0 25 50 75 100 125 150 175 200

```
(1411, 1411)
```


0
200
400
600
800
1000
1200
1400



3. The difference in pixel value distributions between the inverted gray scale and original adaptive histogram distributions are subtle. What key difference can you identify?

End of exercise.

Answers:

3. The shape of the distributions is similar although pixel values are mirrored (the graph is horizontally mirrored). Density for the first original image ranges between 0 and ~1, while density for the first inverted image ranges between 0 and ~0.005. Similar behavior occurs for the second image.

Sampling and resizing images

For many computer vision processes the dimensions of an image must be transformed. We have already explored removing the multi-channel color dimension from an image to form the gray-scale image. Now, we will investigate transformation the pixel row and column dimensions of an image. There are two options:

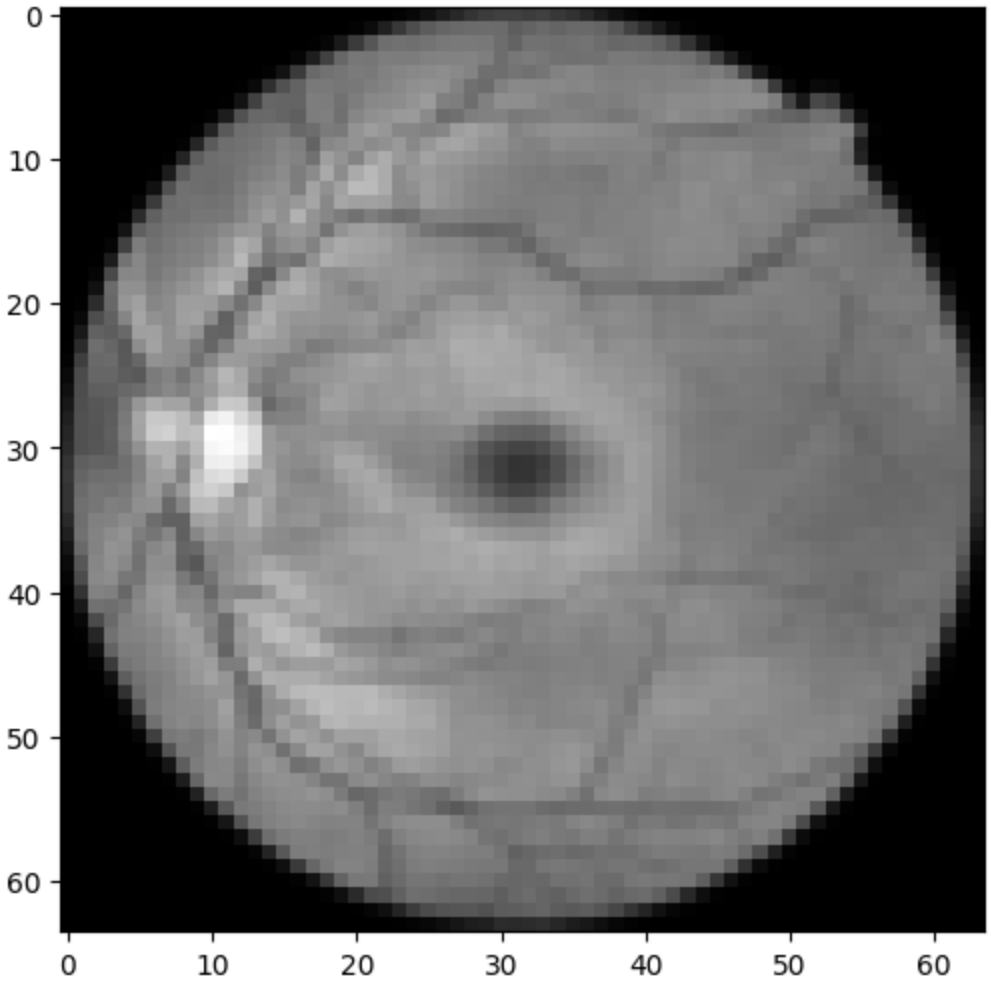
1. **Down-sample:** A down sampled image has a reduced number of pixels. If the multiple between the pixel count of the original image and the down-sampled image an even number, sampled pixel values are used. Otherwise, interpolation is required for arbitrary multiples. Inevitably, down-sampling will reduce the resolution of the image, and fine details will be lost.
2. **Up-sample:** The number of samples can be increased by interpolation between the pixel values. The interpolated values fill in the values of the new pixel. If the pixel

count of the up-sampled image is not an even multiple of the original image most of the values will be interpolated. While up-sampling can increase the number number of the pixels, this process cannot increase the resolution of an image.

> **Exercise 1-10:** You will now resize the adaptive histogram equalized gray-scale image. Using the skimage.transform.resize function, do the following:
>
> 1. Down-sample the image to dimension $(64, 64)$. Print the dimensions and display the resulting image.
> 2. Up-sample the down-sampled image to dimension $(1024, 1024)$. Print the dimensions and display the resulting image.

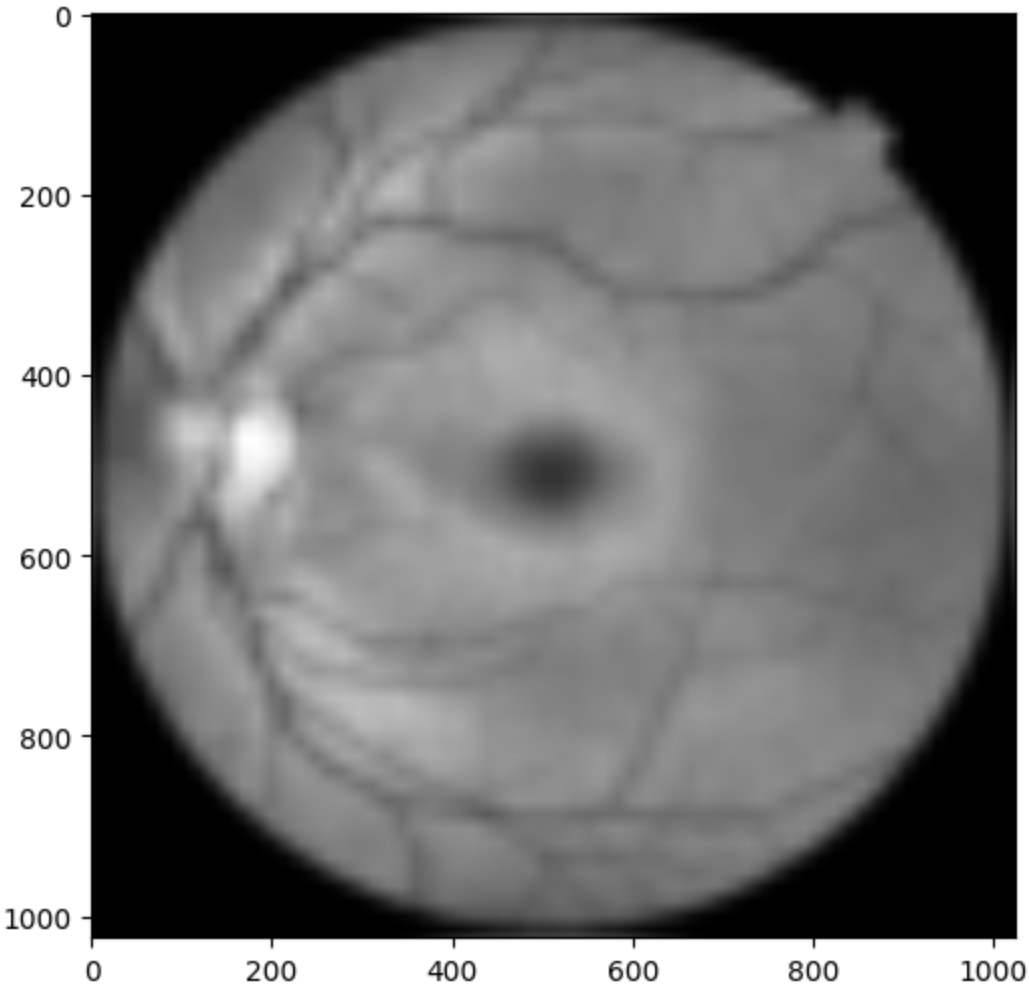
In [ ]:
```
## down sample the image
## Put you code below

# 1. Downsample image
retina_downsampled = skimage.transform.resize(retina_gray_scale_equalized[1]
print('Adaptive histogram equalized gray-scale image - Downsampled')
print(retina_downsampled.shape)
plot_grayscale(retina_downsampled)
```

```
Adaptive histogram equalized gray-scale image - Downsampled
(64, 64)
```



In [ ]:
```
## Up-sample the image
## Put you code below

# 2. Upsample image
retina_upsampled = skimage.transform.resize(retina_downsampled, (1024, 1024)
print('Downsampled image - Upsampled')
print(retina_upsampled.shape)
plot_grayscale(retina_upsampled)
```

```
Downsampled image - Upsampled
(1024, 1024)
```



> Notice the changes in resolution of the down-sampled and up-sampled images.
>
> 1. How is the reduction in resolution of the $(64, 64)$ image exhibited?
> 2. Does up-sampling to $(1024, 1024)$ restore the resolution of the image or simply blur the 'pixelation' visible in the $(64, 64)$ image?

> **Answers:**

> 1. There is a significant loss of resolution and pixelation, where the loss of resolution also implies loss of details.
> 2. No. It just blurs the pixelation of the downsampled image, but lost details are not recovered.

Sampling and Aliasing in Images

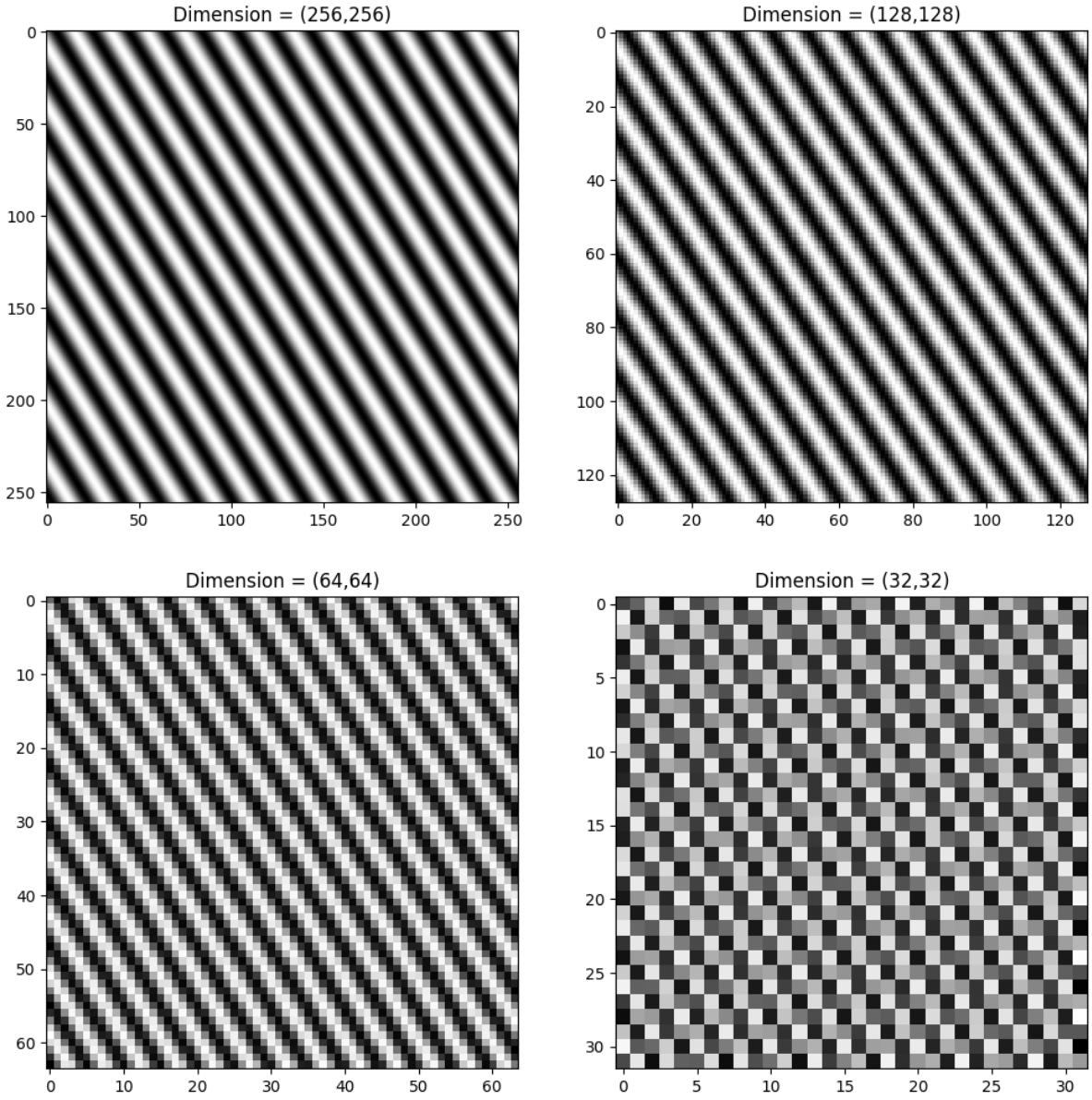
As should be clear from the foregoing, the digital images we work with for computer vision are discretely sampled in the 2-dimensional plane. The discrete pixel values $v_{\mathbf{x}}$ are

the result of this sampling. This discrete sampling limits the **spatial resolution** which can be captured in the image. If the samples are spaced too far apart, aliasing will occur. For sinusoidal components of the image the Nyquist Shannon sampling theorem the sampling frequency must be at least 2 times the frequency of this component. The sampling rate of 2 times the highest frequency component is known as the **Nyquist Frequency**. Sampling below the Nyquist frequency leads to aliasing.

We can demonstrate the concept of aliasing with a simple example. The example is based on an initial image and three down-sampled versions:

1. The initial image has diagonal slashes with sinusoidal amplitudes and dimension $(256, 256)$.
2. The image is down-sampled to dimension $(256, 256)$.
3. The image is down-sampled to dimension $(128, 128)$.
4. The image is down-sampled to dimension $(64, 64)$.

In [ ]:
```
dim = 256
x = np.arange(0, dim * dim, 1)
sin_image = 1.0 - 0.5 * np.sin(np.divide(x, math.pi)).reshape((dim, dim))

fig, ax = plt.subplots(2,2, figsize = (12, 12))
ax = ax.flatten()
for i, dim in enumerate([256, 128, 64, 32]):
    sampled_image = resize(sin_image, (dim, dim))
    _ = ax[i].imshow(sampled_image, cmap = plt.get_cmap('gray'))
    _ = ax[i].set_title('Dimension = (' + str(dim) + ',' + str(dim) + ')')
```



Examine the images above and notice the following:

1. The $(128, 128)$ down-sampled image retains the characteristics of the initial image. Look carefully, you can see a slight blurring.
2. The $(64, 64)$ down-sampled image retains the sinusoidal slash structure. Coarse pixelation is now quite evident and the sampling is very close to the Nyquist frequency.
3. The $(32, 32)$ down-sampled image does not resemble the initial image at all, exhibiting significant aliasing. Run your eye side to side and up and down on the image. You may see patterns that are not representative of the original image. Such false patterns are a common artifact arising from aliasing.

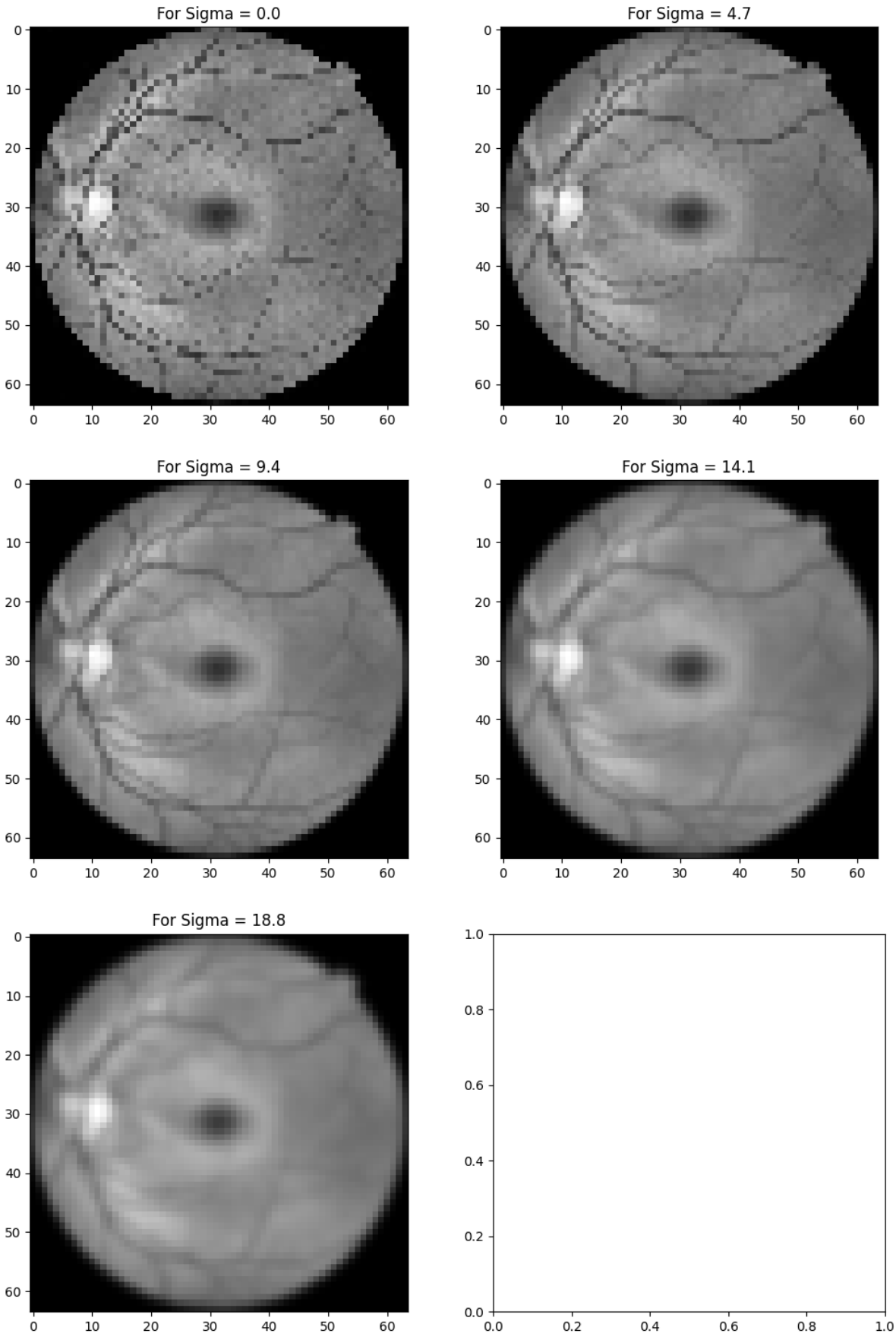
How can aliasing be prevented? A filter can remove the high frequency components of the image which would lead to the aliasing. A common approach is to use a Gaussian

filter. This filter removes high frequency components and has the effect of blurring the image.

> **Exercise 1-11:** You will now investigate how filtering can be applied to prevent aliasing. The skimage.transform.resize function applies a Gaussian filter to prevent aliasing by default. The standard devision of the Gaussian filter, or filter span, can be set to adjust the bandwidth of the filter.
> In this exercise you will resample the adaptive histogram equalized gray-scale retina image using the skimage.transform.resize function with the `anti_aliasing=False` argument. You will use the skimage.filters.gaussian to limit the bandwidth of the image. Do the following to compare the results of different filter bandwidths:
>
> 1. Computer a scale factor, $sf = \sqrt{\frac{original\ dimension}{reduced\ dimension}}$.
> 2. Apply the Gaussian filter with $sigma = mutltiplier * sf$ for multiplier in $[0, 1, 2, 3, 4]$ (`range(5)`) and resize the image to $(64, 64)$ pixels.
> 3. For each value of sigma display the image with a title indicating the value of sigma. You may find interpretation easier if you plot the images on a 3×2 grid.

In [ ]:
```
## Put you code below
## down sample the image

fig, ax = plt.subplots(3,2, figsize=(12, 18))
ax = ax.flatten()

old_size = retina_gray_scale_equalized[1].shape[0]
print('Old size = ' + str(old_size))

new_size = 64
print('New size = ' + str(new_size))

scale_factor = np.sqrt(old_size / new_size)
print('Scale factor = ' + str(scale_factor))

for s in range(5):
    ## Add the missing code

    sigma = s * scale_factor

    retina_gaussian = skimage.filters.gaussian(retina_gray_scale_equalized[1
    retina_decimated = resize(retina_gaussian, (64, 64), anti_aliasing = Fal

    ax[s].set_title('For Sigma = ' + str(round(sigma, 1)))
    _ = ax[s].imshow(retina_decimated, cmap = plt.get_cmap('gray'))
```

```
Old size = 1411
New size = 64
Scale factor = 4.695409992748237
```



> Answer the following questions.
>
> 1. How does the aliasing change with increasing sigma, decreasing bandwidth? Is this the behavior you expect and why?
> 2. How does the blurring of the image change with increasing sigma, decreasing bandwidth? Is this the behavior you expect and why?
>
> **End of exercise.**

> **Answers:**

> 1. Yes, this is the behavior I expected because aliasing decreases when sigma increases. This aliasing makes the image look better as sigma increases, but just until certain point. In our example, up to sigma = 9.4. Crossing this point information starts getting lost.
> 2. Yes, this is the expected behavior. Blurring increases as sigma increases. The effects are explained in point 1.