ijyudwpmq

October 8, 2023

0.0.1 Vessel Enhancement Techniques

1. Contrast Stretching: Enhancing the contrast of the image to make vessels more visible.
2. Histogram Equalization: Spreading out the pixel values to improve contrast.
3. Gabor Filtering: A frequency-based method to highlight vessels.
4. Frangi Filter: Specifically designed for vessel enhancement.
5. Hessian-Based Filtering: Detecting vessel-like structures.
6. Top-hat Transform: Morphological operation to enhance elongated structures. ____ ### Evaluation Techniques To prove that the visibility of tiny blood vessels is better than the original image, you can use the following quantitative evaluation metrics:
7. Structural Similarity Index (SSIM): Measures the similarity between the enhanced image and the original image (0-1):
   - A score of 1 indicates a perfect match, meaning the enhanced image is identical to the original image in terms of structure.
   - A score closer to 1 suggests a high similarity and indicates that the enhanced image effectively preserves structural information.
   - A score around 0 suggests that the images are largely dissimilar, with little structural overlap.
   - A negative score indicates significant dissimilarity. #
8. Peak Signal-to-Noise Ratio (PSNR): Measures the quality of the enhanced image compared to the original image (0-100):
   - Higher PSNR values (in decibels, dB) indicate lower noise or error, which implies a higher quality image.
   - A higher PSNR suggests that the enhancement process has preserved image details well.
   - Lower PSNR values indicate that more noise or distortion has been introduced during enhancement.
   - The PSNR score range is typically between 0 and 100 dB, with higher values indicating better quality. In practice, PSNR values above 30 dB are generally considered to be of high quality, while values below 20 dB are considered to be of poor quality. However, the acceptable range of PSNR values can vary depending on the specific application and the nature of the image or video being evaluated. #
9. Mean Square Error (MSE): Measures the average squared difference between pixel values in the two images (0-inf):
   - A lower MSE value indicates less difference between the original and enhanced images.
   - It measures the average magnitude of errors, so lower MSE values indicate better image similarity.
   - The MSE score range is typically between 0 and infinity, with lower values indicating better quality. In practice, MSE values below 1 are generally considered to be of high quality, while values above 10 are considered to be of poor quality. However, the ac-

ceptable range of MSE values can vary depending on the specific application and the nature of the images being evaluated. #

10. Receiver Operating Characteristic (ROC) Curve: Evaluates the ability of the method to discriminate between vessels and non-vessel regions (0-1):
    - A steeper ROC curve that approaches the upper-left corner suggests better discrimination ability.
    - The area under the ROC curve (AUC) quantifies the overall discrimination performance. An AUC of 1 indicates perfect discrimination, while 0.5 suggests random guessing.
    - The score range of the ROC curve is between 0 and 1, with higher values indicating better performance. The area under the ROC curve (AUC) is a commonly used metric to evaluate the performance of a classifier. The AUC score range is also between 0 and 1, with higher values indicating better performance. An AUC score of 0.5 indicates that the classifier is performing no better than random chance, while an AUC score of 1.0 indicates perfect classification performance. #
11. F1-score, Precision, and Recall: Evaluate the performance of vessel detection (0-1):
    - F1-score ranges from 0 to 1, where higher values indicate better performance.
    - Precision measures the accuracy of positive predictions, while recall measures the ability to find all positive instances.
    - A higher F1-score indicates a better balance between precision and recall.
    - The F1-score is the harmonic mean of the precision and recall, and is a measure of the balance between the two metrics. It is expressed as a value between 0 and 1, with higher values indicating better performance. An F1-score of 1 indicates perfect precision and recall, while a score of 0 indicates poor performance.
    - Precision is the proportion of true positive cases among all positive predictions made by the model. It is expressed as a value between 0 and 1, with higher values indicating better performance. A precision score of 1 indicates that all positive predictions made by the model were correct, while a score of 0 indicates that all positive predictions were incorrect.
    - Recall is the proportion of true positive cases that were correctly identified by the model among all actual positive cases. It is expressed as a value between 0 and 1, with higher values indicating better performance. A recall score of 1 indicates that all actual positive cases were correctly identified by the model, while a score of 0 indicates that none of the actual positive cases were identified by the model.

[13]:
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from os.path import join
from skimage.filters import frangi
from skimage import exposure as ex

%matplotlib inline
```

```
image_paths = [join('assets', '16_left.jpeg'), join('assets', '16_right.jpeg'),␣
 ↪join('assets', '20_left.jpeg'), join('assets', '20_right.jpeg'),␣
 ↪join('assets', '40_left.jpeg'), join('assets', '40_right.jpeg'),␣
 ↪join('assets', '65_left.jpeg'), join('assets', '65_right.jpeg'),␣
 ↪join('assets', '156_left.jpeg'), join('assets', '156_right.jpeg')]
scale = 300

def plotImages(images):
    plt.figure(figsize=(10, 10))
    columns = 2
    rows = int(np.ceil(len(images) / columns))
    for i, image in enumerate(images):
        plt.subplot(rows, columns, i + 1)
        plt.imshow(image / 255.0)

def scaleRadius(image, scale_factor):
    horizontal_cross_section = image[image.shape[0] // 2, :, :].sum(1)
    pupil_radius = (horizontal_cross_section > horizontal_cross_section.mean() /
 ↪ 10).sum() / 2
    scaling_factor = scale_factor * 1.0 / pupil_radius
    resized_image = cv2.resize(image, (0, 0), fx=scaling_factor,␣
 ↪fy=scaling_factor)

    return resized_image

def evaluateEnhancement(original_img, enhanced_img, ground_truth_img,␣
 ↪threshold):
    original_gray = cv2.cvtColor(original_img, cv2.COLOR_RGB2GRAY)
    enhanced_gray = cv2.cvtColor(enhanced_img, cv2.COLOR_RGB2GRAY)
    ground_truth_gray = cv2.cvtColor(ground_truth_img, cv2.COLOR_RGB2GRAY)

    # Binarize the images using the threshold value
    original_binary = cv2.threshold(original_gray, threshold, 255, cv2.
 ↪THRESH_BINARY)[1]
    enhanced_binary = cv2.threshold(enhanced_gray, threshold, 255, cv2.
 ↪THRESH_BINARY)[1]
    ground_truth_binary = cv2.threshold(ground_truth_gray, threshold, 255, cv2.
 ↪THRESH_BINARY)[1]

    # Calculate F1-score, Precision, and Recall
    tp = np.sum(np.logical_and(enhanced_binary == 255, ground_truth_binary ==␣
 ↪255))
    fp = np.sum(np.logical_and(enhanced_binary == 255, ground_truth_binary ==␣
 ↪0))
    fn = np.sum(np.logical_and(enhanced_binary == 0, ground_truth_binary ==␣
 ↪255))
```

```
    precision = tp / (tp + fp) # if (tp + fp) > 0 else 0
    recall = tp / (tp + fn) # if (tp + fn) > 0 else 0
    f1_score = 2 * (precision * recall) / (precision + recall) # if (precision␣
↪+ recall) > 0 else 0

    # Calculate PSNR
    mse = np.mean((original_img - enhanced_img) ** 2)
    psnr = 10 * np.log10((255 ** 2) / mse)

    return f1_score, precision, recall, psnr
```

`plotImages` function:

1. `def plotImages(images)::` This line defines the function and takes a list of images as input.
2. `plt.figure(figsize=(10, 10))`: This line creates a new figure with a size of 10x10 inches using Matplotlib.
3. `columns = 2`: This line sets the number of columns in the grid to 2.
4. `rows = int(np.ceil(len(images) / columns))`: This line calculates the number of rows needed to display all the images in the list. It first finds the length of the list using the `len` function, then divides it by the number of columns and rounds up to the nearest integer using the `np.ceil` function.
5. `for i, image in enumerate(images)::` This line starts a loop that iterates over each image in the list and assigns a unique index `i` to each image.
6. `plt.subplot(rows, columns, i + 1)`: This line creates a new subplot in the grid with the given number of rows and columns, and selects the `i+1`th subplot for the current image. The `i+1` is used because subplot indices start from 1, not 0.
7. `plt.imshow(image)`: This line displays the current image in the selected subplot using Matplotlib's `imshow` function.

`scaleRadius` function:

1. `def scaleRadius(image, scale_factor):` - This line defines the function and takes an image and a scale factor as input.
2. `horizontal_cross_section = image[image.shape[0] // 2, :, :].sum(1)` - This line extracts the horizontal cross-section of the image at the center of the vertical axis. It does this by selecting the row at the center of the image (`image.shape[0] // 2`), all columns (`:`), and all color channels (`:`), and then summing the pixel values along the third axis (`sum(1)`).
3. The `sum(1)` function call in the `scaleRadius` function sums the pixel values along the third axis of the image array, which corresponds to the horizontal axis. This is because the `horizontal_cross_section` variable is a 1D array that represents the sum of pixel values along each row of the image at the center of the vertical axis. By summing the pixel values along the horizontal axis, we can obtain a 1D array that represents the intensity profile of the image at the center of the vertical axis. This intensity profile can then be used to estimate the radius of the pupil, as described in the `pupil_radius` calculation.

4. `pupil_radius = (horizontal_cross_section > horizontal_cross_section.mean() / 10).sum() / 2` - This line calculates the radius of the pupil by finding the number of pixels whose sum is greater than one-tenth of the mean sum of all pixels in the horizontal cross-section. It does this by first dividing the mean sum by 10 (`horizontal_cross_section.mean() / 10`), then finding the number of pixels whose sum is greater than this threshold (`(horizontal_cross_section > horizontal_cross_section.mean() / 10)`), and finally dividing the count by 2 to get the radius (`(horizontal_cross_section > horizontal_cross_section.mean() / 10).sum() / 2`).
5. `scaling_factor = scale_factor * 1.0 / pupil_radius` - This line calculates the scaling factor by dividing the given scale factor by the pupil radius.
6. `resized_image = cv2.resize(image, (0, 0), fx=scaling_factor, fy=scaling_factor)` - This line resizes the image using OpenCV's `resize` function, with the scaling factor `scaling_factor` applied to both the horizontal and vertical axes.
7. `return resized_image` - This line returns the resized image.

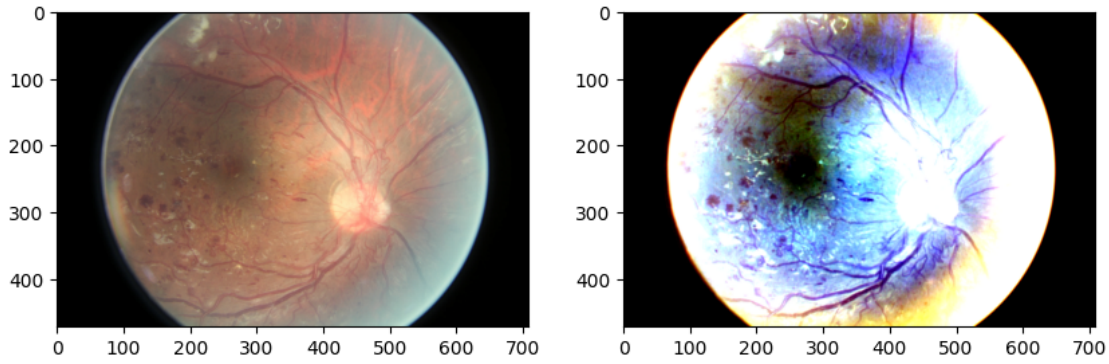
Overall, the `scaleRadius` function scales the radius of the pupil in the input image by a factor that is proportional to the given scale factor and inversely proportional to the pupil radius. This helps to normalize the size of the pupil across different images, which can be useful for further image processing.

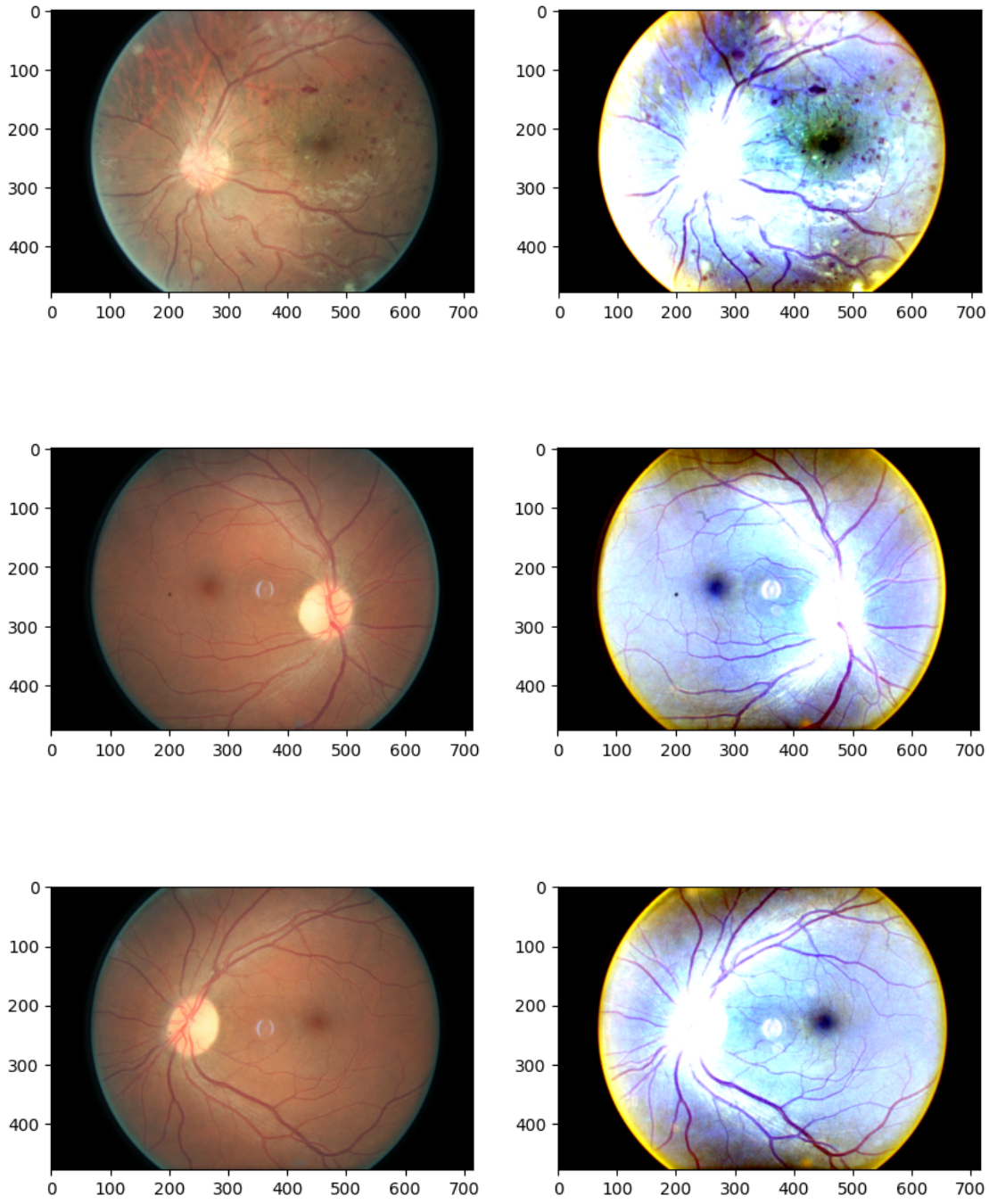
1 Method 1

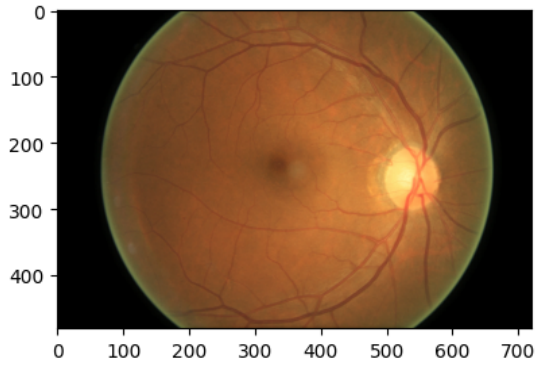
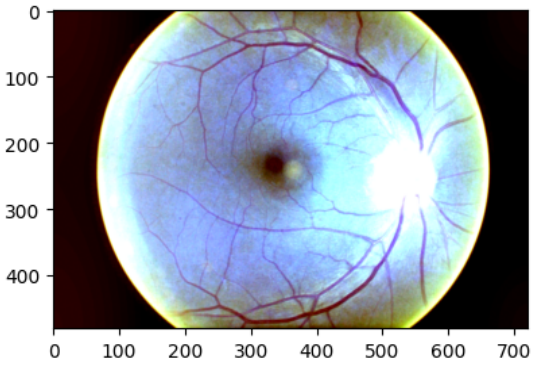
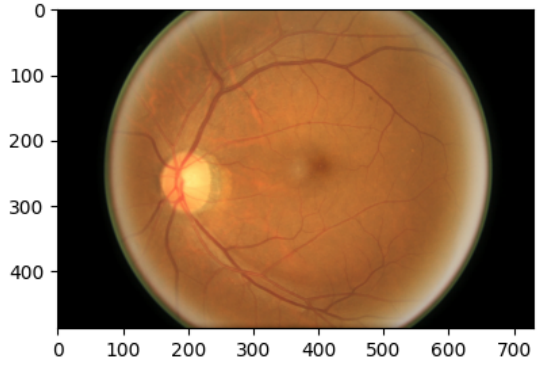
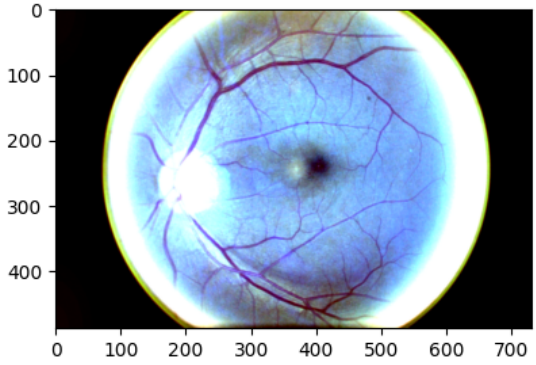
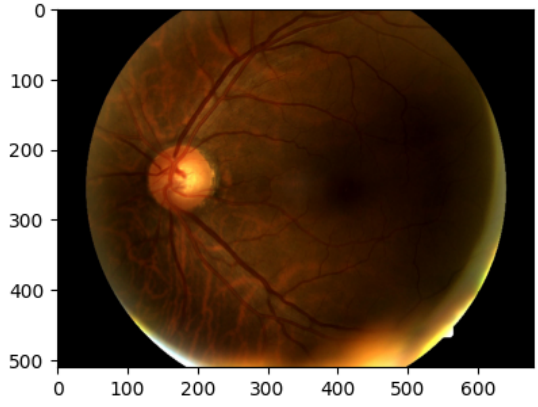
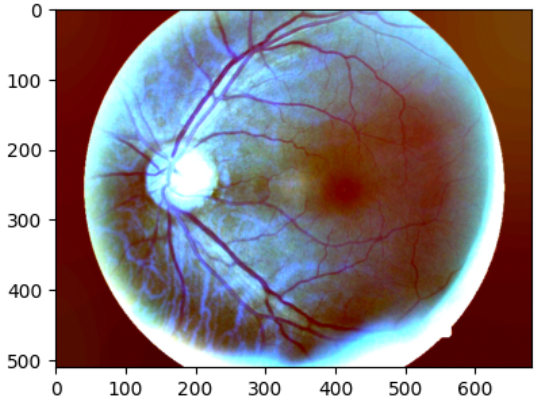
[19]:
```
def enhanceImageMethod1(path, scale, sigmaX=128, alpha=-5, beta=5):
    img = cv2.imread(path)

    # Step 1: Scale the image
    scaled_img = scaleRadius(img, scale)

    # Step 2: Convert to RGB color space
    scaled_img = cv2.cvtColor(scaled_img, cv2.COLOR_BGR2RGB)

    # Step 3: Apply a combination of filters
    # Filter 1: Gaussian Blur with variable sigmaX
    filtered_image1 = cv2.GaussianBlur(scaled_img, (0, 0), sigmaX)

    # Filter 2: Bilateral Filter
    filtered_image2 = cv2.bilateralFilter(scaled_img, 2, 128, 128)

    # Combine filtered images
    enhanced_image = cv2.addWeighted(filtered_image1, alpha, filtered_image2,␣
↪beta, 128)

    # Step 4: Convert back to RGB color space
    enhanced_image = cv2.cvtColor(enhanced_image, cv2.COLOR_BGR2RGB)
```

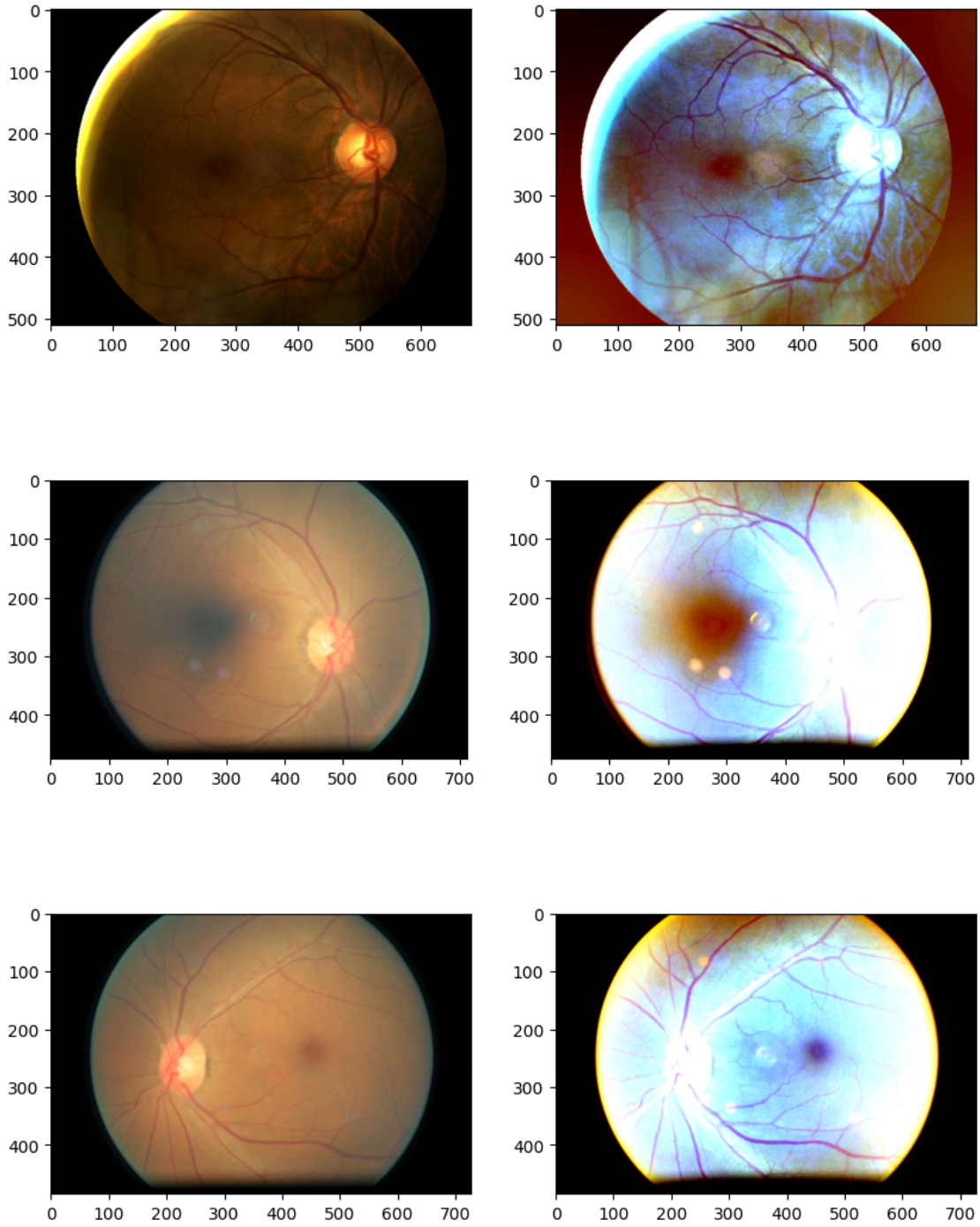
```
    return scaled_img, enhanced_image

counter = 0
for path in image_paths:
    original, enhanced = enhanceImageMethod1(path, scale)

    f1_score, precision, recall, psnr = evaluateEnhancement(original, enhanced,␣
 ↪ground_truth_img=original, threshold=128)
    plotImages([original, enhanced])
    print(f"Image {counter}\t| F1-score: {f1_score:.4f}\t| Precision:␣
 ↪{precision:.4f}\t| Recall: {recall:.4f}\t| PSNR: {psnr:.4f}")
    counter += 1
```

```
Image 0 | F1-score: 0.8146      | Precision: 0.7032     | Recall: 0.9679
| PSNR: 29.0549
Image 1 | F1-score: 0.6121      | Precision: 0.4421     | Recall: 0.9945
| PSNR: 29.2276
Image 2 | F1-score: 0.4066      | Precision: 0.2552     | Recall: 1.0000
| PSNR: 29.2392
Image 3 | F1-score: 0.4483      | Precision: 0.2889     | Recall: 1.0000
| PSNR: 29.2457
Image 4 | F1-score: 0.2651      | Precision: 0.1528     | Recall: 1.0000
| PSNR: 29.0241
Image 5 | F1-score: 0.5550      | Precision: 0.3841     | Recall: 1.0000
| PSNR: 29.1971
Image 6 | F1-score: 0.1284      | Precision: 0.0686     | Recall: 1.0000
| PSNR: 28.2175
Image 7 | F1-score: 0.1784      | Precision: 0.0979     | Recall: 1.0000
| PSNR: 28.3581
Image 8 | F1-score: 0.6007      | Precision: 0.4293     | Recall: 0.9999
| PSNR: 29.2827
Image 9 | F1-score: 0.6358      | Precision: 0.4660     | Recall: 1.0000
| PSNR: 29.1664
```




0
100
200
300
400
0
100
200
300
400
500
600
700



`enhanceImageMethod1` function:

1. The first line applies a Gaussian blur to the `scaled_img` using the `cv2.GaussianBlur()` function. The `sigmaX` parameter controls the amount of blur to apply, with higher values resulting in more blur. In this case, the kernel size is `(0, 0)`, which means that the function will automatically calculate the kernel size based on the given `sigmaX` parameter.

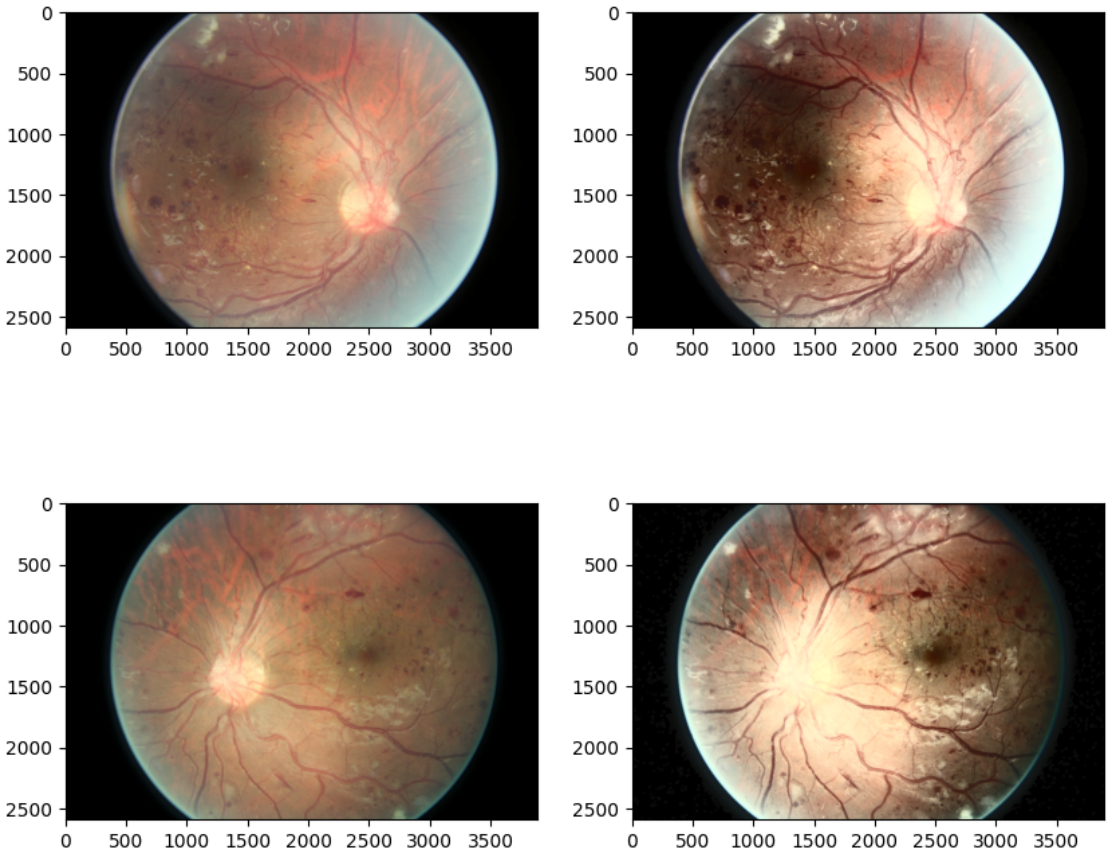
2. The second line applies a bilateral filter to the `scaled_img` using the `cv2.bilateralFilter()` function. The bilateral filter is a non-linear filter that preserves edges while smoothing the image. The first parameter is the input image, the second parameter is the diameter of each pixel neighborhood, and the third and fourth parameters are the filter sigma values in the color space and coordinate space, respectively.
    - The bilateral filter is useful for removing noise from an image while preserving the edges and details of the image. It is often used in image processing applications such as image denoising, image enhancement, and image segmentation. The bilateral filter can be computationally expensive, especially for large images or large filter sizes, but it is effective at reducing noise while preserving image features. #
3. The third line combines the filtered images using the `cv2.addWeighted()` function. This function applies a weighted sum of the two images, with the `alpha` and `beta` parameters controlling the weights of the two images. The `10` parameter is a scalar value added to the result. The resulting image is a combination of the two filtered images, with the weights controlled by the `alpha` and `beta` parameters.

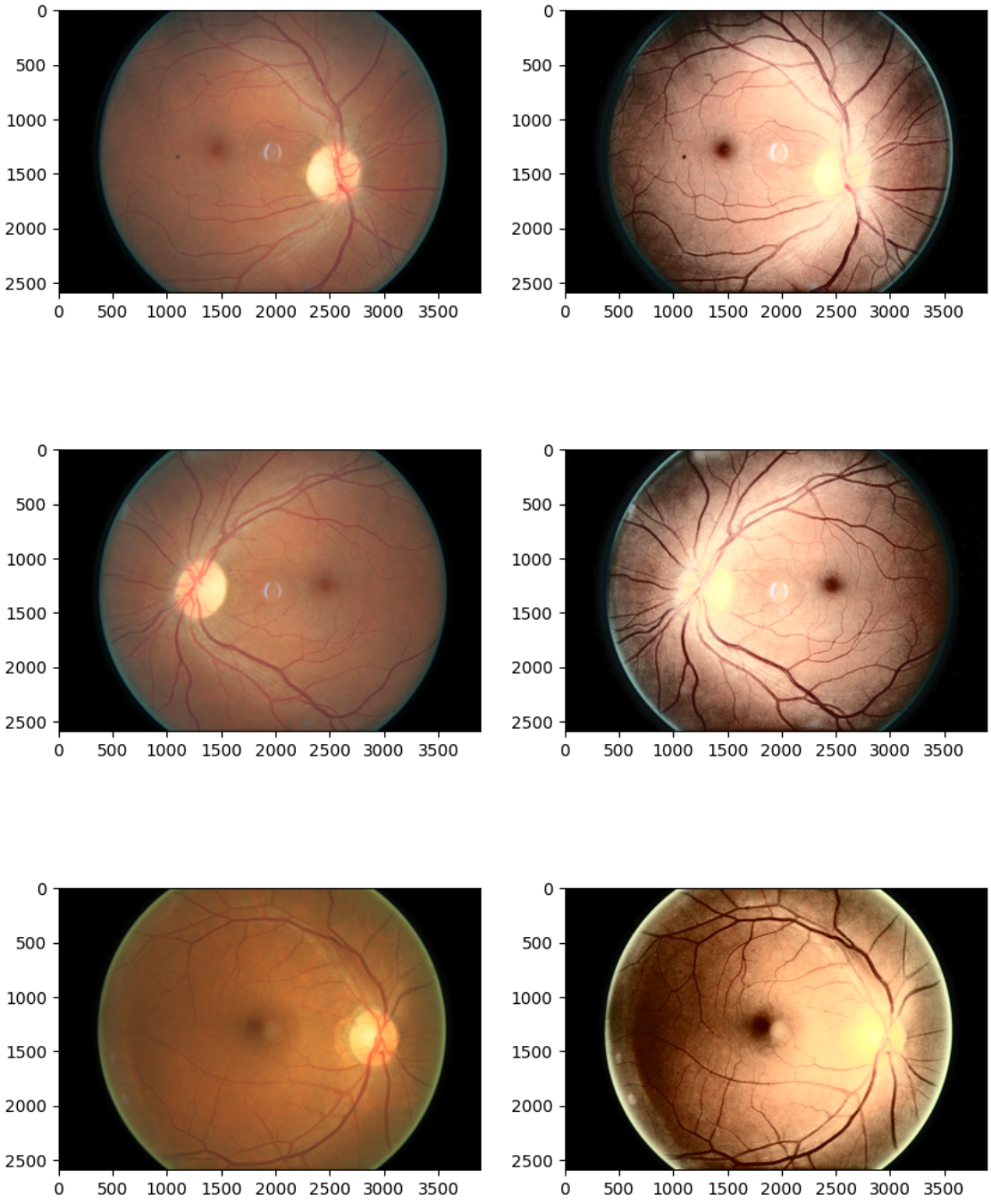
2 Method 2

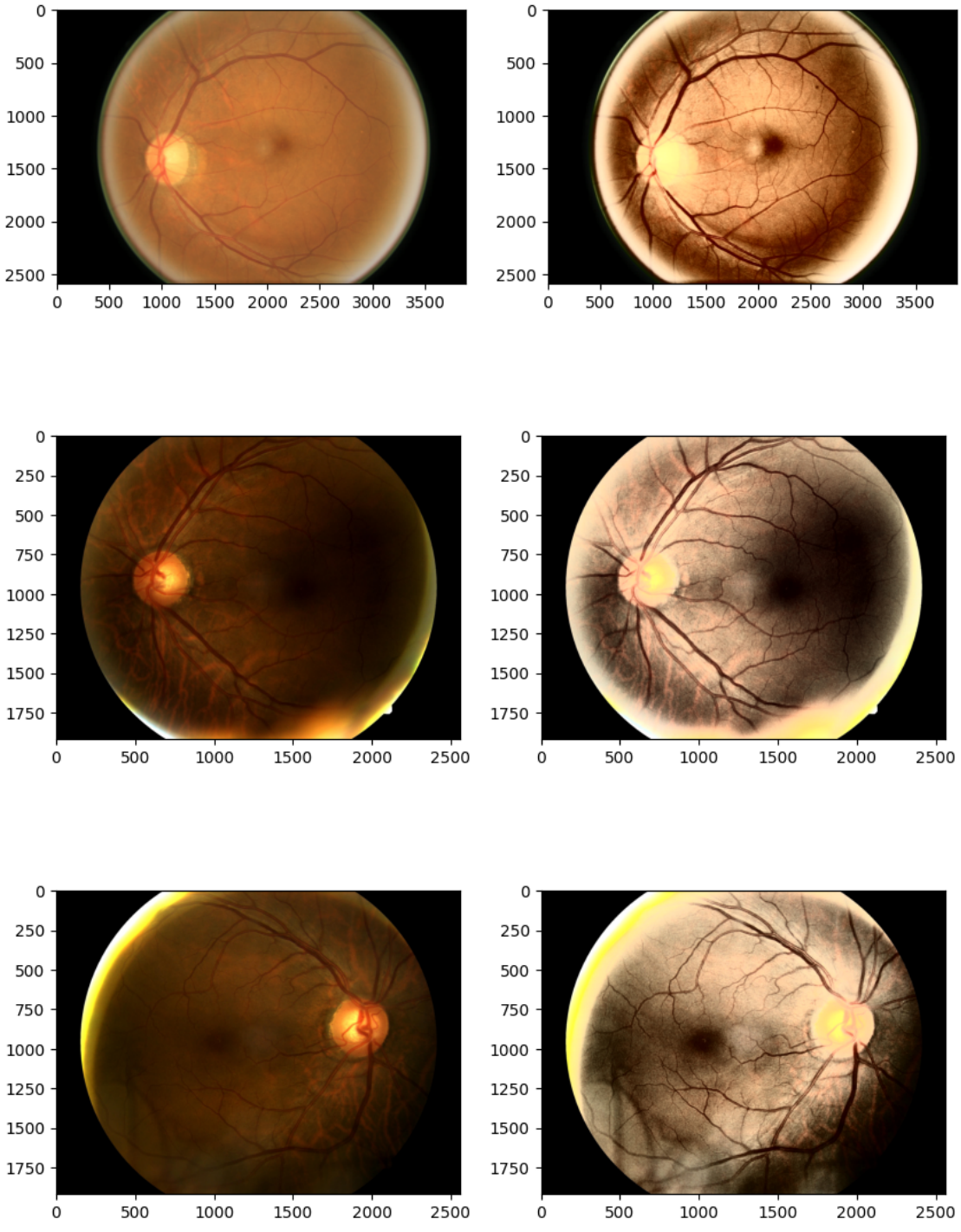
[15]:
```
def enhanceImageMethod2(image, scale):
    scaled = image.copy()
    binary_mask = np.zeros(scaled.shape)
    cv2.circle(binary_mask, (scaled.shape[1] // 2, scaled.shape[0] // 2),␣
 ↪int(scale * 0.9), (1, 1, 1), -1, 8, 0)

    img_yuv = cv2.cvtColor(scaled, cv2.COLOR_BGR2YCrCb)
    img_yuv[:, :, 0] = cv2.equalizeHist(scaled[:, :, 1])
    return cv2.cvtColor(img_yuv, cv2.COLOR_YCR_CB2RGB)

counter = 0
for path in image_paths:
    img = cv2.imread(path)

    original = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    enhanced = enhanceImageMethod2(img, scale)

    f1_score, precision, recall, psnr = evaluateEnhancement(original, enhanced,␣
 ↪ground_truth_img=original, threshold=128)
    plotImages([original, enhanced])
    print(f"Image {counter}\t| F1-score: {f1_score:.4f}\t| Precision:␣
 ↪{precision:.4f}\t| Recall: {recall:.4f}\t| PSNR: {psnr:.4f}")
    counter += 1
```

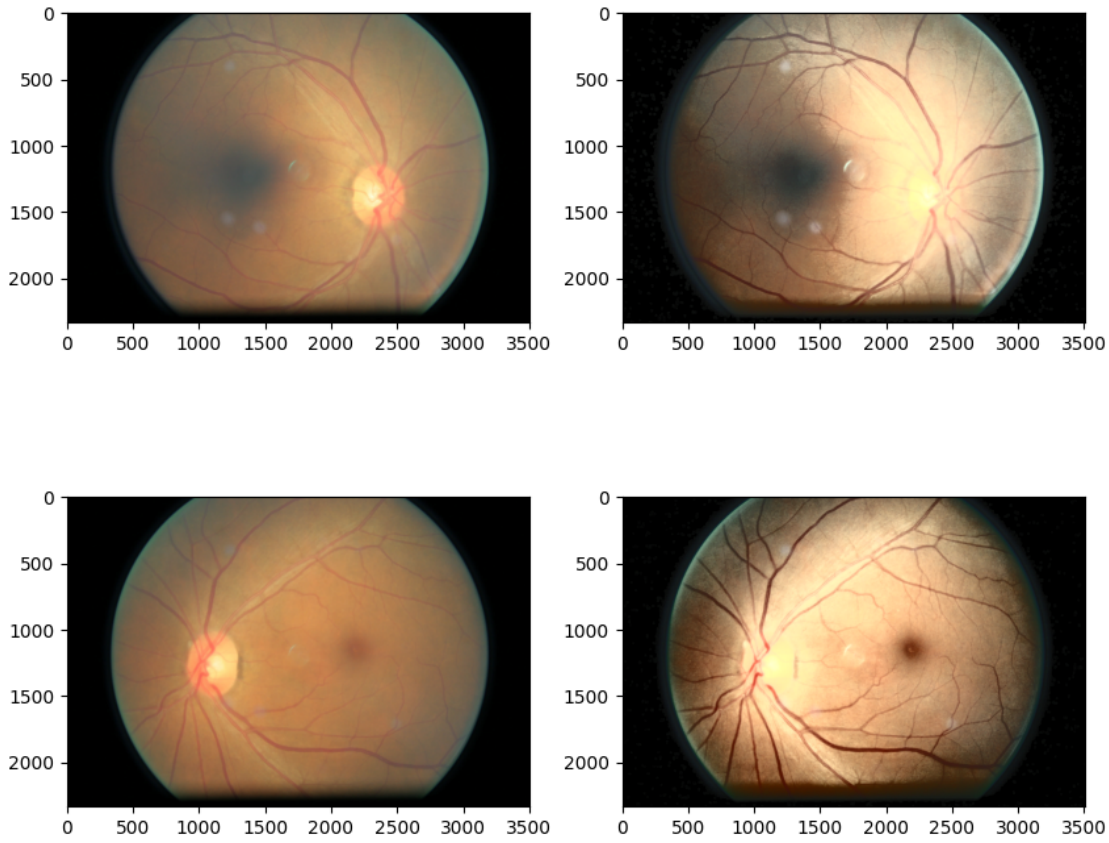
```
Image 0 | F1-score: 0.9653      | Precision: 0.9853     | Recall: 0.9462
| PSNR: 29.3163
Image 1 | F1-score: 0.8084      | Precision: 0.6785     | Recall: 0.9999
| PSNR: 28.7590
```

```
Image 2 | F1-score: 0.5775      | Precision: 0.4059     | Recall: 1.0000
| PSNR: 29.0326
Image 3 | F1-score: 0.6344      | Precision: 0.4645     | Recall: 1.0000
| PSNR: 29.0688
Image 4 | F1-score: 0.3998      | Precision: 0.2498     | Recall: 1.0000
| PSNR: 29.1122
Image 5 | F1-score: 0.7781      | Precision: 0.6374     | Recall: 0.9985
| PSNR: 29.1125
Image 6 | F1-score: 0.1299      | Precision: 0.0695     | Recall: 1.0000
| PSNR: 29.4083
Image 7 | F1-score: 0.1774      | Precision: 0.0974     | Recall: 1.0000
| PSNR: 29.6307
Image 8 | F1-score: 0.8074      | Precision: 0.6770     | Recall: 0.9998
| PSNR: 29.1622
Image 9 | F1-score: 0.8694      | Precision: 0.7695     | Recall: 0.9991
| PSNR: 28.9221
```




0
500
1000
1500
2000
2500
3000
3500





`enhanceImageMethod2` function:

1. `def enhanceImageMethod2(image):` - This line defines the function and takes an image as input.
2. `img_yuv = cv2.cvtColor(image, cv2.COLOR_BGR2YCrCb)` - This line converts the color space of the input image from BGR to YCrCb using OpenCV's `cvtColor` function. YCrCb is a color space that separates the image into three channels: Y (luma), Cr (red difference), and Cb (blue difference).
3. `img_yuv[:, :, 0] = cv2.equalizeHist(image[:, :, 1])` - This line applies histogram equalization to the red channel of the YCrCb image. Histogram equalization is a technique that enhances the contrast of an image by spreading out the intensity values across the entire range of the image. In this case, the `equalizeHist` function is applied to the second channel (`[:, :, 1]`) of the YCrCb image, which corresponds to the red channel.
4. `return cv2.cvtColor(img_yuv, cv2.COLOR_YCR_CB2RGB)` - This line converts the color space of the enhanced YCrCb image back to RGB using OpenCV's `cvtColor` function. The resulting image has improved contrast and is suitable for further image processing or analysis.

Overall, the `enhanceImageMethod2` function enhances an input image using a method that involves converting the color space of the image to YCrCb, applying histogram equalization to the red channel, and converting the color space back to RGB. The resulting image has improved contrast

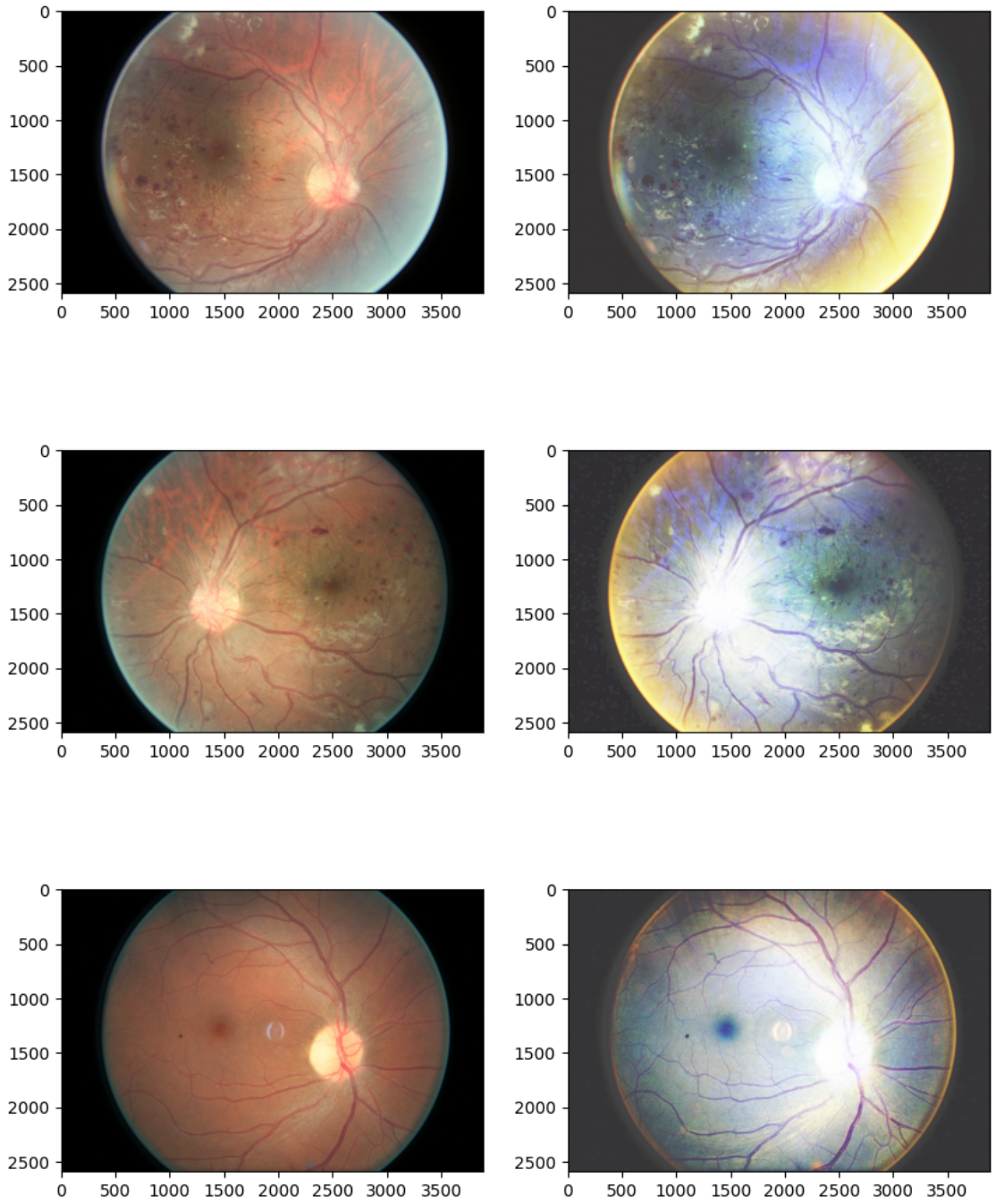
and is suitable for further image processing or analysis.

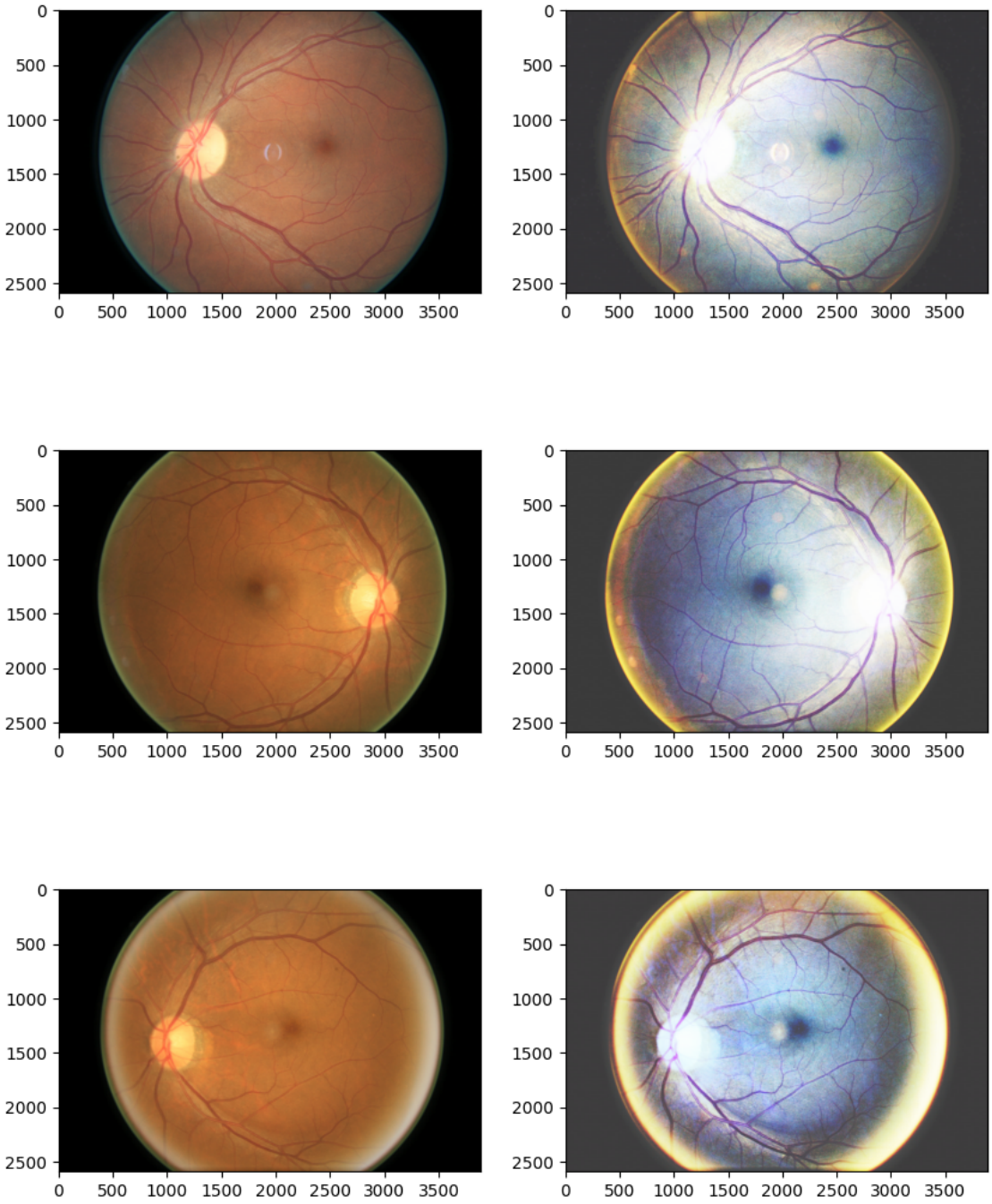
3 Method 3

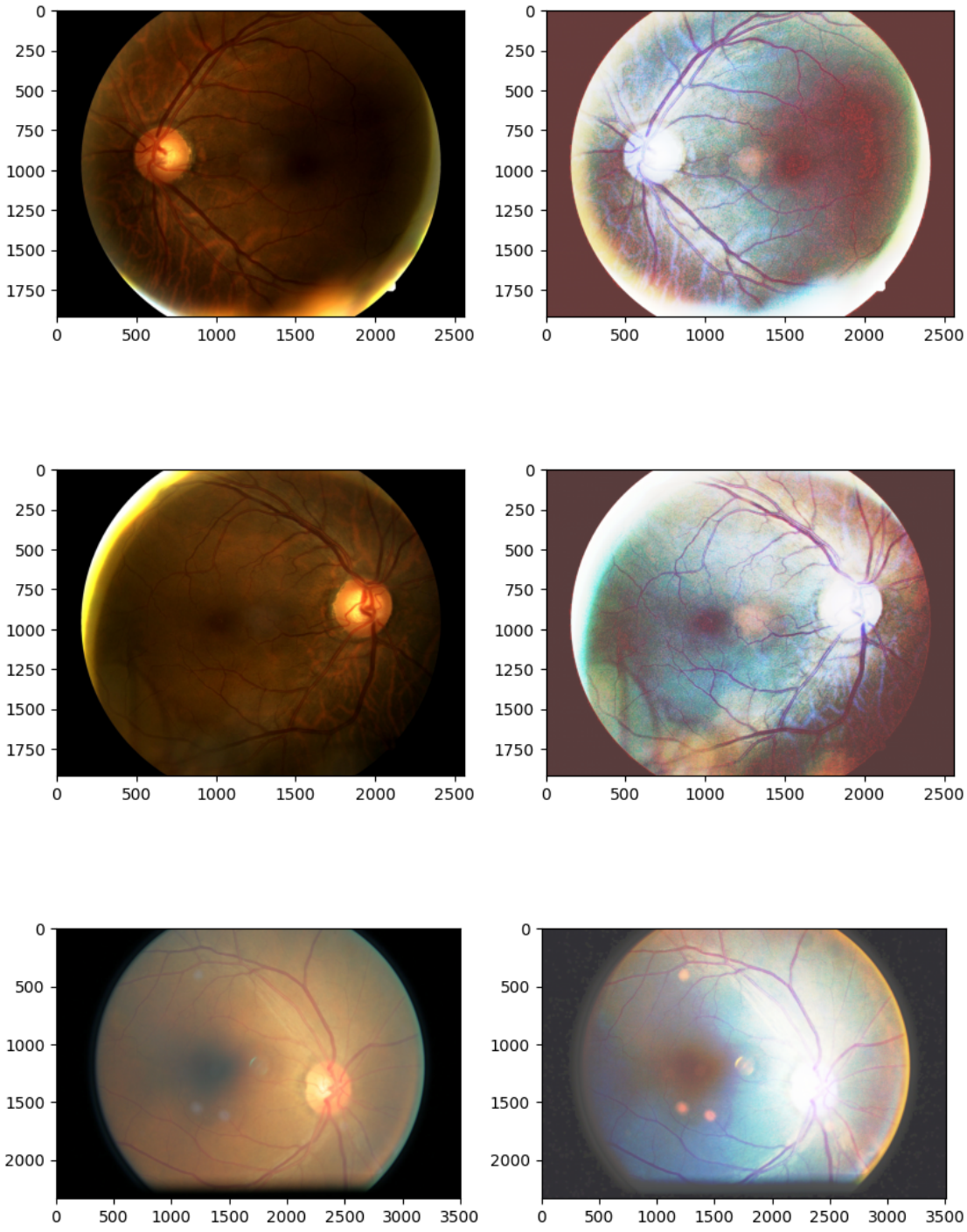
[16]:
```
def enhanceImageMethod3(img):
    scaled = img.copy()
    output = np.zeros((scaled.shape[0], scaled.shape[1], 3))
    cv2.circle(output, (scaled.shape[1] // 2, scaled.shape[0] // 2), int(scale␣
 ↪* 0.9), (1, 1, 1), -1, 8, 0)

    for channel in range(scaled.shape[2]):
        output[:, :, channel] = ex.equalize_hist(scaled[:, :, channel]) * 255

    output[output >= 255] = 255
    output[output <= 0] = 0
    return output.astype(np.uint8)

counter = 0
for path in image_paths:
    img = cv2.imread(path)

    original = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    enhanced = enhanceImageMethod3(img)

    f1_score, precision, recall, psnr = evaluateEnhancement(original, enhanced,␣
 ↪ground_truth_img=original, threshold=128)
    plotImages([original, enhanced])
    print(f"Image {counter}\t| F1-score: {f1_score:.4f}\t| Precision:␣
 ↪{precision:.4f}\t| Recall: {recall:.4f}\t| PSNR: {psnr:.4f}")
    counter += 1
```

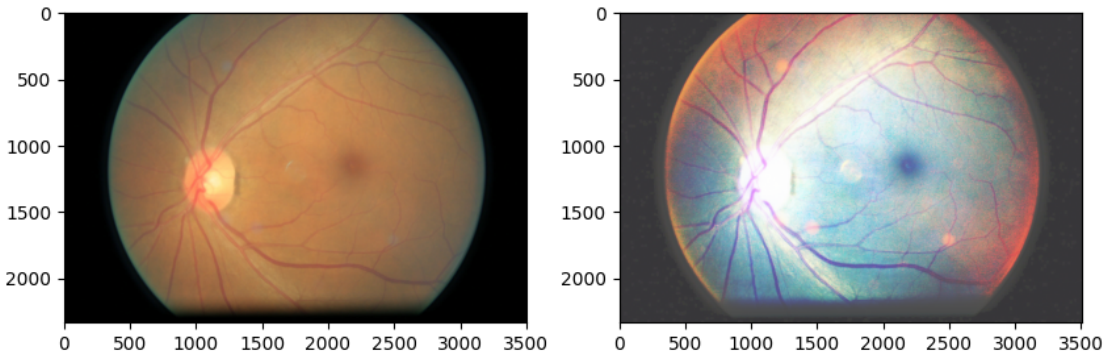
```
Image 0 | F1-score: 0.9055      | Precision: 0.8274     | Recall: 1.0000
| PSNR: 27.7379
Image 1 | F1-score: 0.7026      | Precision: 0.5416     | Recall: 1.0000
| PSNR: 27.9988
Image 2 | F1-score: 0.4852      | Precision: 0.3203     | Recall: 1.0000
| PSNR: 27.5256
Image 3 | F1-score: 0.5317      | Precision: 0.3621     | Recall: 1.0000
| PSNR: 27.5458
Image 4 | F1-score: 0.3223      | Precision: 0.1921     | Recall: 1.0000
| PSNR: 27.3872
Image 5 | F1-score: 0.6395      | Precision: 0.4700     | Recall: 1.0000
| PSNR: 28.6034
Image 6 | F1-score: 0.0967      | Precision: 0.0508     | Recall: 1.0000
| PSNR: 28.0249
Image 7 | F1-score: 0.1360      | Precision: 0.0730     | Recall: 1.0000
| PSNR: 27.2829
```

```
Image 8 | F1-score: 0.7062      | Precision: 0.5458     | Recall: 1.0000
| PSNR: 27.6338
Image 9 | F1-score: 0.7303      | Precision: 0.5751     | Recall: 1.0000
| PSNR: 28.0624
```




0
500
1000
1500
2000
2500
3000
3500





`enhanceImageMethod3` function:

1. The scaled image is processed channel-wise using a for loop that iterates over the number of channels in the image. For each channel, the `ex.equalize_hist()` function is applied to perform histogram equalization on the channel. This helps to improve the contrast and brightness of the image.
    - The `ex.equalize_hist()` function takes an input image as an argument and returns the histogram-equalized image. It works by first converting the input image to grayscale, computing the histogram of the grayscale image, and then computing the CDF of the histogram. The CDF is then used to map the pixel values in the grayscale image to a new range of values that are more evenly distributed. Finally, the histogram-equalized grayscale image is converted back to the original color space of the input image.
2. The enhanced channels are merged back into a single image with the same shape as the scaled image, and the pixel values are scaled to the range [0, 255] using the expression `output[:, :, channel] = ex.equalize_hist(scaled[:, :, channel]) * 255`.
3. The pixel values in the resulting image are clipped to the range [0, 255] using the `np.clip()` function with the arguments `output[output >= 255] = 255` and `output[output <= 0] = 0`.
4. The resulting image is returned as a 3D numpy array with data type `np.uint8`.
    - `np.uint8` is a NumPy data type that represents an 8-bit unsigned integer. An unsigned integer is a non-negative integer that does not have a sign bit, which means that it can only represent positive values. An 8-bit unsigned integer has a range of values from 0 to 255, which makes it useful for representing pixel values in digital images.

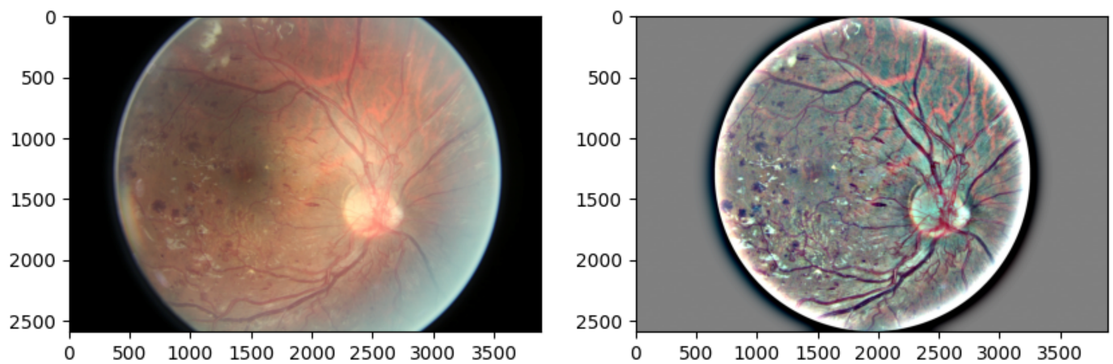
The resulting image has improved contrast and brightness, and any unwanted artifacts in the image have been removed. This can help in better visualization and analysis of the image.

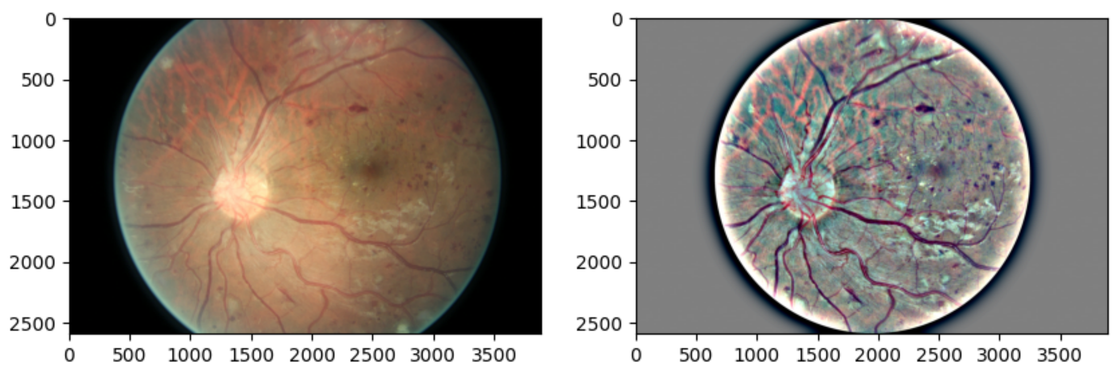
4 Method 4

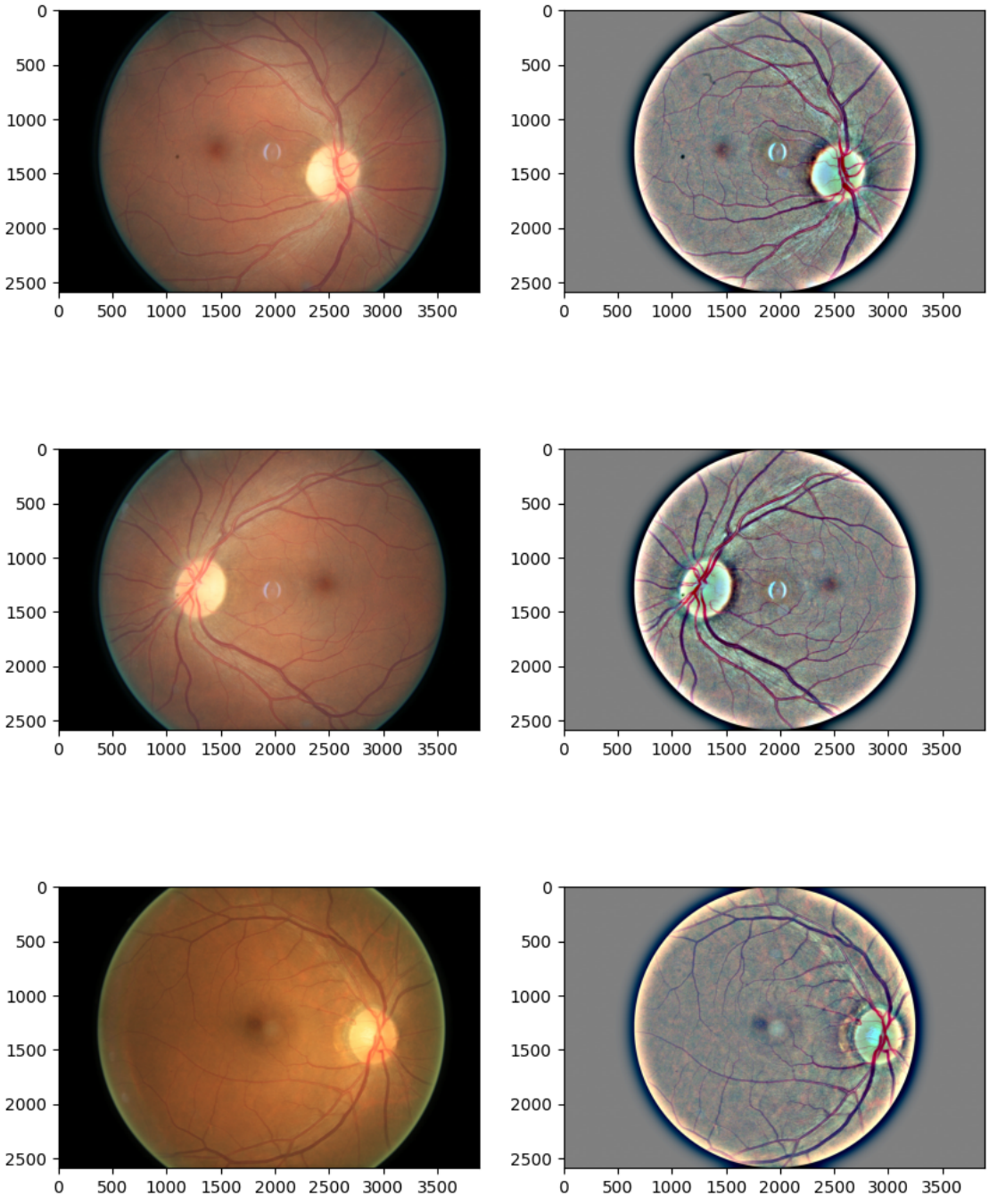
[17]:
```
def circularCrop(image, sigmaX=70):
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    height, width, depth = image.shape

    xCenter = int(width / 2)
    yCenter = int(height / 2)
    radius = np.amin((xCenter, yCenter))

    circleMask = np.zeros((height, width), np.uint8)
    cv2.circle(circleMask, (xCenter, yCenter), int(radius), 1, thickness=-1)

    image = cv2.bitwise_and(image, image, mask=circleMask)
    image = cv2.addWeighted(image, 4, cv2.GaussianBlur(image, (0, 0), sigmaX),␣
 ↪-4, 128)

    return image

def enhanceImageMethod4(image):
    image = circularCrop(image)
    return image

counter = 0
for image_path in image_paths:
    image = cv2.imread(image_path)

    original = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    enhanced = enhanceImageMethod4(image)

    f1_score, precision, recall, psnr = evaluateEnhancement(original, enhanced,␣
 ↪ground_truth_img=original, threshold=128)
    plotImages([original, enhanced])
    print(f"Image {counter}\t| F1-score: {f1_score:.4f}\t| Precision:␣
 ↪{precision:.4f}\t| Recall: {recall:.4f}\t| PSNR: {psnr:.4f}")
    counter += 1
```

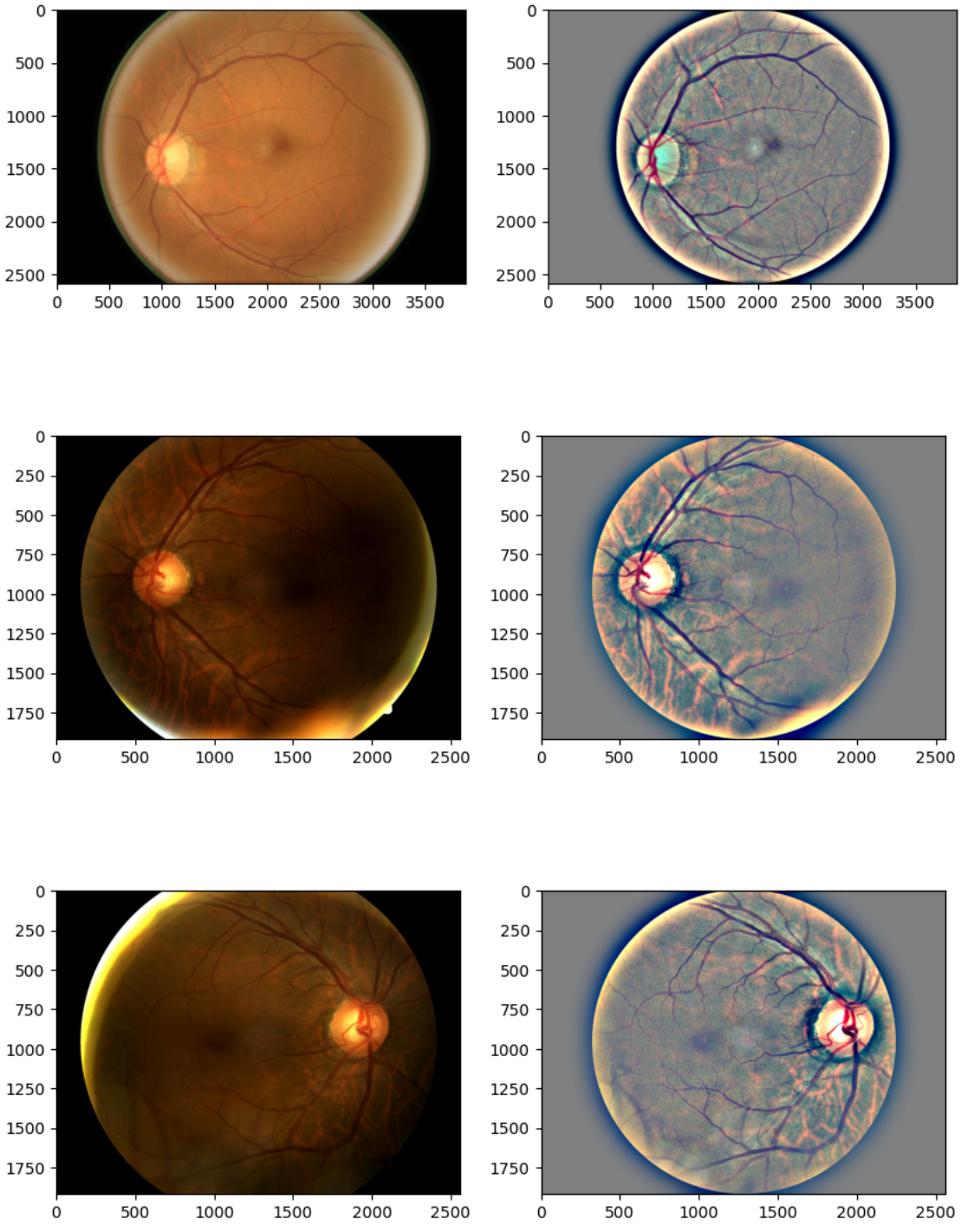
```
Image 0 | F1-score: 0.5405      | Precision: 0.6403     | Recall: 0.4676
| PSNR: 29.0917
Image 1 | F1-score: 0.5449      | Precision: 0.5017     | Recall: 0.5964
| PSNR: 29.1618
Image 2 | F1-score: 0.4122      | Precision: 0.3059     | Recall: 0.6319
| PSNR: 29.1041
Image 3 | F1-score: 0.4580      | Precision: 0.3531     | Recall: 0.6515
| PSNR: 29.0991
Image 4 | F1-score: 0.3138      | Precision: 0.2039     | Recall: 0.6803
```

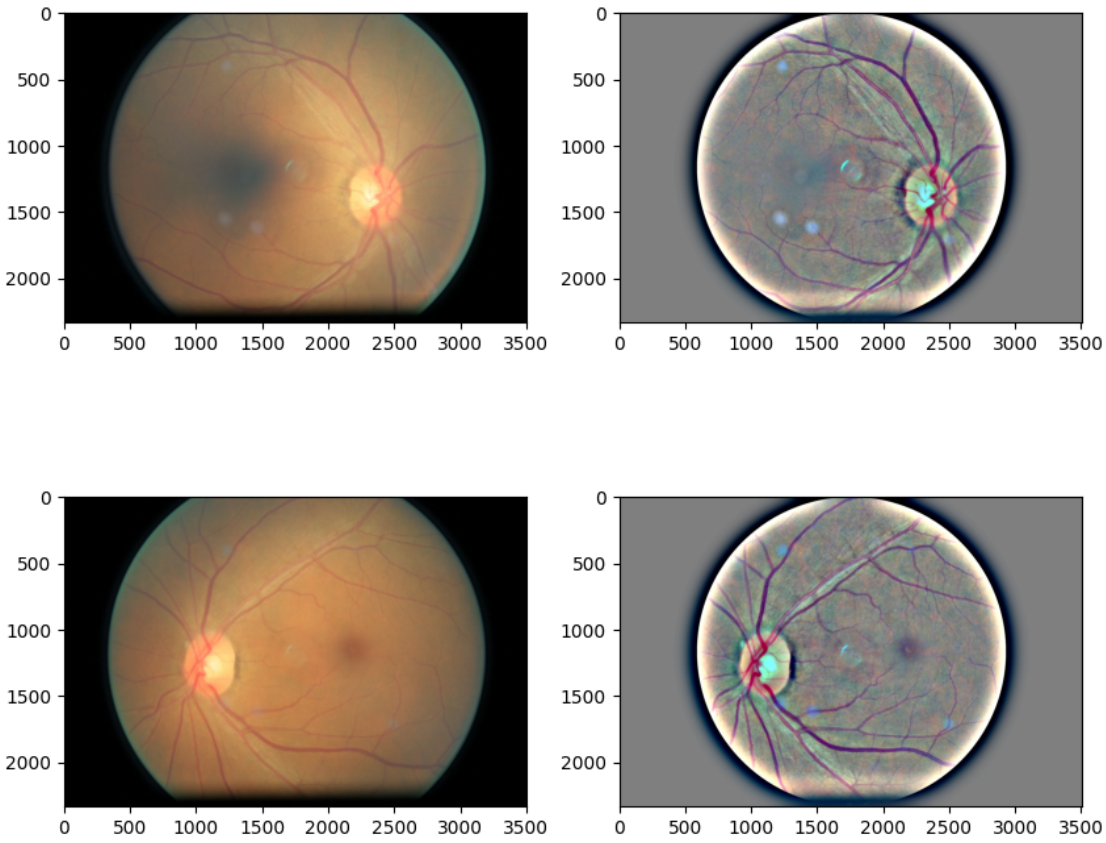
```
| PSNR: 29.0253
Image 5 | F1-score: 0.3724      | Precision: 0.3250      | Recall: 0.4359
| PSNR: 29.0091
Image 6 | F1-score: 0.0414      | Precision: 0.0224      | Recall: 0.2715
| PSNR: 29.8308
Image 7 | F1-score: 0.0486      | Precision: 0.0269      | Recall: 0.2469
| PSNR: 29.6495
Image 8 | F1-score: 0.5865      | Precision: 0.5340      | Recall: 0.6504
| PSNR: 29.1818
Image 9 | F1-score: 0.5941      | Precision: 0.5561      | Recall: 0.6376
| PSNR: 29.1331
```






0
500
1000
1500
2000
2500
3000
3500





Sure, here's a detailed explanation of the two functions:

- `cropImageFromGray(image, tolerance=7)`: This function takes an image as input and returns a cropped version of the image based on the gray intensity of the pixels. If the image is grayscale (i.e., has only one channel), it simply applies a threshold to the image and returns the cropped version of the image. If the image is RGB (i.e., has three channels), it first converts the image to grayscale and then applies the threshold. It then checks if the resulting image is too dark (i.e., has no pixels above the threshold) and returns the original image if it is. Otherwise, it crops the image based on the threshold and returns the cropped version of the image.
    1. `def cropImageFromGray(image, tolerance=7)`: This line defines the function and specifies two parameters: `image`, which is the input image, and `tolerance`, which is the threshold for the gray intensity.
    2. `if image.ndim == 2:`: This line checks if the image is grayscale (i.e., has only one channel).
    3. `mask = image > tolerance`: This line creates a binary mask where the pixel values above the threshold are set to 1 and the pixel values below the threshold are set to 0.
    4. `return image[np.ix_(mask.any(1), mask.any(0))]`: This line returns the cropped version of the image by selecting the rows and columns where the mask has any non-zero values.
    5. `elif image.ndim == 3:`: This line checks if the image is RGB (i.e., has three channels).

6. `grayImage = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)`: This line converts the RGB image to grayscale.
7. `mask = grayImage > tolerance`: This line creates a binary mask based on the gray intensity of the pixels.
8. `checkShape = image[:, :, 0][np.ix_(mask.any(1), mask.any(0))].shape[0]`: This line checks if the resulting image is too dark (i.e., has no pixels above the threshold) by selecting the rows and columns where the mask has any non-zero values and checking the shape of the resulting image.
9. `if checkShape == 0:`: This line checks if the resulting image is too dark.
10. `imageChannels = [image[:, :, i][np.ix_(mask.any(1), mask.any(0))] for i in range(3)]`: This line selects the rows and columns where the mask has any non-zero values for each channel of the RGB image.
11. `image = np.stack(imageChannels, axis=-1)`: This line stacks the resulting image channels to create the cropped version of the image.

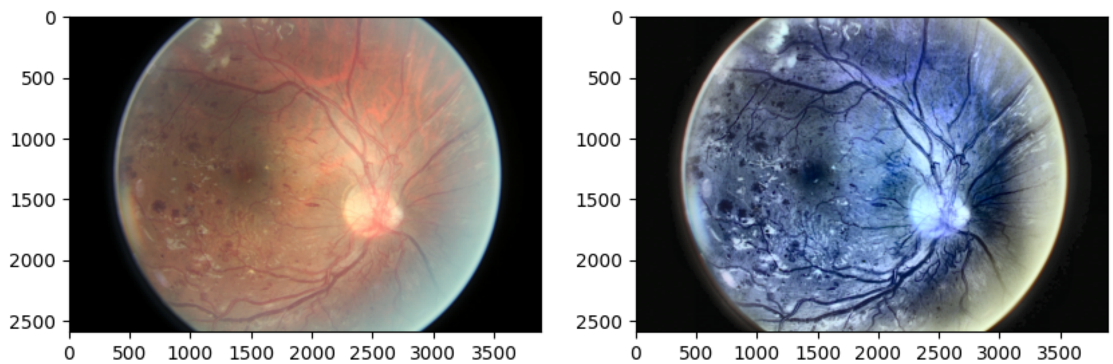
- `circularCrop(image, sigmaX=50)`: This function takes an image as input and returns a circularly cropped and enhanced version of the image. It first calls the `cropImageFromGray` function to crop the image based on gray intensity. It then converts the image to RGB format and calculates the center and radius of the circular mask. It creates a circular mask using the center and radius and applies it to the image using a bitwise AND operation. Finally, it enhances the image using Gaussian blur and returns the enhanced image.
  1. `def circularCrop(image, sigmaX=50)`: This line defines the function and specifies two parameters: `image`, which is the input image, and `sigmaX`, which is the standard deviation of the Gaussian blur.
  2. `image = cropImageFromGray(image)`: This line calls the `cropImageFromGray` function to crop the image based on gray intensity.
  3. `image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)`: This line converts the image from BGR format to RGB format.
  4. `height, width, depth = image.shape`: This line gets the height, width, and depth of the image.
  5. `xCenter = int(width / 2)`: This line calculates the x-coordinate of the center of the circular mask.
  6. `yCenter = int(height / 2)`: This line calculates the y-coordinate of the center of the circular mask.
  7. `radius = np.amin((xCenter, yCenter))`: This line calculates the radius of the circular mask as the minimum of the x-coordinate and y-coordinate of the center.
  8. `circleMask = np.zeros((height, width), np.uint8)`: This line creates a binary mask with the same dimensions as the image.
  9. `cv2.circle(circleMask, (xCenter, yCenter), int(radius), 1, thickness=-1)`: This line creates a circular mask by drawing a filled circle on the binary mask.
  10. `image = cv2.bitwise_and(image, image, mask=circleMask)`: This line applies the circular mask to the image using a bitwise AND operation.
  11. `image = cv2.addWeighted(image, 4, cv2.GaussianBlur(image, (0, 0), sigmaX), -4, 128)`: This line enhances the image using Gaussian blur and returns the enhanced image.

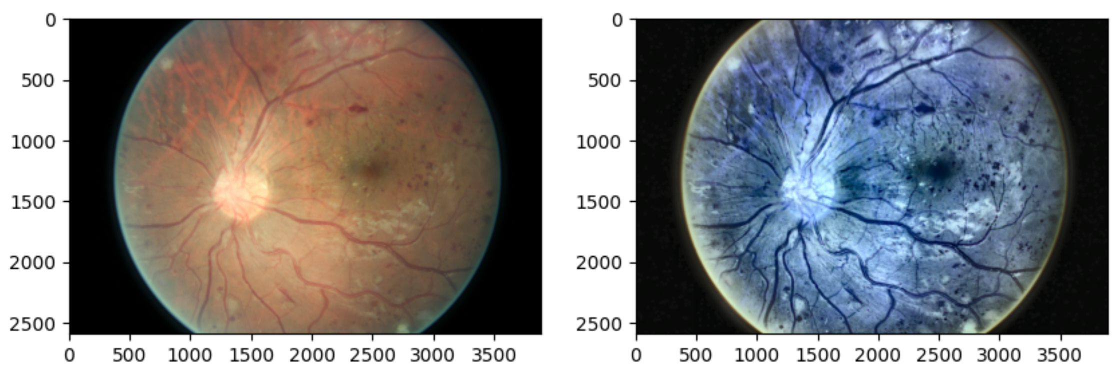
5 Method 5

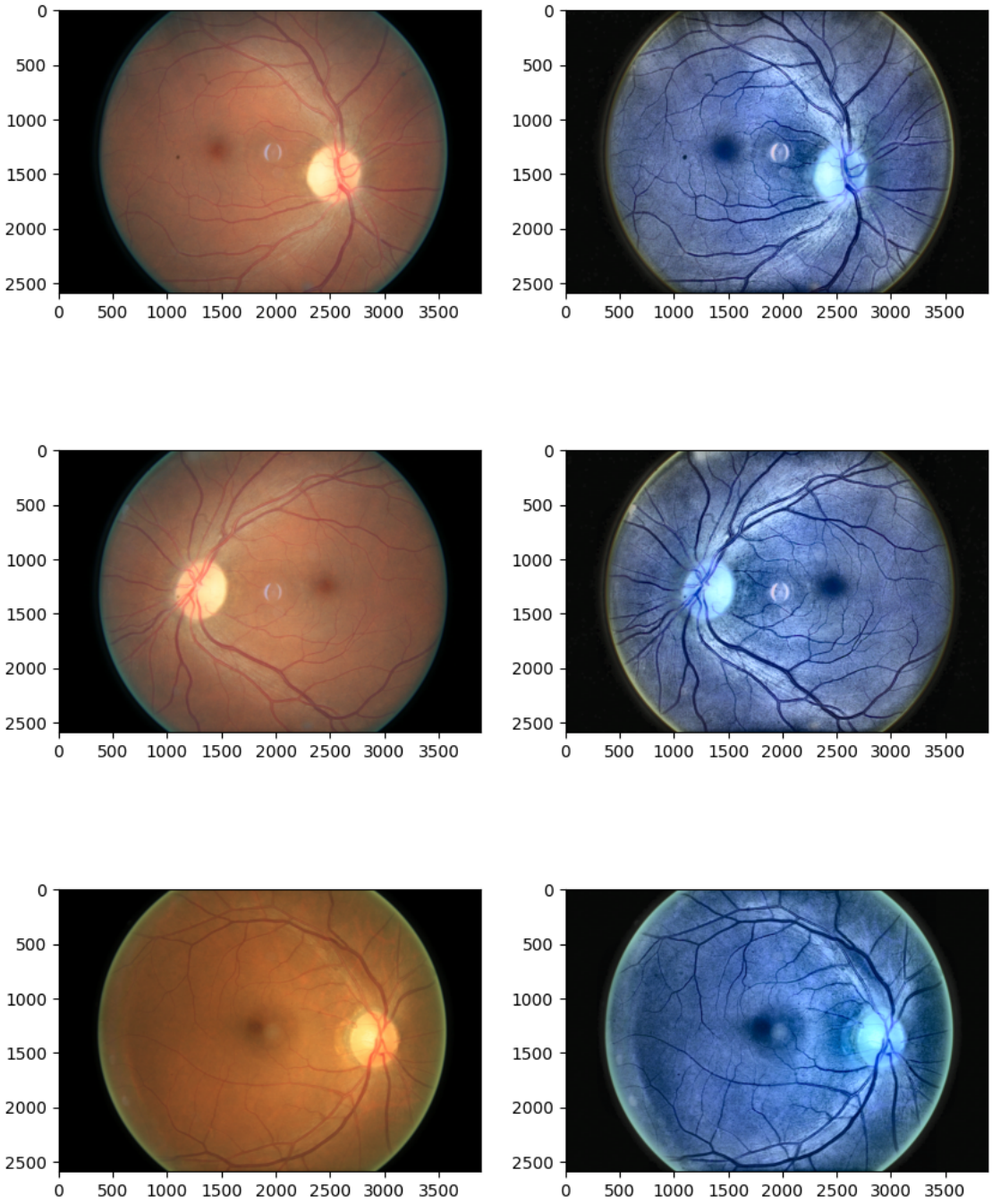
[18]:
```
def enhanceImageMethod5(img):

    gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

    blurred_gray = cv2.GaussianBlur(gray, (5, 5), 0)
    binary = cv2.threshold(blurred_gray, 10, 255, cv2.THRESH_BINARY)[1]

    contours = cv2.findContours(binary, cv2.RETR_EXTERNAL, cv2.
 ↪CHAIN_APPROX_SIMPLE)[0][0]
    contours = contours[:, 0, :]

    lab_image = cv2.cvtColor(img, cv2.COLOR_RGB2LAB)
    l_channel, a_channel, b_channel = cv2.split(lab_image)

    clahe = cv2.createCLAHE(clipLimit=5.0, tileGridSize=(8, 8))
    clahe_l_channel = clahe.apply(l_channel)
    clahe_l_channel_merged = cv2.merge((clahe_l_channel, a_channel, b_channel))

    enhanced = cv2.cvtColor(clahe_l_channel_merged, cv2.COLOR_LAB2RGB)
    median_enhanced = cv2.medianBlur(enhanced, 3)
    background = cv2.medianBlur(enhanced, 37)
    mask = cv2.addWeighted(median_enhanced, 1, background, -1, 255)
    final_image = cv2.bitwise_and(mask, median_enhanced)

    return final_image

counter = 0
for image_path in image_paths:
    img = cv2.imread(image_path)

    original = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    enhanced = enhanceImageMethod5(img)

    f1_score, precision, recall, psnr = evaluateEnhancement(original, enhanced,␣
 ↪ground_truth_img=original, threshold=128)
    plotImages([original, enhanced])
    print(f"Image {counter}\t| F1-score: {f1_score:.4f}\t| Precision:␣
 ↪{precision:.4f}\t| Recall: {recall:.4f}\t| PSNR: {psnr:.4f}")
    counter += 1
```

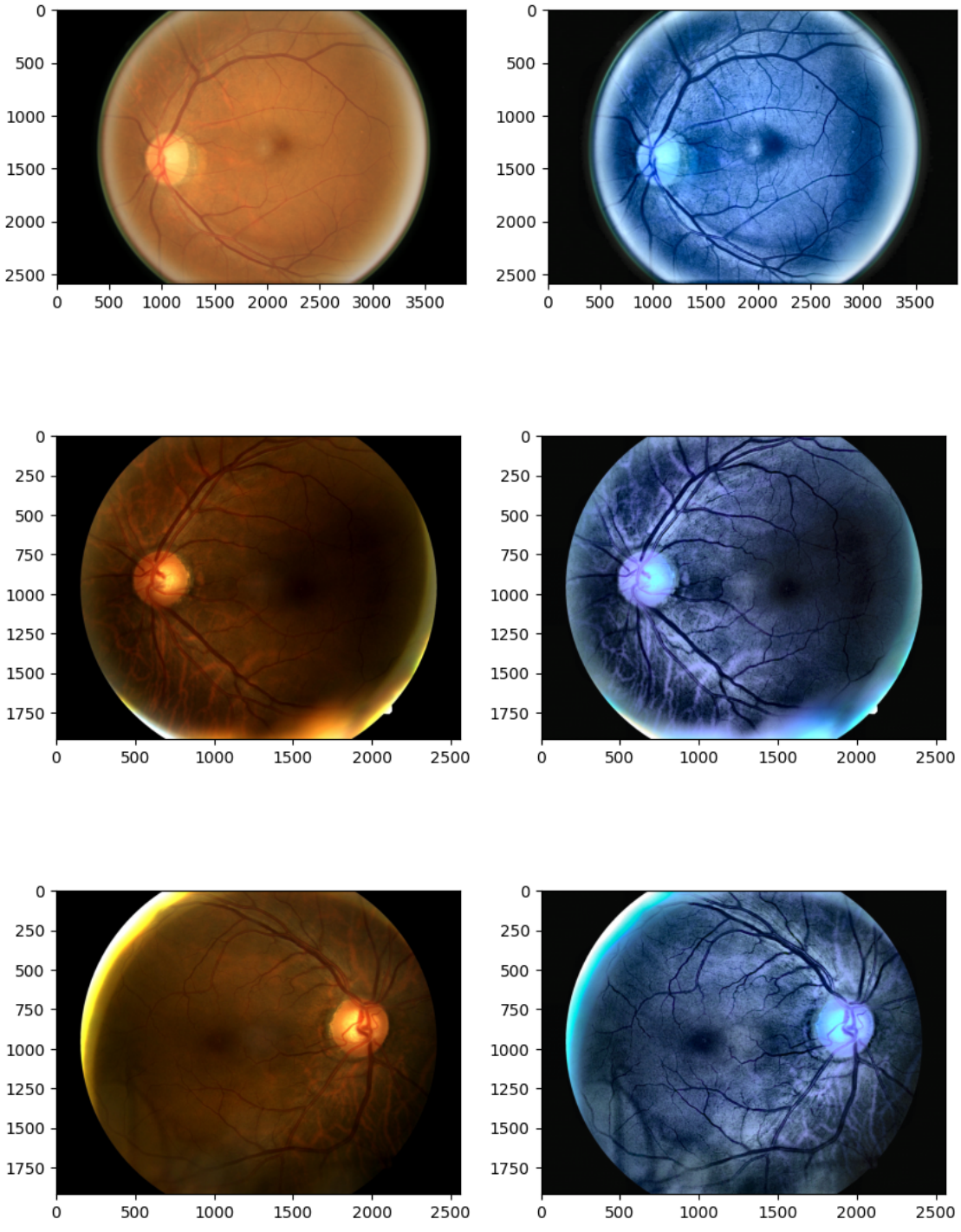
```
Image 0 | F1-score: 0.6654      | Precision: 0.7940     | Recall: 0.5727
| PSNR: 28.2480
Image 1 | F1-score: 0.6448      | Precision: 0.6597     | Recall: 0.6306
| PSNR: 28.0556
Image 2 | F1-score: 0.6316      | Precision: 0.5904     | Recall: 0.6789
```

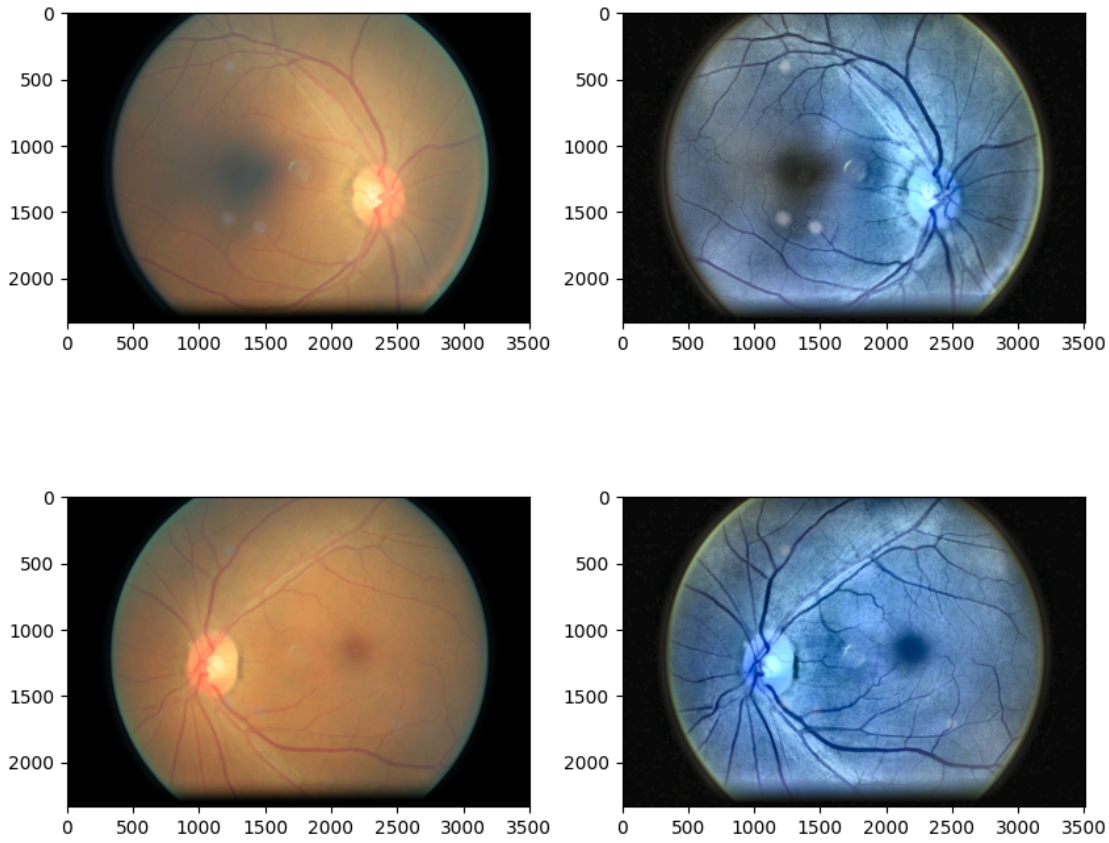
```
| PSNR: 28.1645
Image 3 | F1-score: 0.6288      | Precision: 0.5807      | Recall: 0.6855
| PSNR: 28.1705
Image 4 | F1-score: 0.5317      | Precision: 0.4368      | Recall: 0.6791
| PSNR: 28.3038
Image 5 | F1-score: 0.7564      | Precision: 0.7480      | Recall: 0.7649
| PSNR: 28.3435
Image 6 | F1-score: 0.3796      | Precision: 0.2358      | Recall: 0.9731
| PSNR: 28.5665
Image 7 | F1-score: 0.5295      | Precision: 0.3621      | Recall: 0.9843
| PSNR: 28.4335
Image 8 | F1-score: 0.6133      | Precision: 0.6688      | Recall: 0.5664
| PSNR: 28.2093
Image 9 | F1-score: 0.6660      | Precision: 0.7120      | Recall: 0.6256
| PSNR: 28.2886
```






0
500
1000
1500
2000
2500
3000
3500





`enhanceImageMethod5` function:

1. `cv2.cvtColor()` function is used to convert the input image from RGB color space to grayscale. The `cv2.COLOR_RGB2GRAY` parameter specifies the conversion type.
2. `cv2.GaussianBlur()` function is used to apply a Gaussian blur to the grayscale image. The `(5, 5)` parameter specifies the kernel size of the Gaussian filter, and the `0` parameter specifies the sigma value.
    - The kernel size refers to the size of the Gaussian kernel used to blur the image. The kernel is a square matrix of values, and the kernel size specifies the width and height of the matrix. A larger kernel size results in more blurring, while a smaller kernel size results in less blurring.
    - The sigma value, also known as the standard deviation, determines the spread of the Gaussian distribution used to generate the kernel. A larger sigma value results in a wider distribution and more blurring, while a smaller sigma value results in a narrower distribution and less blurring.
3. `cv2.threshold()` function is used to apply thresholding to the blurred image. The `10` parameter specifies the threshold value, the `255` parameter specifies the maximum value, and the `cv2.THRESH_BINARY` parameter specifies the thresholding type.

- `cv2.THRESH_BINARY` is used as the thresholding type in the `cv2.threshold()` function. This means that all pixels in the blurred image with intensity values below the threshold value of 10 are set to 0, while all pixels with intensity values above 10 are set to 255, creating a binary image.

4. `cv2.findContours()` function is used to find the contours in the binary image. The `cv2.RETR_EXTERNAL` parameter specifies the retrieval mode, and the `cv2.CHAIN_APPROX_SIMPLE` parameter specifies the contour approximation method.
    - `cv2.RETR_EXTERNAL` is a retrieval mode that retrieves only the external contours of an object in the image. This means that only the outermost contours of an object are returned, and any contours inside the object are ignored. This is useful when we only want to detect the boundaries of an object in the image.
    - `cv2.CHAIN_APPROX_SIMPLE` is a contour approximation method that compresses the contour by removing any redundant points. This means that only the endpoints of the contour are retained, which reduces the memory required to store the contour. This is useful when we want to simplify the contour and reduce the number of points required to represent it.
5. `cv2.cvtColor()` function is used to convert the input image from RGB color space to LAB color space. The `cv2.COLOR_RGB2LAB` parameter specifies the conversion type.
    - LAB color space is a color model used in image processing that separates color information into three channels: L (lightness), A (green-red), and B (blue-yellow). The L channel represents the brightness of the image, while the A and B channels represent the color information.
6. `cv2.split()` function is used to split the LAB image into its three channels (L, A, and B).
7. `cv2.createCLAHE()` function is used to apply Contrast Limited Adaptive Histogram Equalization (CLAHE) to the L channel. The `clipLimit` parameter specifies the clip limit, and the `tileGridSize` parameter specifies the size of the tile grid.
    - The `clipLimit` parameter in `cv2.createCLAHE()` specifies the contrast limit for the CLAHE algorithm. This parameter limits the amount of contrast enhancement that can be applied to the image. A higher value of `clipLimit` results in more contrast enhancement, while a lower value results in less contrast enhancement. In the code provided, `clipLimit` is set to 5.0, which means that the contrast enhancement is limited to a maximum of 5.0.
    - The `tileGridSize` parameter in `cv2.createCLAHE()` specifies the size of the grid used to divide the image into smaller regions for local contrast enhancement. The image is divided into a grid of tiles, and the contrast enhancement is applied to each tile individually. A larger `tileGridSize` results in larger tiles and more global contrast enhancement, while a smaller `tileGridSize` results in smaller tiles and more local contrast enhancement. In the code provided, `tileGridSize` is set to (8, 8), which means that the image is divided into a grid of 8x8 tiles for local contrast enhancement.
8. `cv2.merge()` function is used to merge the enhanced L channel with the A and B channels.
9. `cv2.cvtColor()` function is used to convert the enhanced LAB image back to RGB color space. The `cv2.COLOR_LAB2RGB` parameter specifies the conversion type.

10. `cv2.medianBlur()` function is used to apply median blur to the enhanced image. The `3` parameter specifies the kernel size.

    - `cv2.medianBlur()` is a function in OpenCV that is used to apply median filtering to an image. Median filtering is a type of non-linear filtering that is used to remove noise from an image. In the code provided, a kernel size of 3 is used, which means that a 3x3 kernel is used for median filtering. This removes any remaining noise in the image after the previous steps of the `enhanceImageMethod5` function, resulting in a smoother and cleaner image.

11. `cv2.medianBlur()` function is used to apply median blur to a background version of the enhanced image. The `37` parameter specifies the kernel size.

12. `cv2.addWeighted()` function is used to create a mask to subtract the background from the enhanced image. The `1` parameter specifies the weight of the enhanced image, the `-1` parameter specifies the weight of the background image, and the `255` parameter specifies the scalar value added to each pixel in the mask.

    - `cv2.addWeighted()` is a function in OpenCV that is used to blend two images together. In the code you provided, `cv2.addWeighted(median_enhanced, 1, background, -1, 255)` is used to create a mask that will be used to subtract the background from the enhanced image. The `1` parameter in `cv2.addWeighted()` specifies the weight of the `median_enhanced` image, and the `-1` parameter specifies the weight of the `background` image. The `255` parameter specifies a scalar value that is added to each pixel in the mask. The resulting mask is a grayscale image where the foreground (the object in the image) is represented by white pixels, and the background is represented by black pixels. The mask is then applied to the enhanced image using the `cv2.bitwise_and()` function to remove the background from the image, leaving only the foreground.

13. `cv2.bitwise_and()` function is used to apply the mask to the enhanced image. This removes the background from the enhanced image, leaving only the foreground.

    - `cv2.bitwise_and()` is a function in OpenCV that is used to perform a bitwise AND operation between two images. In the code you provided, `cv2.bitwise_and(mask, median_enhanced)` is used to apply a mask to the enhanced image, removing the background and leaving only the foreground object. The `mask` parameter is a binary image that was generated using `cv2.addWeighted()` function in the previous step of the `enhanceImageMethod5` function. The `median_enhanced` parameter is the enhanced image that was generated in the previous steps of the function. The `cv2.bitwise_and()` function performs a bitwise AND operation between the corresponding pixels in the `mask` and `median_enhanced` images. If the pixel in the `mask` image is white (255), the corresponding pixel in the `median_enhanced` image is retained. If the pixel in the `mask` image is black (0), the corresponding pixel in the `median_enhanced` image is set to black. The resulting image is an image where the background has been removed, leaving only the foreground object.