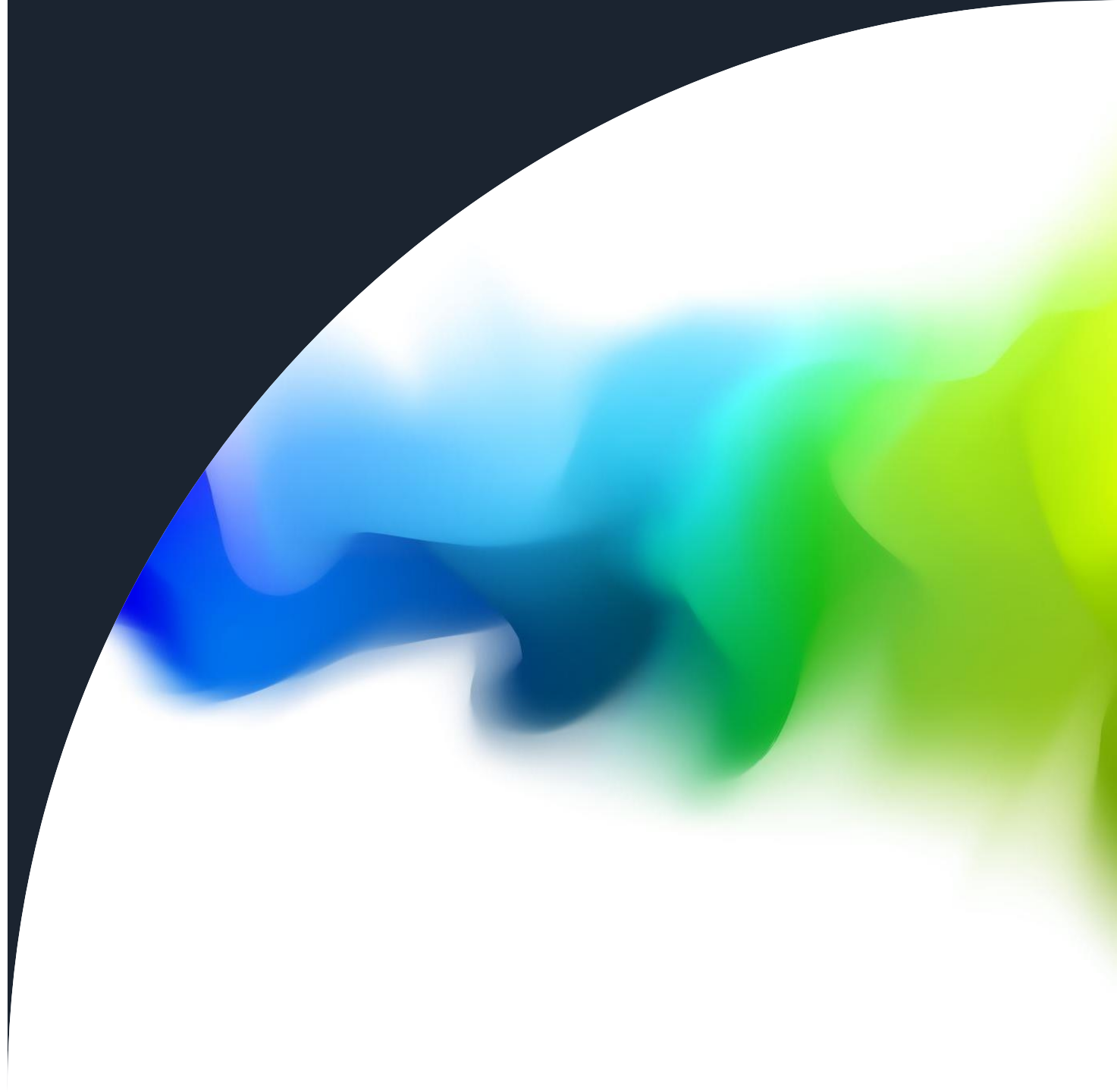



Activity 5: Color Feature Extraction

Ron Michael V. Acda

2019-03839



Objectives

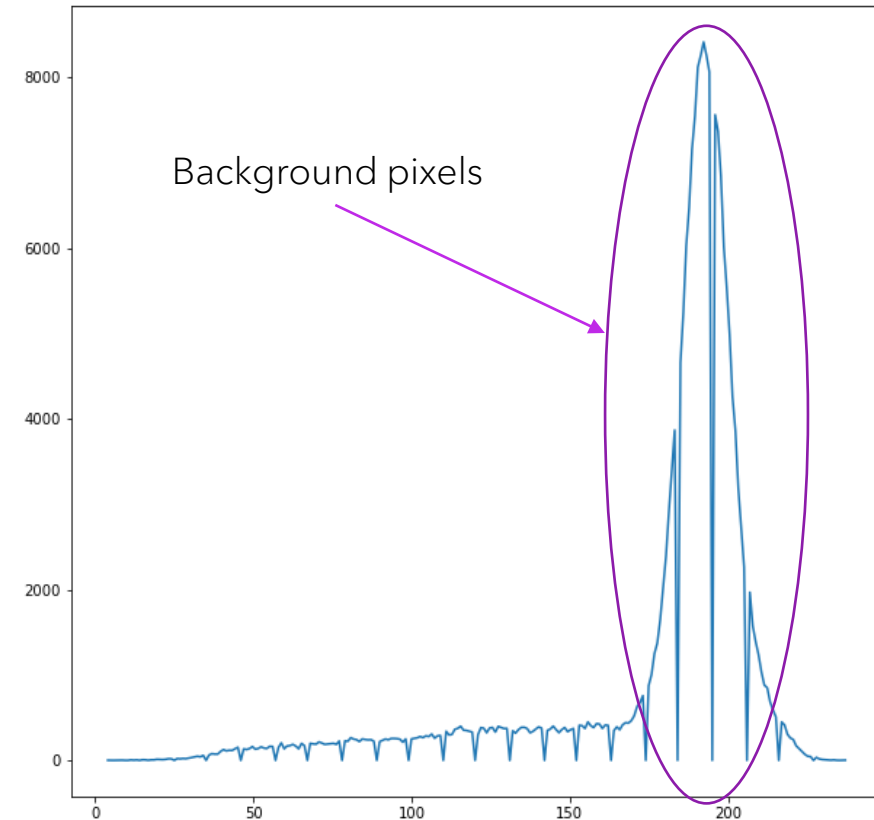
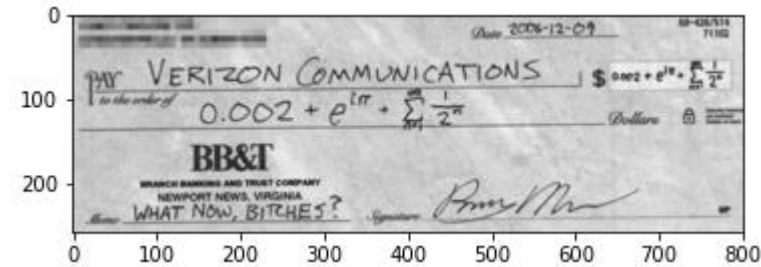
- Extract the background from an image using grayscale thresholding.
 - Transform an image into normalized chromaticity coordinate (NCC) space.
 - Perform Gaussian parametric segmentation of colored images.
 - Perform non-parametric segmentation of colored images.
 - Identify the strengths and weaknesses of parametric and non-parametric segmentation in practical applications.
- 

Objectives

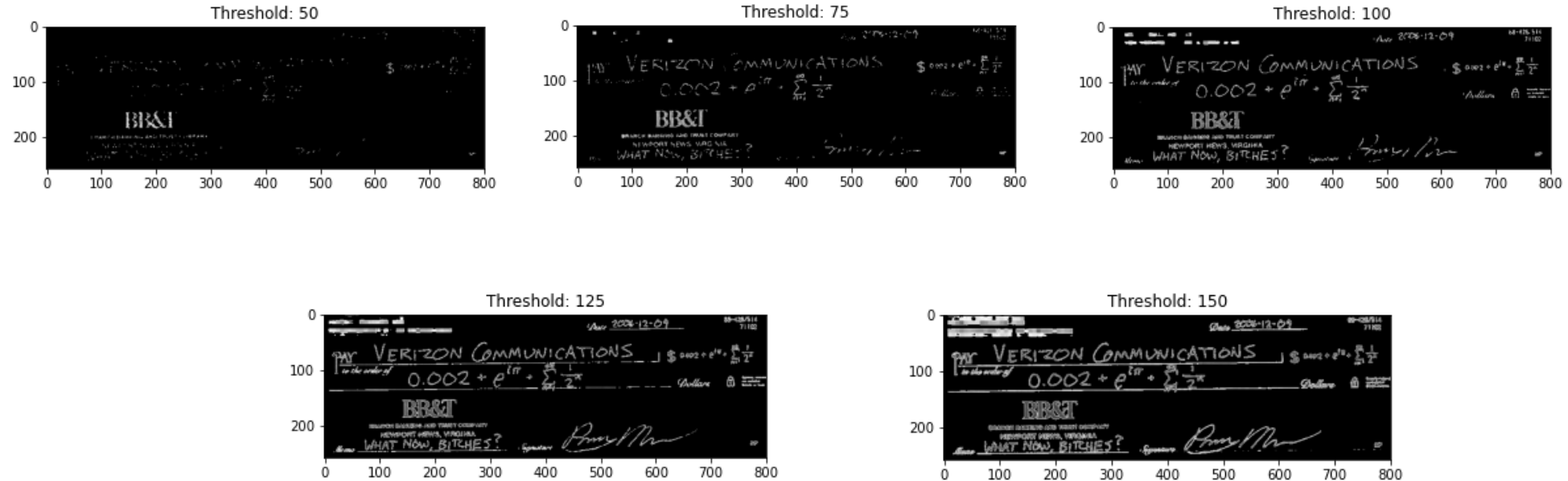
- Extract the background from an image using grayscale thresholding.
- Transform an image into normalized chromaticity coordinate (NCC) space.
- Perform Gaussian parametric segmentation of colored images.
- Perform non-parametric segmentation of colored images.
- Identify the strengths and weaknesses of parametric and non-parametric segmentation in practical applications.

Background Removal

- The background of an image can be removed by setting the grayscale threshold. The image is first loaded in grayscale and then clipped according to a threshold.
- The grayscale histogram is used as a tool to identify the proper threshold. The background is usually located on the histogram peaks, so we clip them off.



Background Removal



- Based from the histogram previously, the background pixels must be somewhere between 175 and 225. Thus, a good threshold to clip the image is approximately within this range or close to it. Too low a threshold and too much is removed, and too high and too few background pixels will be removed.

Background Removal Code

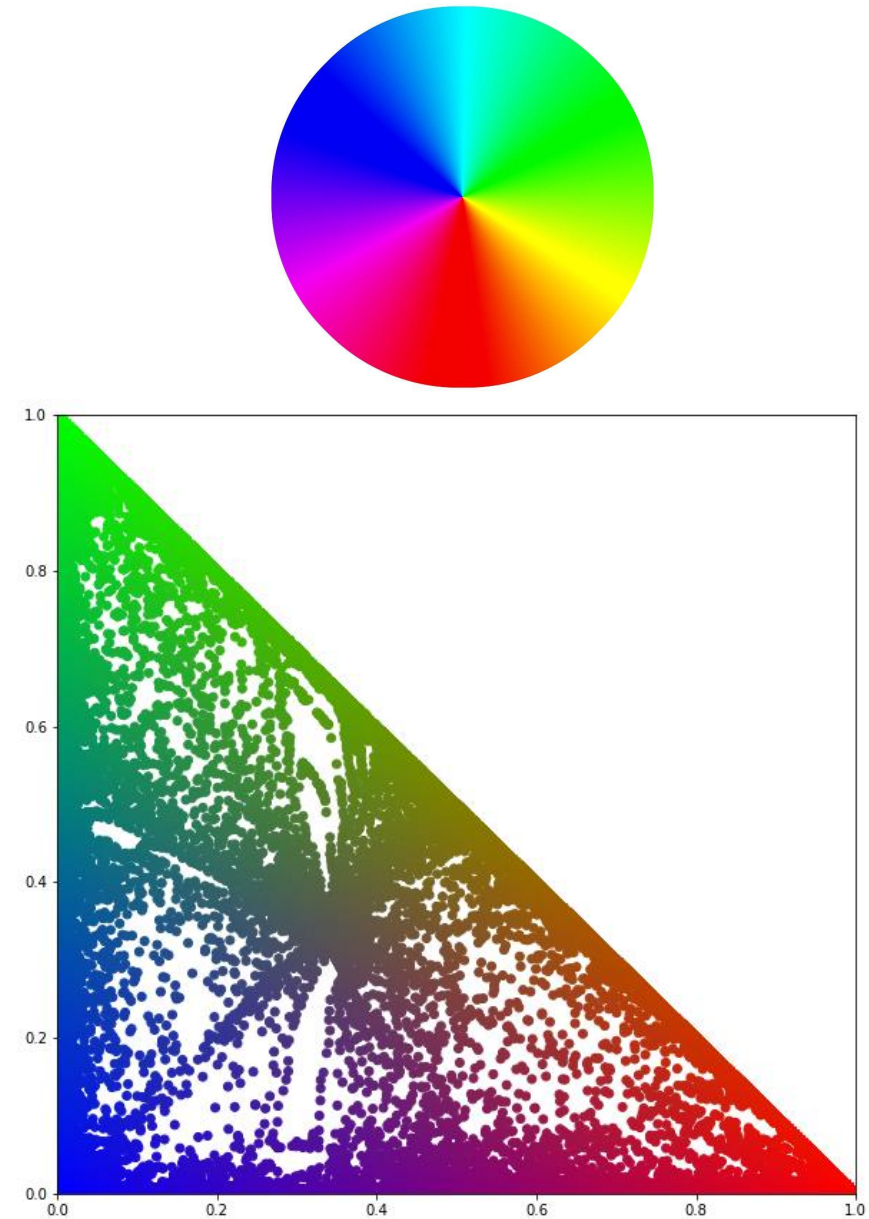
```
file1 = '\cheque.png'
cheque = cv2.imread(path+file1,0) #load image in grayscale
thresholds = [50,75,100,125,150]
for t in thresholds: #set threshold grayscale intensity to remove background
    cheque_tresholded = (cheque<t)*cheque
    plt.figure()
    plt.imshow(cheque_tresholded,cmap='gray')
    plt.title('Threshold: {}'.format(t))
```

Objectives

- Extract the background from an image using grayscale thresholding.
- Transform an image into normalized chromaticity coordinate (NCC) space.
- Perform Gaussian parametric segmentation of colored images.
- Perform non-parametric segmentation of colored images.
- Identify the strengths and weaknesses of parametric and non-parametric segmentation in practical applications.

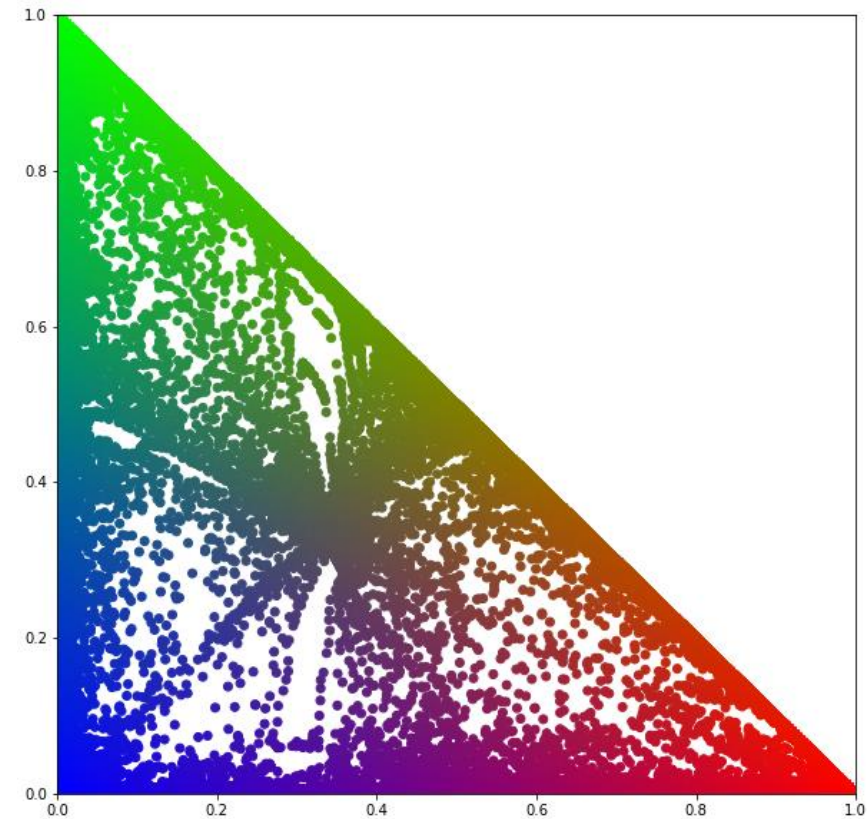
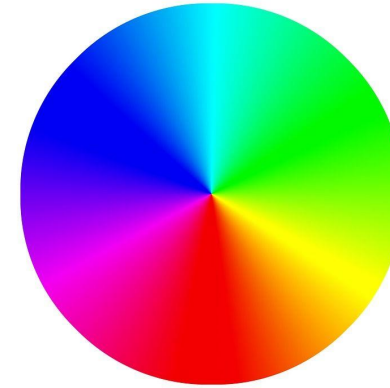
NCC Space

- By dividing each of the RGB channels by the per-pixel intensity $R+G+B$, we obtain the normalized chromaticity coordinates. This is used to account for various shades and reflections of colors when performing color segmentation.
- We plot the normalized reds in the x-axis and normalized greens in the y-axis. The NCC space of a color wheel is shown on the right.



NCC Space

- Note that the spectrum of colors in the color wheel isn't complete, as can be seen in the gaps in the NCC plot. There is a lack of whites and shades of green, red and blue.



NCC Space

#Convert to normalized chromaticity coordinates

```
def ncc( RGB ):
```

```
    r,g= (RGB[:, :,0]/np.sum( RGB,axis=2)), (RGB[:, :,1]/np.sum( RGB,axis=2))
```

```
    b = 1-(r+g)
```

```
    result = np.stack([r,g,b],axis=2)
```

```
    result[result==0] = 100000000
```

```
    return np.stack([r,g,b],axis=2) # Stack R, G, and B arrays along the third  
dimension
```

NCC Space

#Plots only the r and g channels in the NCC of the image.

```
def ncc_plotter(image, size=(10,10), color=True):  
    image_ncc = ncc(image)  
    r,g = image_ncc[:, :, 0], image_ncc[:, :, 1]  
    b=1-(r+g)  
    RGB_flat = image_ncc.reshape(-1,3)  
    plt.figure(figsize=size)  
    if color:  
        plt.scatter(r,g, c=RGB_flat)  
    else:  
        plt.scatter(r,g, c='k')  
    plt.xlim([0,1])  
    plt.ylim([0,1])
```

Objectives

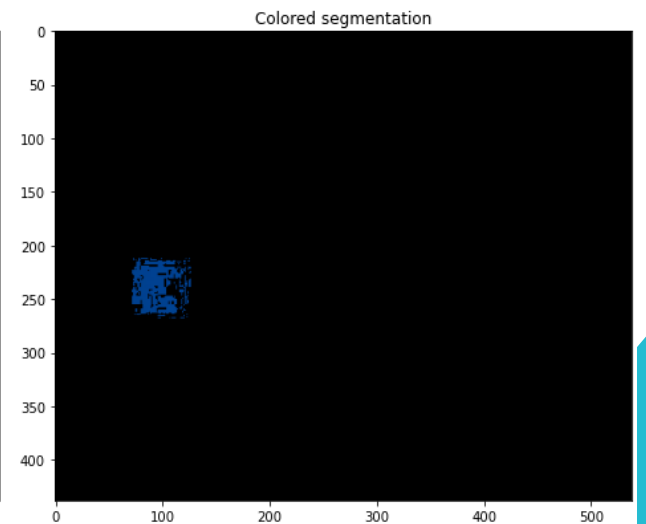
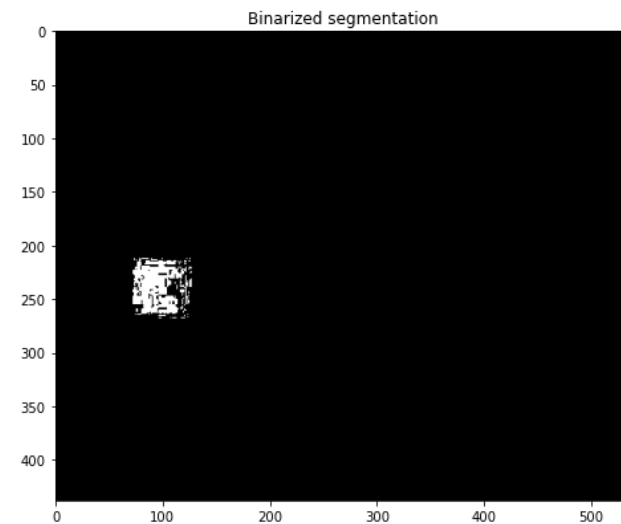
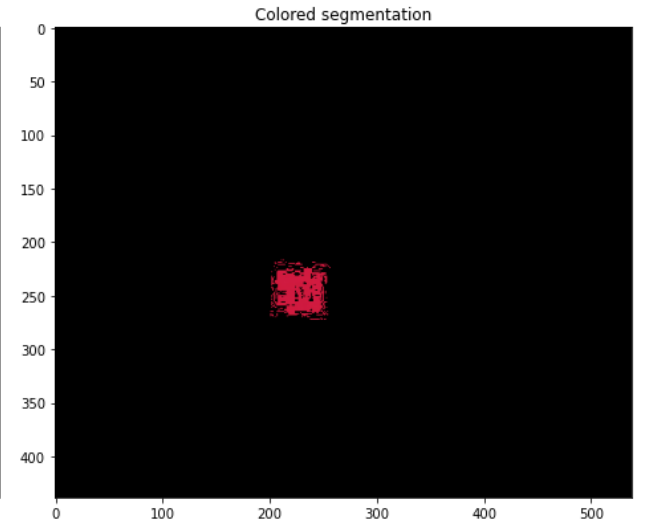
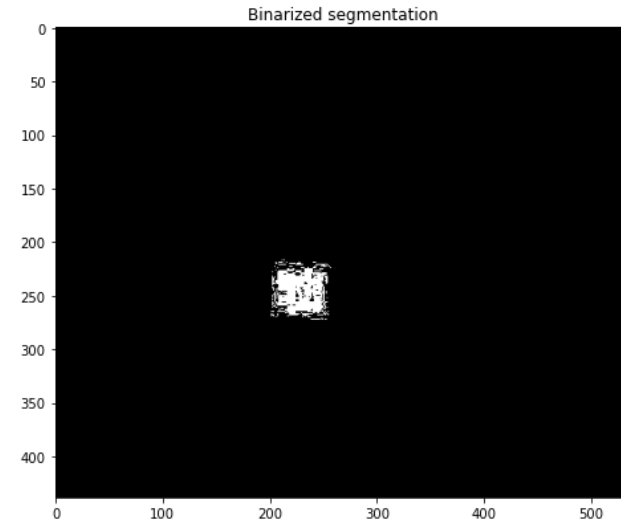
- Extract the background from an image using grayscale thresholding.
- Transform an image into normalized chromaticity coordinate (NCC) space.
- Perform Gaussian parametric segmentation of colored images.
- Perform non-parametric segmentation of colored images.
 - Compare runtimes of brute-force (nested for loop) pixel-by-pixel lookup vs vectorized lookup algorithm.
- Identify the strengths and weaknesses of parametric and non-parametric segmentation in practical applications by testing the algorithms on a color wheel.

Parametric segmentation

To segment an image, we obtain the mean and standard deviations of the r and g channels (in NCC space) of the ROI. Each pixel in the original image is then assigned a value based on how likely it is to be part of the ROI i.e., the “farther” the r and g values of that pixel to the mean of the r and g in the ROI, the less likely it is to be part of the ROI. This likelihood is measured by the Gaussian distribution.

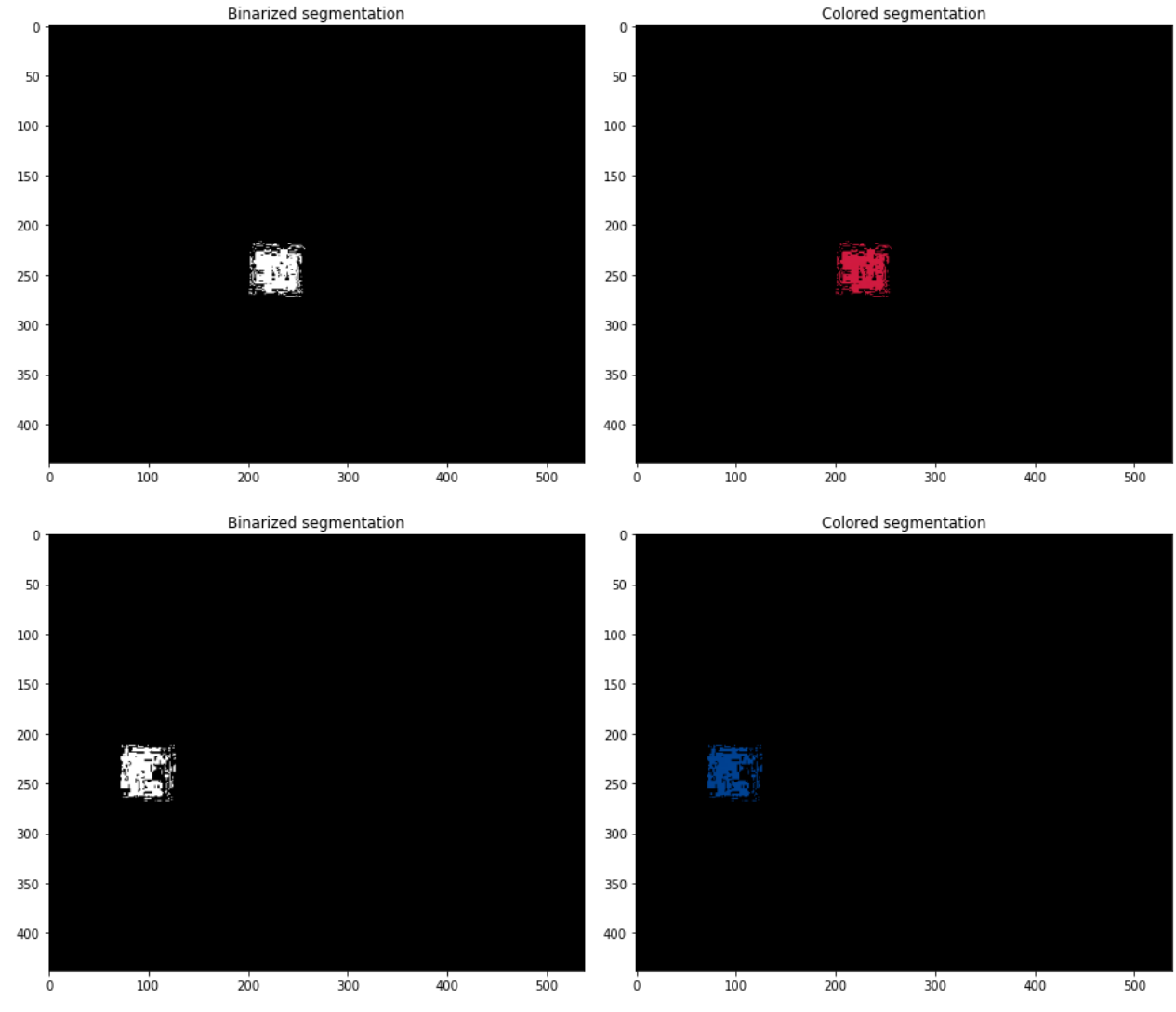
Therefore, this can be thought of as a probabilistic/statistical segmentation.

Parametric Segmentation

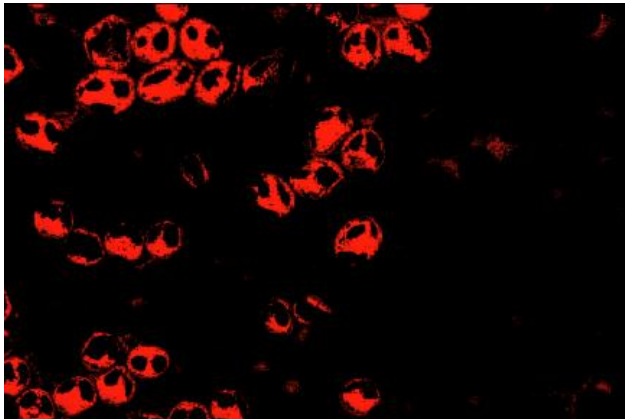


Parametric Segmentation

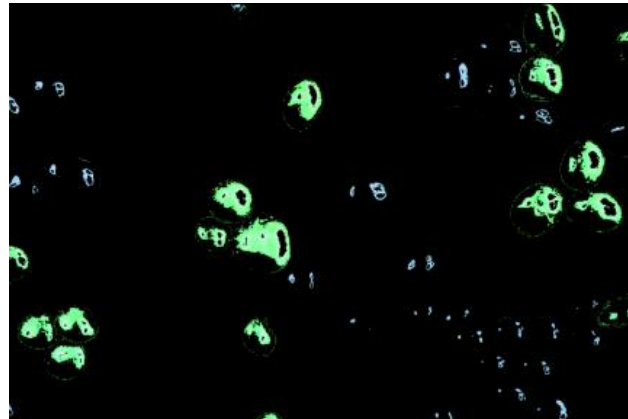
- Note the granularity in the segmentation. In fact, as the size or number of pixels in the ROI increases, the granularity in segmentation is reduced (as I will explain in later slides).
- Spoiler alert: Parametric segmentation is based on statistical segmentation; a greater sample size (i.e., a higher number of pixels in the ROI) will mean a more robust segmentation.



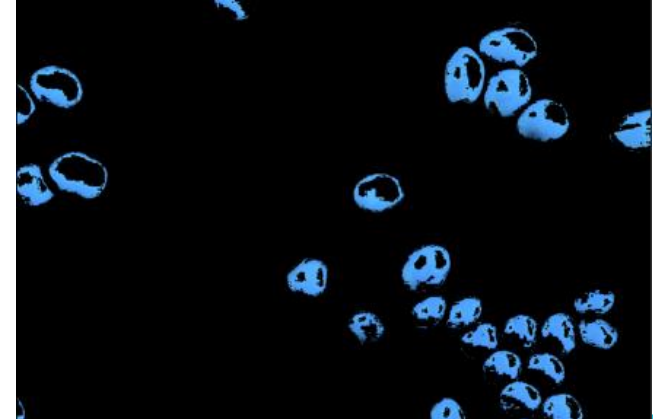
Parametric Segmentation



Reds

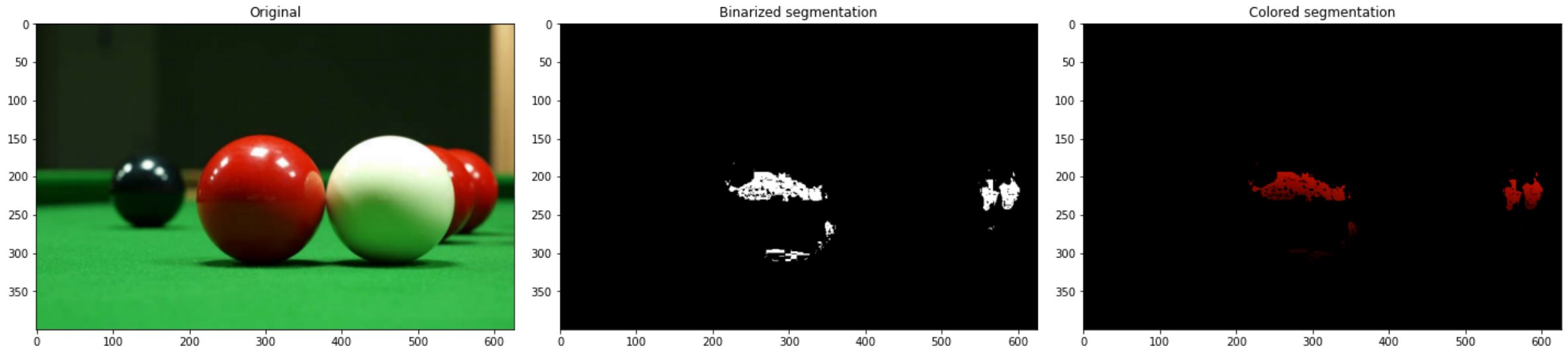


Greens



Blues

Parametric Segmentation



- By itself, the output \mathbf{P} of a Gaussian parametric segmentation is probabilistic i.e., a range of values from 0 to 1. This begs the question: how do we binarize the segmentation so we can restore the color from the segmented image?
- Answer: Use `skimage.filters.threshold_otsu(P[np.isfinite(P)])`
- This is the Python equivalent of Matlab's `imbinarize()`.

Parametric Segmentation

```
def gaussian(mu,sigma, x): #Compute probability of a certain value or pixel
    mu,sigma,x= np.array(mu), np.array(sigma), np.array(x)
    if sigma == 0:
        a = np.zeros_like(x)
        b=a
    else:
        a = (1/(sigma*np.sqrt(2*np.pi)))
        b = np.exp(-(x-mu)**2/(2*sigma**2))
    return a*b
```

Parametric Segmentation

```
def gaussian_segmenter(image, binarized=True):
    ROI_image = ROI(image)
    ROI_ncc = ncc(ROI_image)
    image_ncc = ncc(image)
    r_mean, g_mean = np.mean(ROI_ncc[:,:,:0]), np.mean(ROI_ncc[:,:,:1])
    r_std, g_std = np.std(ROI_ncc[:,:,:0]), np.std(ROI_ncc[:,:,:1])
    P = gaussian(r_mean, r_std, image_ncc[:,:,:0])*gaussian(g_mean, g_std,
image_ncc[:,:,:1])
    if binarized:
        threshold_value = filters.threshold_otsu(P[np.isfinite(P)])    #handle only
finite values, remove NaNs
        P = P > threshold_value
    return P

def colored_gaussian_segmenter(image):
    P = gaussian_segmenter(image, binarized=True)
    segmented_image = np.stack([P*image[:,:,:0], P*image[:,:,:1], P*image[:,:,:2]], axis=2)
    return segmented_image
```

Objectives

- Extract the background from an image using grayscale thresholding.
- Transform an image into normalized chromaticity coordinate (NCC) space.
- Perform Gaussian parametric segmentation of colored images.
- Perform non-parametric segmentation of colored images.
 - Compare runtimes of brute-force (nested for loop) pixel-by-pixel lookup vs vectorized lookup algorithm.
- Identify the strengths and weaknesses of parametric and non-parametric segmentation in practical applications.

Non-parametric segmentation

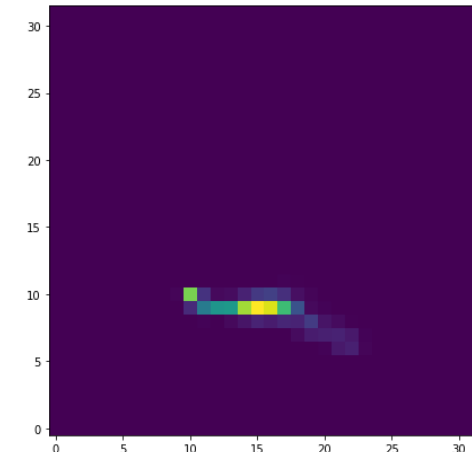
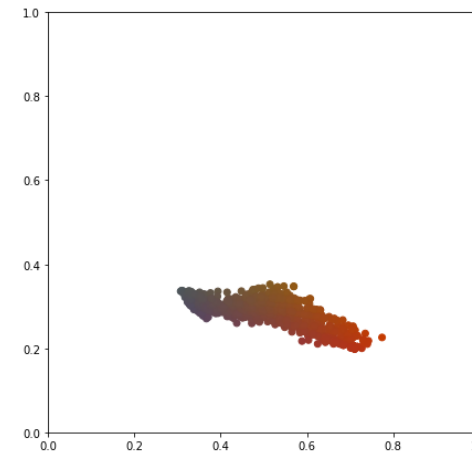
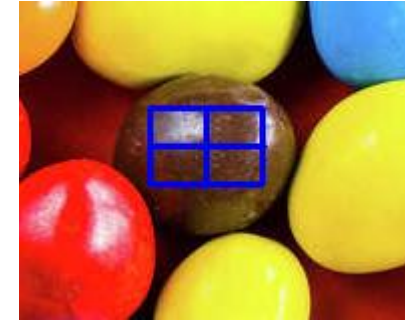
Non-parametric segmentation does not require the fitting of a Gaussian or the calculation of the the mean and standard deviation of the r and g channels of the ROI.

Here are the general steps in performing non-parametric segmentation.

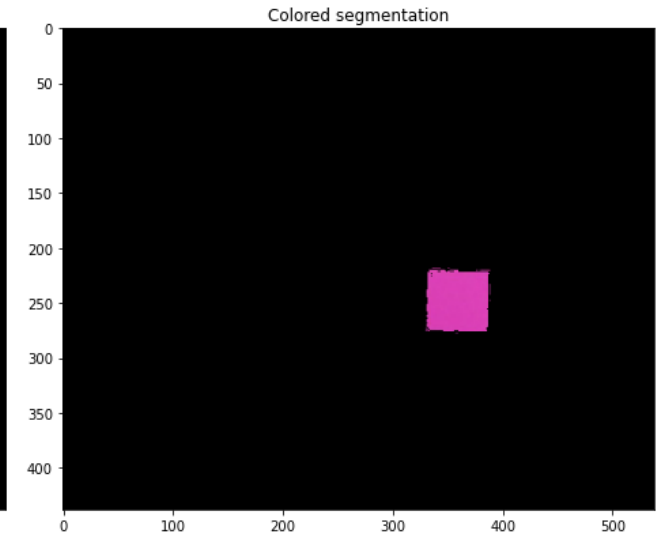
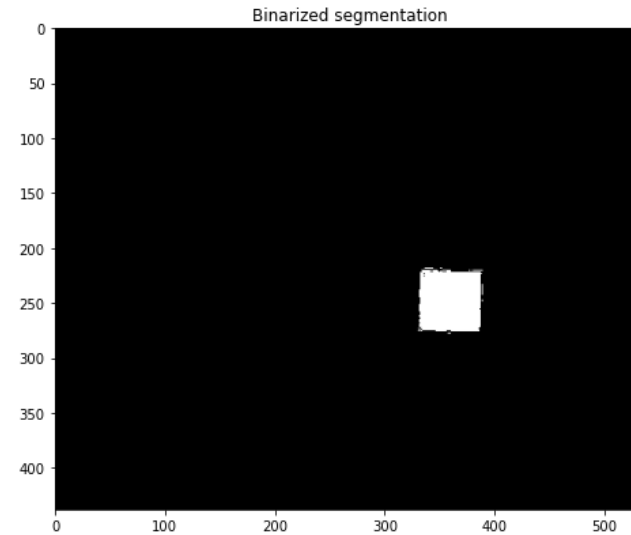
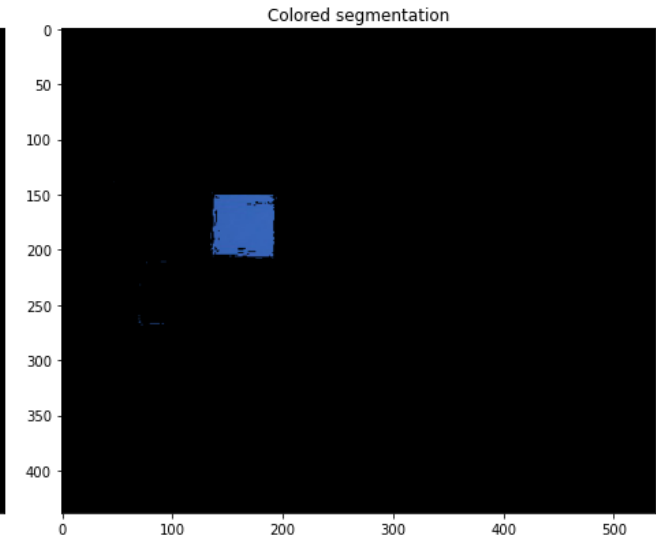
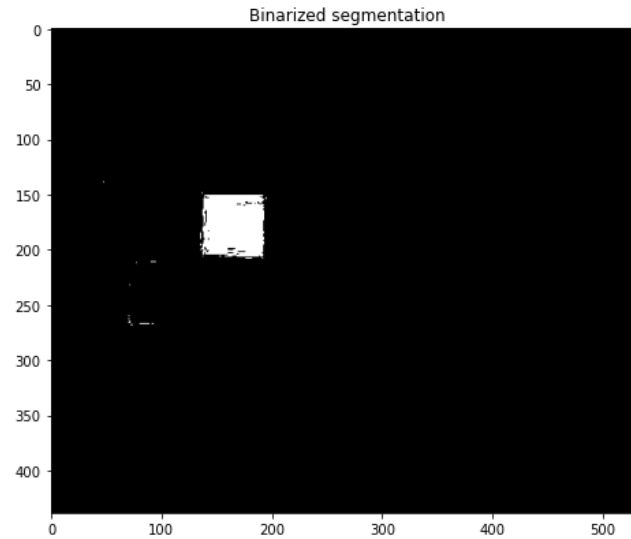
1. Create a 2D histogram of the ROI in NCC (r in y axis, g in x axis).
2. Per pixel of the image, obtain the r and g in NCC space.
3. Obtain the histogram count of the corresponding (r,g)
4. Assign a grayscale value equal to that count.

2D Histogram

- The indices of a 2D histogram array are the r and g (NCC) bins. The value at each point corresponds to a pixel count in the ROI.
- It should look similar to the NCC space plot of the ROI. From the 2D histogram, we can see which pixels (in NCC) are abundant in the ROI.
- From the results, it can be seen that the 2D histogram indeed looks like the NCC plot of the ROI.

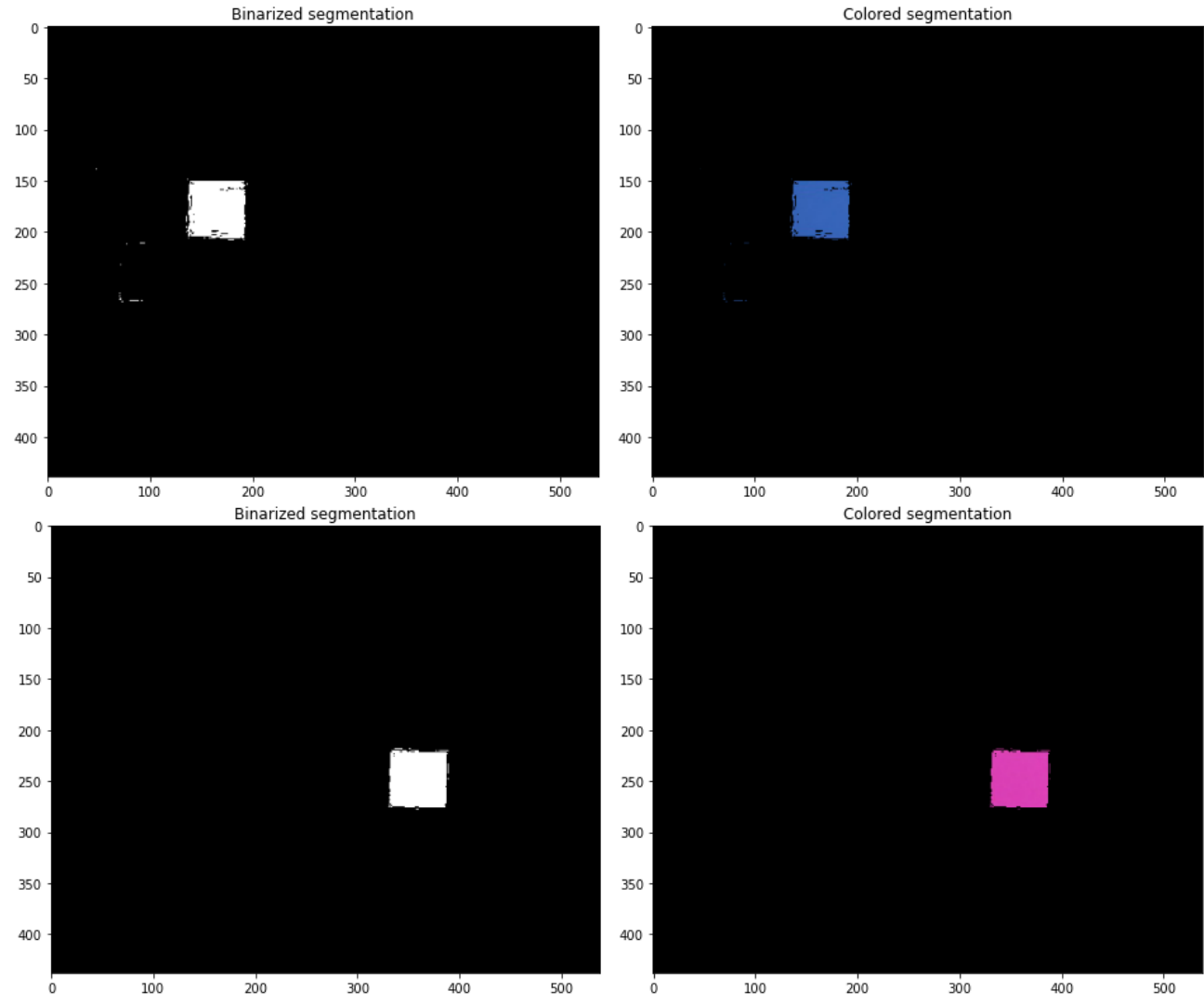


Non-parametric Segmentation

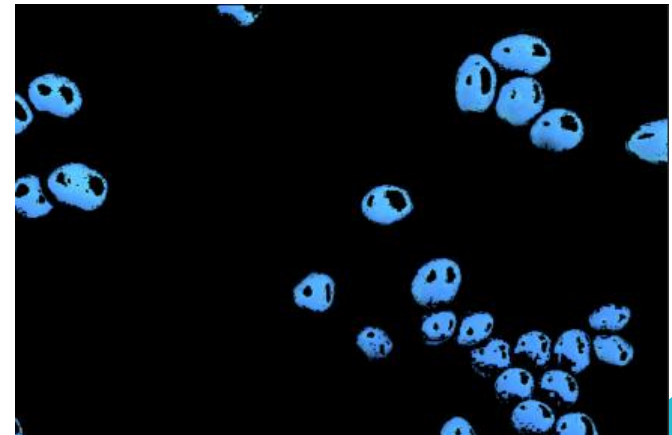
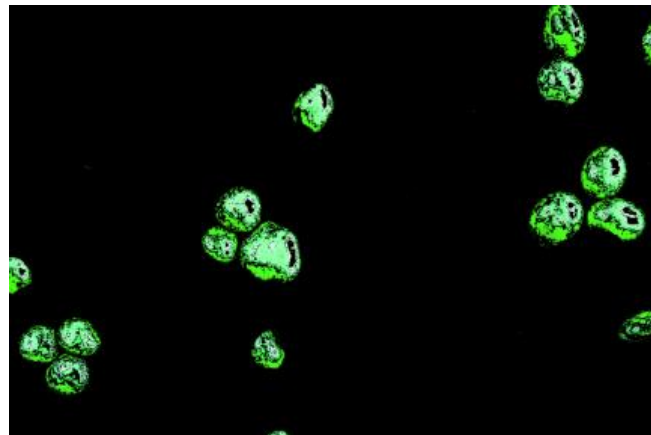
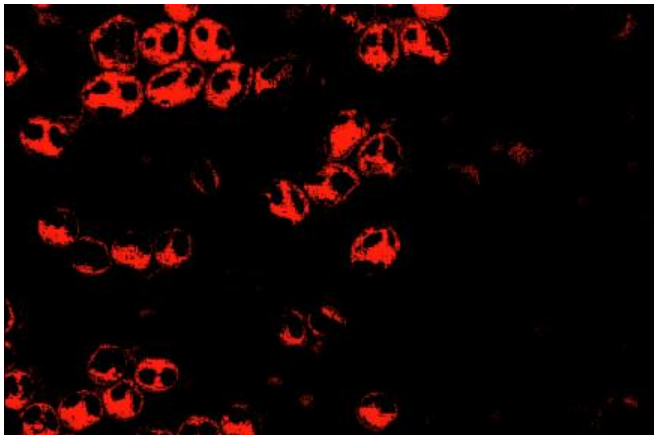


Non-parametric Segmentation

- Here, we use a bin size of 32.
- We will see later that non-parametric segmentation often resolves solid colors better than parametric segmentation especially when the ROI pixel count is low.
- There is less granularity in the segmented images when using non-parametric segmentation.



Non-parametric Segmentation



Objectives

- Extract the background from an image using grayscale thresholding.
- Transform an image into normalized chromaticity coordinate (NCC) space.
- Perform Gaussian parametric segmentation of colored images.
- Perform non-parametric segmentation of colored images.
 - Compare runtimes of brute-force (nested for loop) pixel-by-pixel lookup vs vectorized lookup algorithm.
- Identify the strengths and weaknesses of parametric and non-parametric segmentation in practical applications.

Non-parametric Segmentation Algorithms

- One way to implement the lookup search is by brute-force (pixel-by-pixel) assignment of values (as done in the module). This means using a nested for loop.

```
bins=32
x, y= ROI_ncc[:, :, 0].flatten(), ROI_ncc[:, :, 1].flatten()
H = np.histogram2d(y,x, bins=bins, range=[[0,1],[0,1]])

M,N = np.shape(image_ncc[:, :, 0])
P = np.zeros_like(image_ncc[:, :, 0])

for i in range(M):
    for j in range(N):
        i_index = int(image_ncc[i,j,0]*(bins-1) + 1)
        j_index = int(image_ncc[i,j,1]*(bins-1) + 1)
        if i_index == bins:
            i_index = bins-1
        if j_index == bins:
            j_index = bins-1
        P[i_index, j_index] = H[0][i_index, j_index]
if binarized:
    threshold_value = filters.threshold_otsu(P[np.isfinite(P)]) #handle only finite values, remove NaNs
    P = P > threshold_value
```

Non-parametric Segmentation Algorithms

- However, this can be made more efficient. We use a cleaner, vectorized implementation using `numpy.digitize()` to automate the bin lookup.

```
bins=32
x, y= ROI_ncc[:, :, 0].flatten(), ROI_ncc[:, :, 1].flatten()
H = np.histogram2d(y,x, bins=bins, range=[[0,1],[0,1]])

M,N = np.shape(image_ncc[:, :, 0])
P = np.zeros_like(image_ncc[:, :, 0])

#Vectorized look-up algorithm based on 2D histogram
x_indices = np.digitize(image_ncc[:, :, 0], H[1]) - 1
y_indices = np.digitize(image_ncc[:, :, 1], H[2]) - 1

# Prevent possible IndexError by clipping the values to the valid index ranges.
x_indices = np.clip(x_indices, 0, bins-1)
y_indices = np.clip(y_indices, 0, bins-1)

P = H[0][y_indices, x_indices]
if binarized:
    threshold_value = filters.threshold_otsu(P[np.isfinite(P)]) #handle only finite values, remove NaNs
    P = P > threshold_value
```

Runtime Comparisons

- Test 1: Billiard image (image size: 626x400, ROI size: 50x50, bin size = 32)

Runtime using nested for loops: 2.323850631713867 s
Runtime using vectorized look-up: 0.02889847755432129 s

- Test 2: M&Ms (image size: 1200x800, ROI size: 50x50, bin size = 32).

Runtime using nested for loops: 3.07297945022583 s
Runtime using vectorized look-up: 0.1066887378692627 s

- Test 3: Billiard image (image size: 626x400, ROI size: 100x100, bin size = 32)

Runtime using nested for loops: 2.23114275932312 s
Runtime using vectorized look-up: 0.02847456932067871 s

- Test 4: Billiard image (image size: 626x400, ROI size: 50x50, bin size = 64)

Runtime using nested for loops: 2.275197744369507 s
Runtime using vectorized look-up: 0.027926921844482422 s

Conclusion: The vectorized implementation is faster than the brute-force implementation regardless of image size, ROI size and bin size by roughly two orders of magnitude. This shouldn't be surprising.

Function implementation

```
def nonparametric_segmenter(image, ROI_image, bins=32, binarized=True):
    image_ncc, ROI_ncc = ncc(image), ncc(ROI_image)

    #2D histogram of ROI
    x, y= ROI_ncc[:, :, 0].flatten(), ROI_ncc[:, :, 1].flatten()
    H = np.histogram2d(y,x, bins=bins, range=[[0,1],[0,1]])

    #Vectorized look-up algorithm based on 2D histogram
    x_indices = np.digitize(image_ncc[:, :, 0], H[1]) - 1
    y_indices = np.digitize(image_ncc[:, :, 1], H[2]) - 1

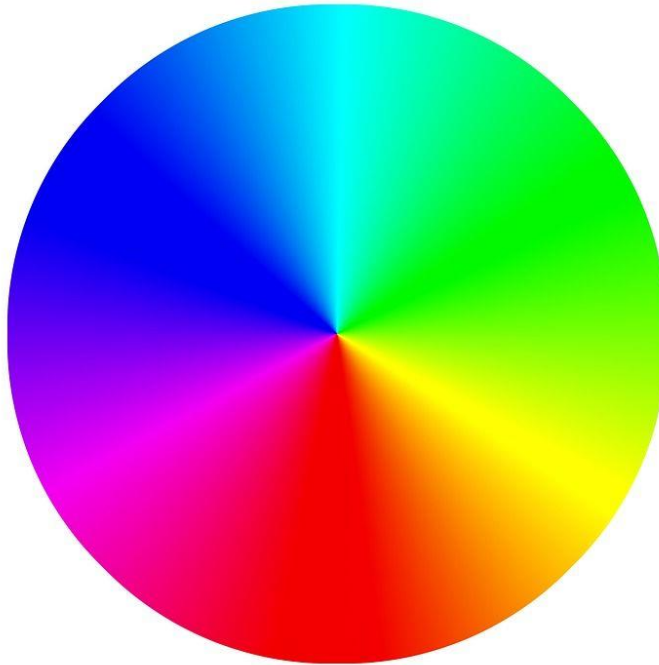
    # Prevent possible IndexError by clipping the values to the valid index ranges.
    x_indices = np.clip(x_indices, 0, bins-1)
    y_indices = np.clip(y_indices, 0, bins-1)

    P = H[0][y_indices, x_indices]
    if binarized:
        threshold_value = filters.threshold_otsu(P[np.isfinite(P)])    #handle only finite values, remove NaNs
        P = P > threshold_value
    return P
```

Objectives

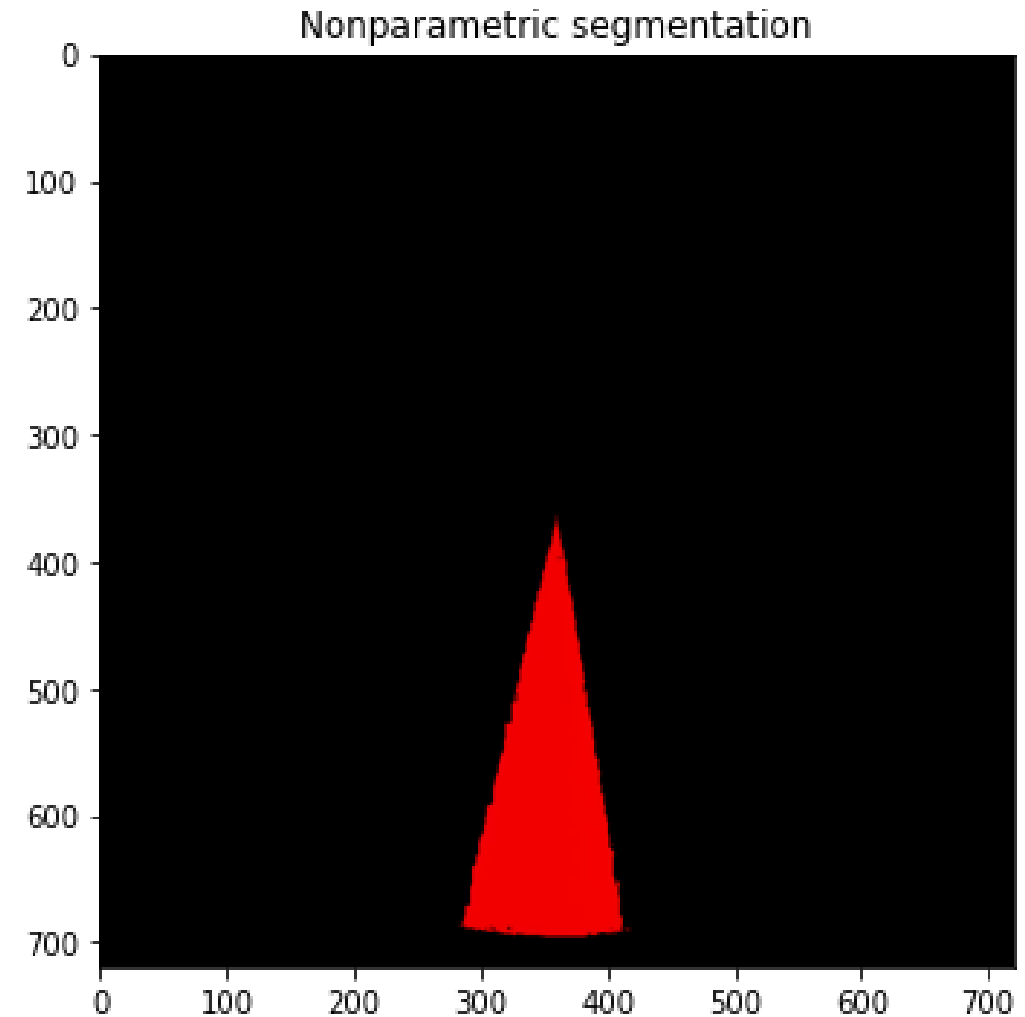
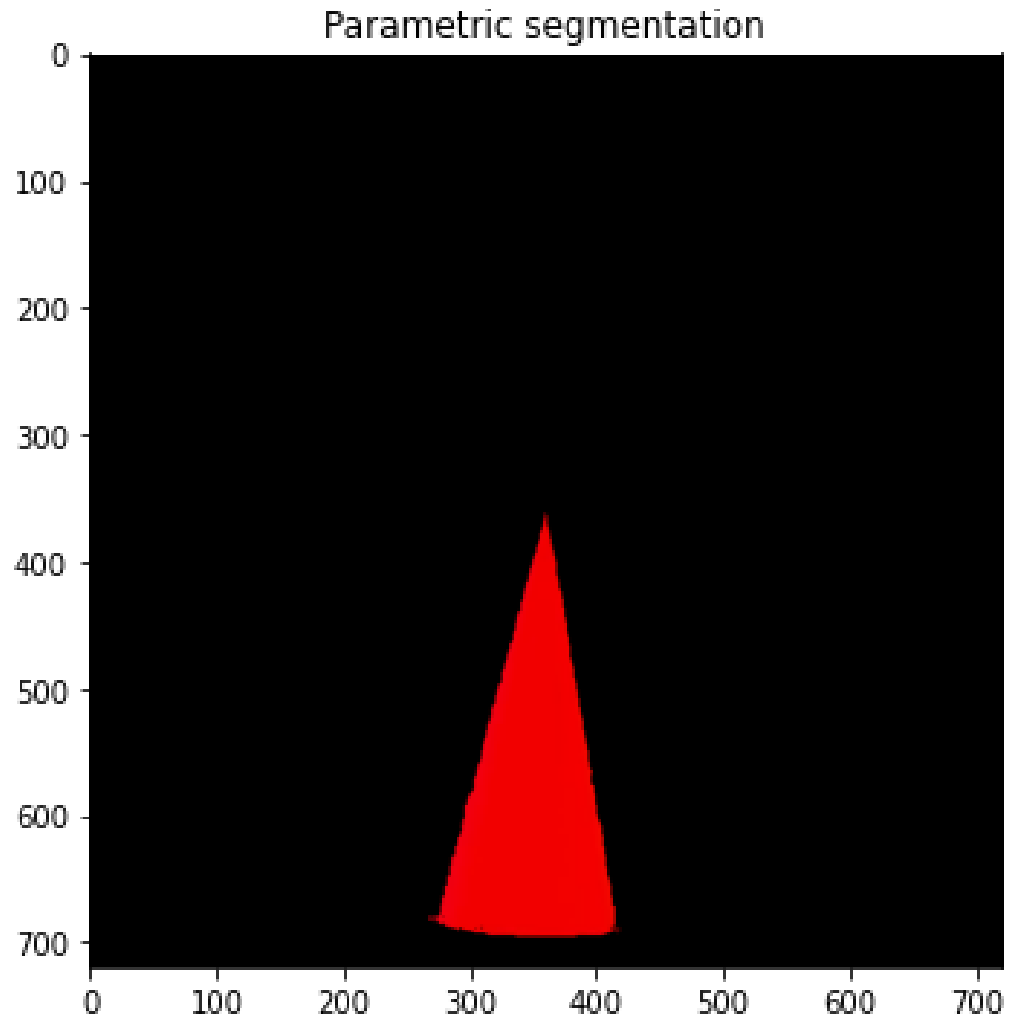
- Extract the background from an image using grayscale thresholding.
- Transform an image into normalized chromaticity coordinate (NCC) space.
- Perform Gaussian parametric segmentation of colored images.
- Perform non-parametric segmentation of colored images.
 - Compare runtimes of brute-force (nested for loop) pixel-by-pixel lookup vs vectorized lookup algorithm.
- Identify the strengths and weaknesses of parametric and non-parametric segmentation in practical applications.

Comparison using the Color Wheel

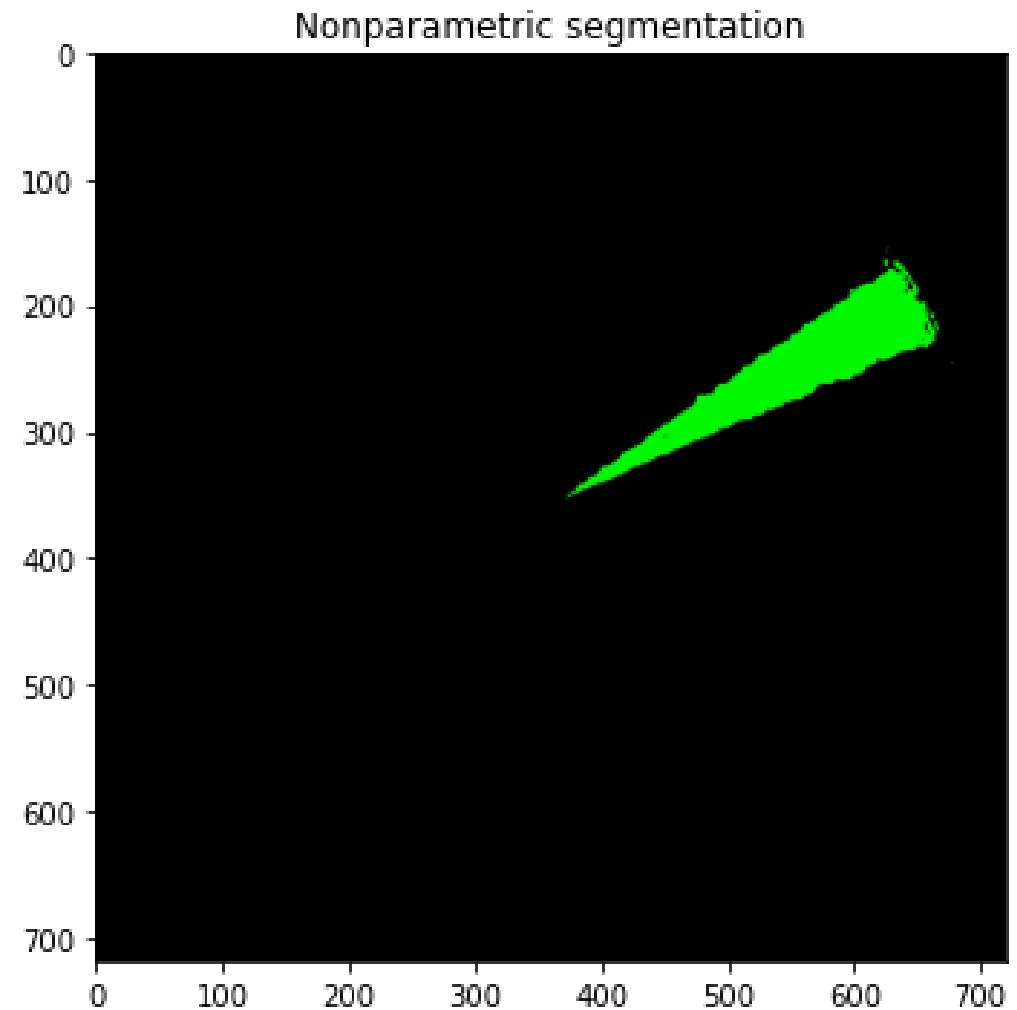
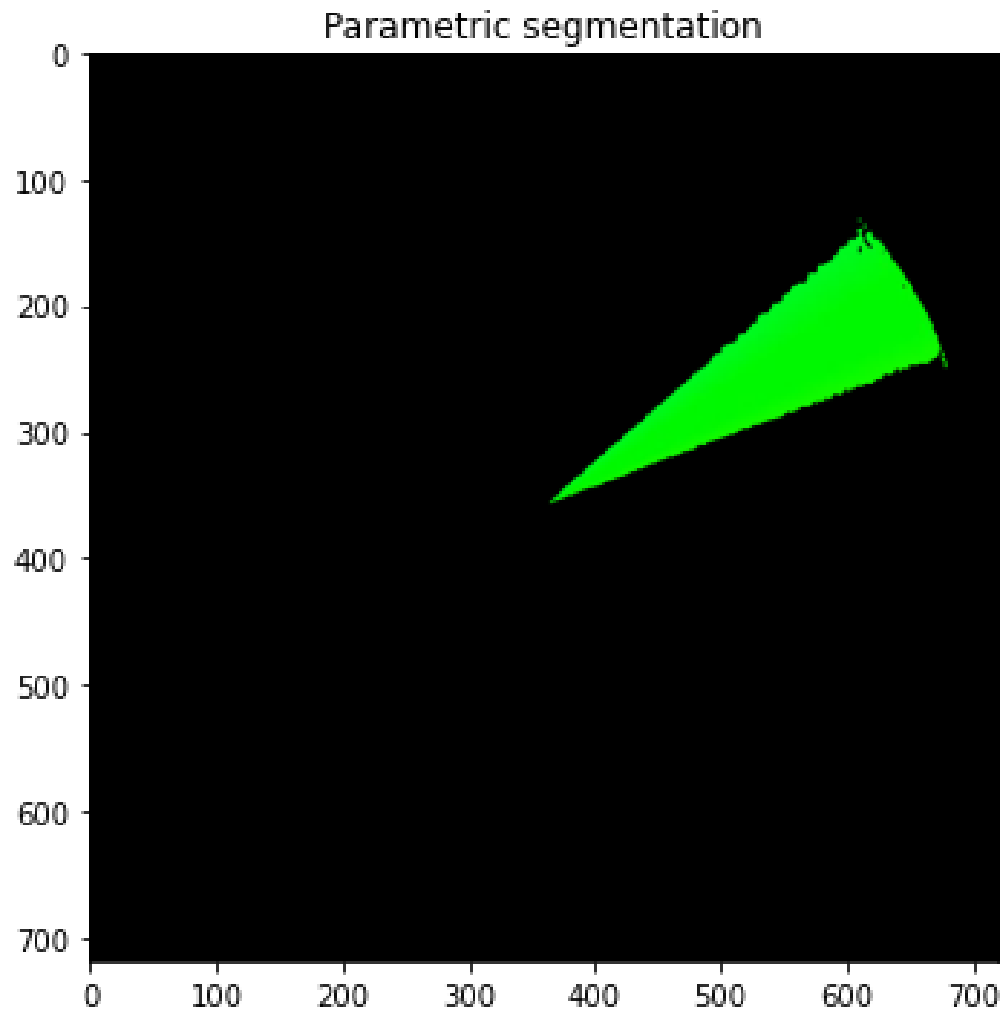


- We test both segmentation methods based on their capabilities to discern color/hue, saturation, and discern color gradients (analogous colors).
- We also test the effect of ROI size for both methods.
- For non-parametric segmentation, we investigate the effect of increasing bin size.

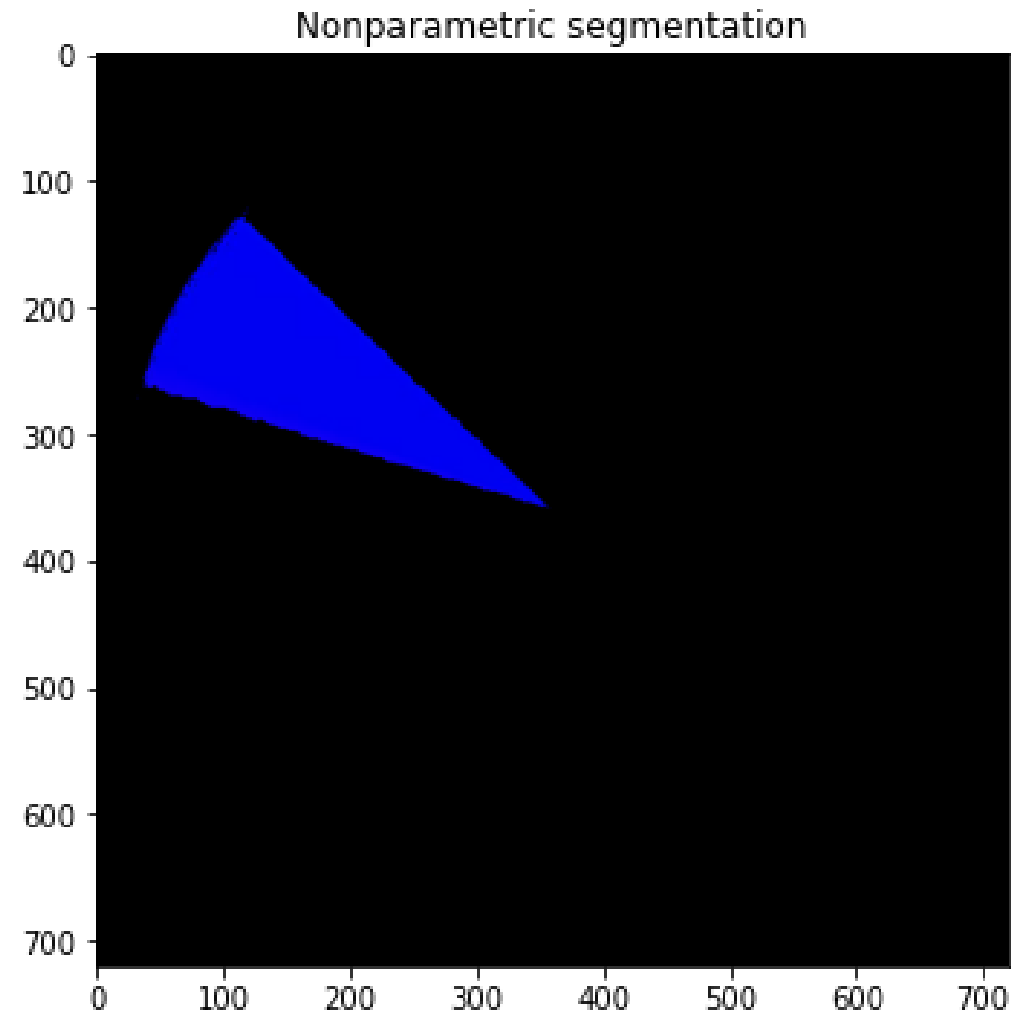
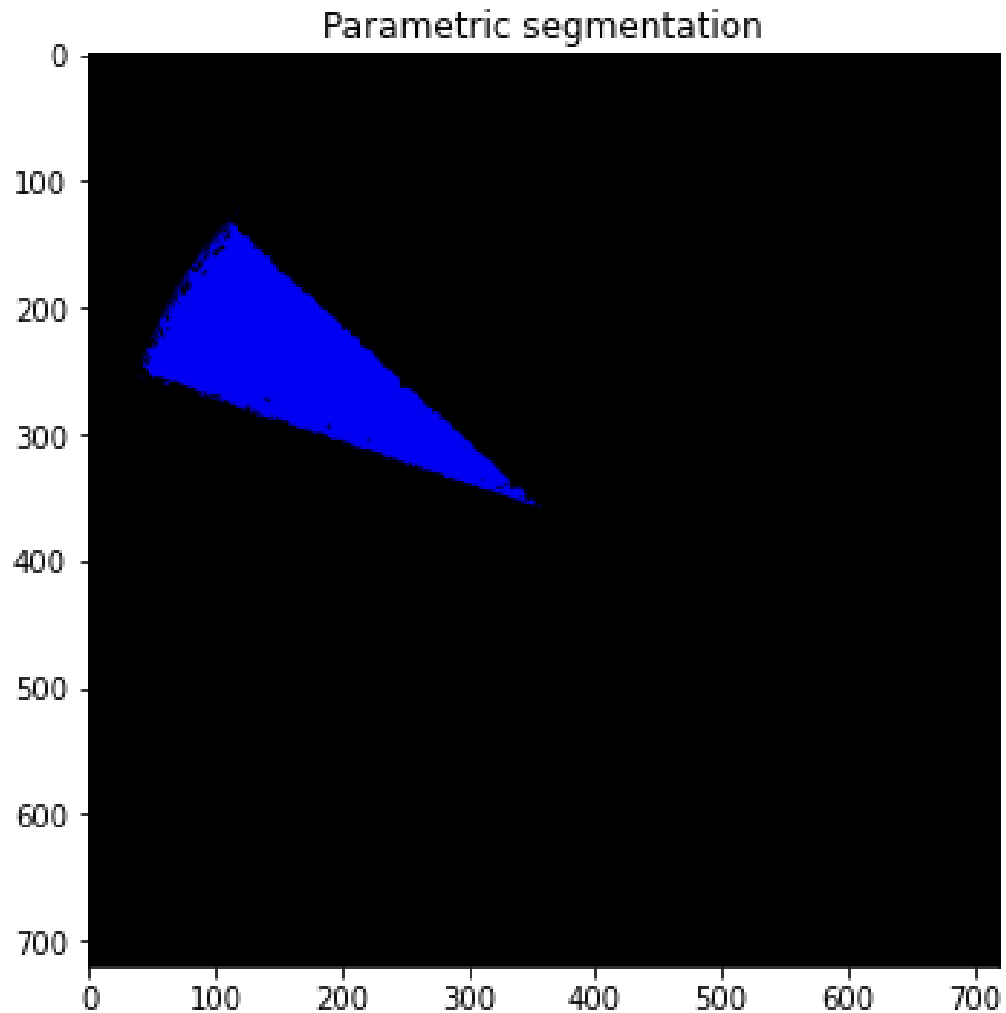
Comparison 1: RGB Discernment



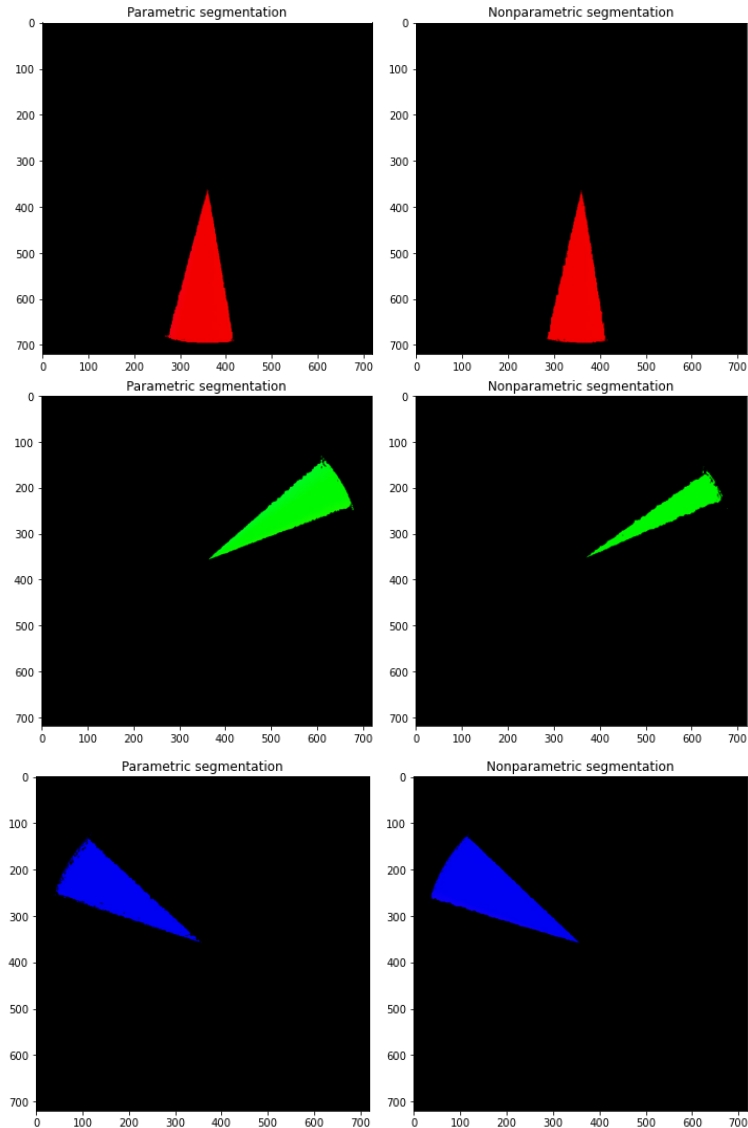
Comparison 1: RGB Discernment



Comparison 1: RGB Discernment



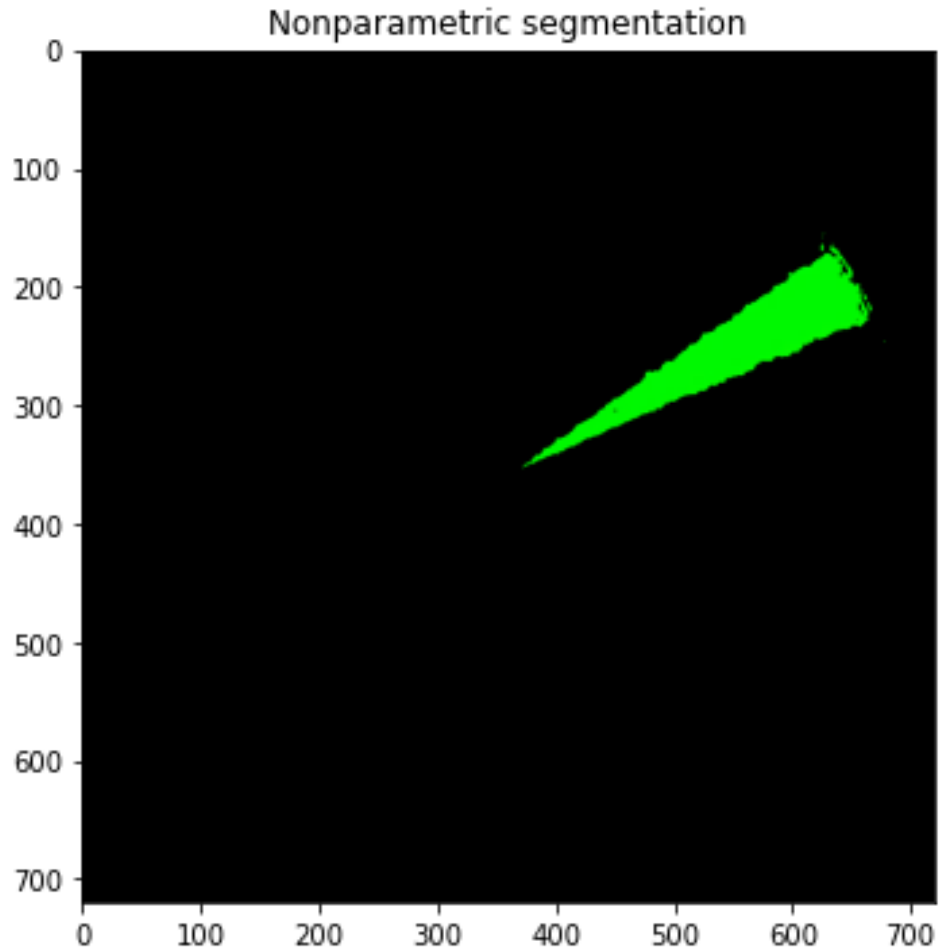
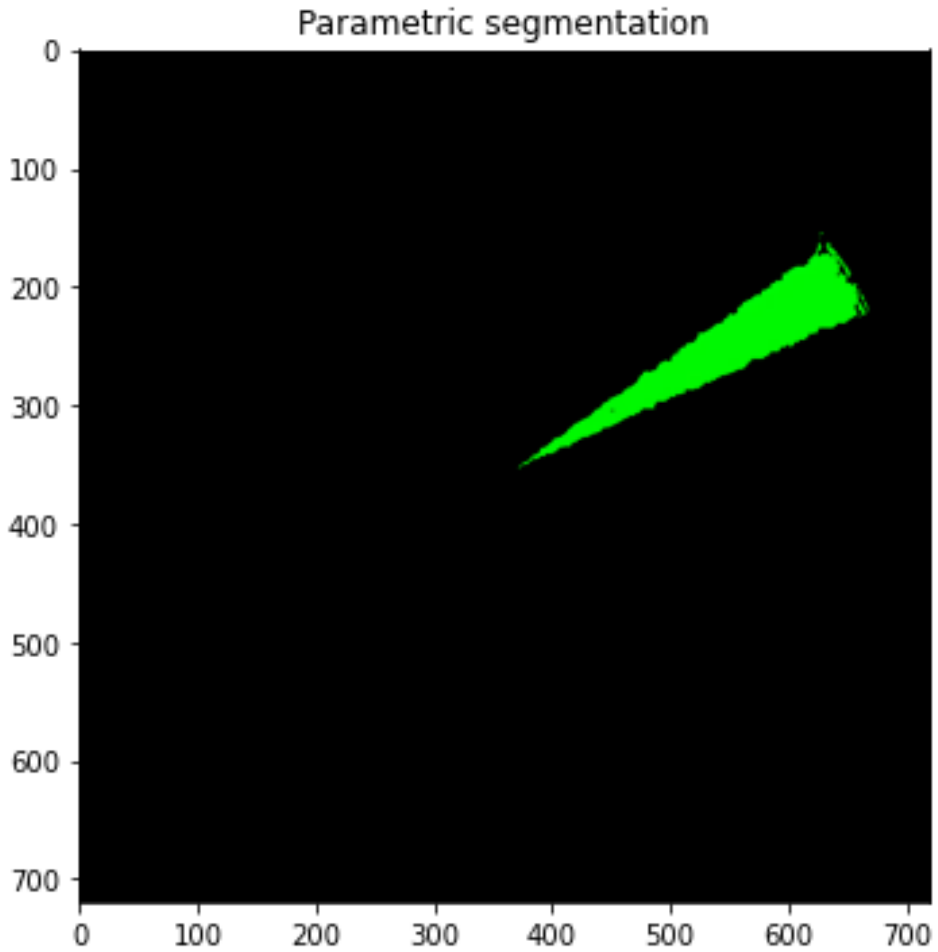
Comparison 1: RGB Discernment



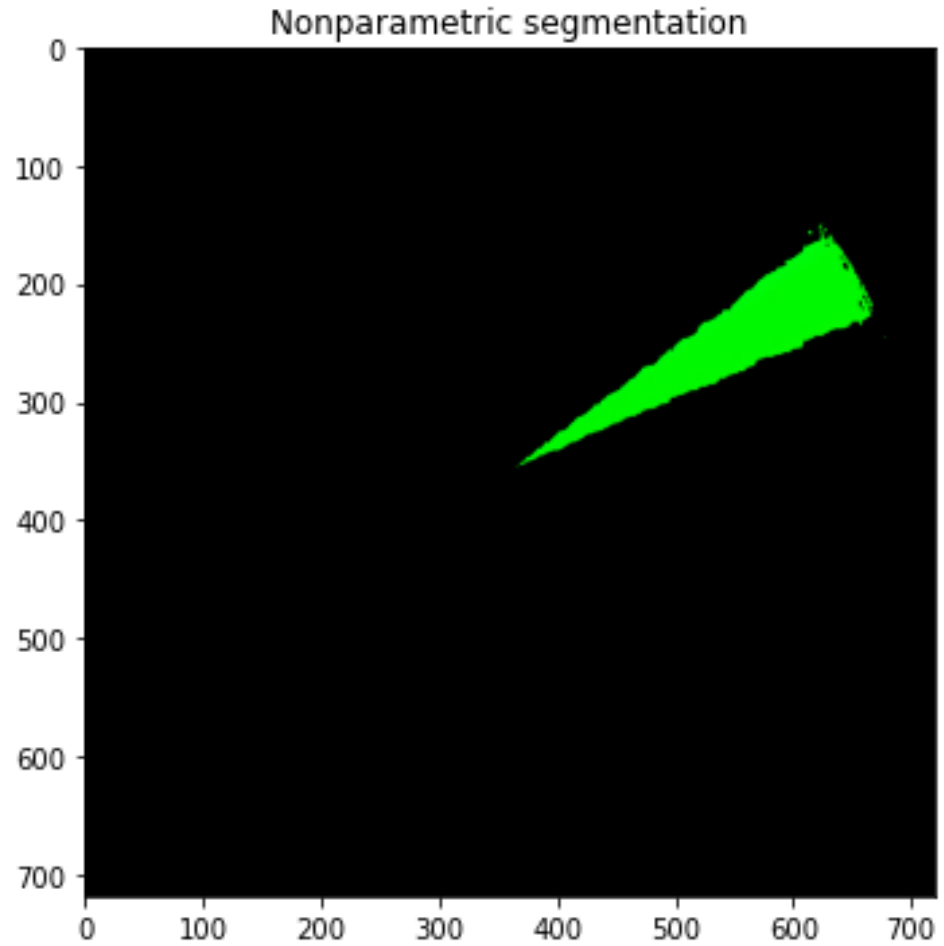
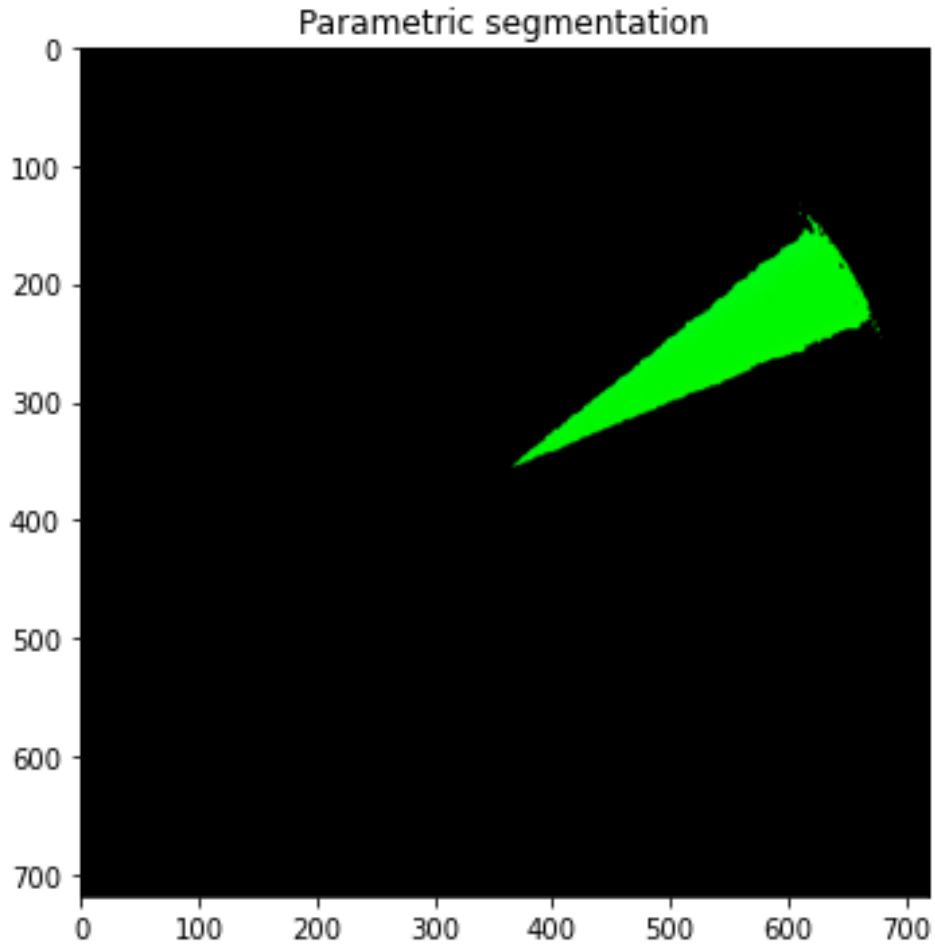
Under constant ROI (100x100) and a bin size of 32 for the non-parametric algorithm, both methods are able to discern RGB almost to the same degree.

Interestingly, slightly more greens were segmented by the parametric algorithm compared to the non-parametric algorithm.

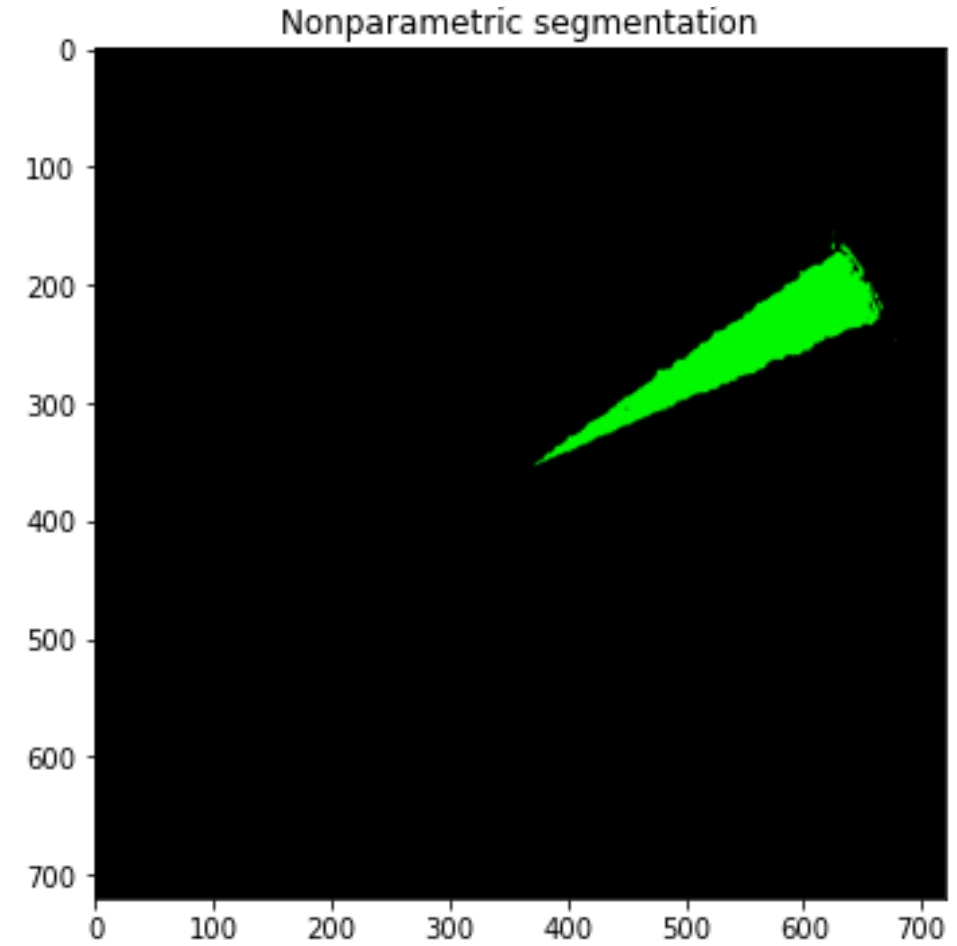
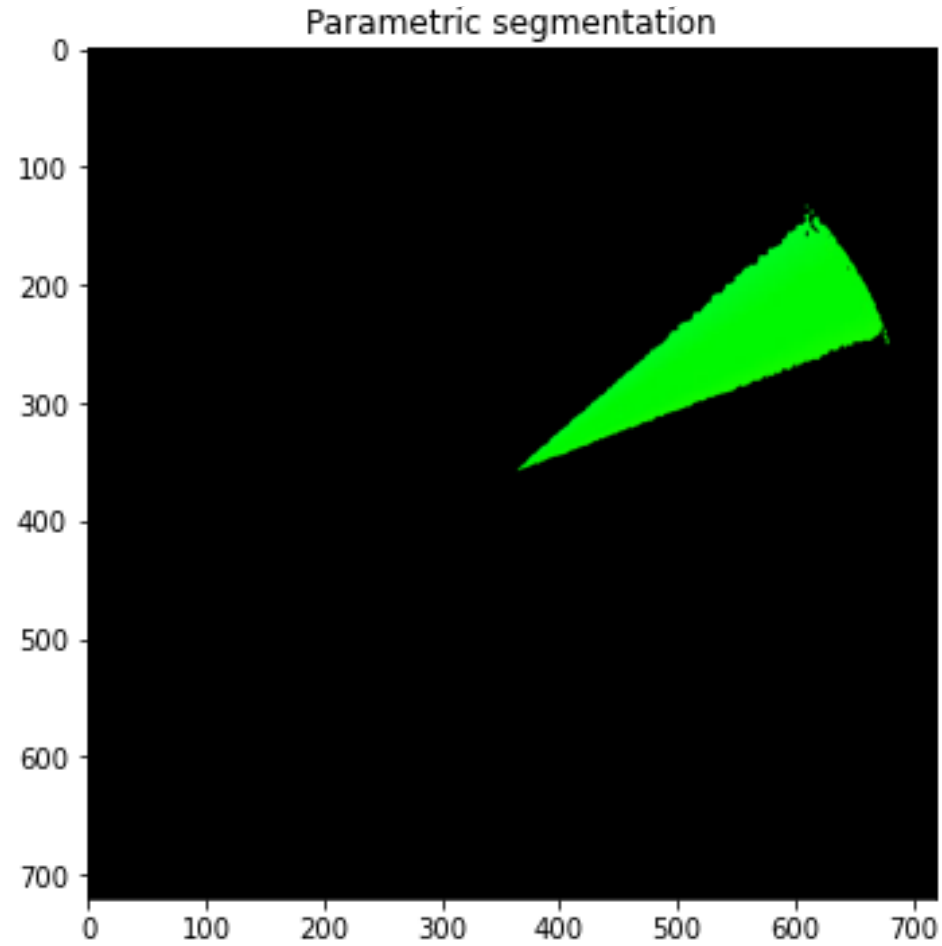
Comparison 2: ROI Size (50x50)



Comparison 2: ROI Size (80x80)



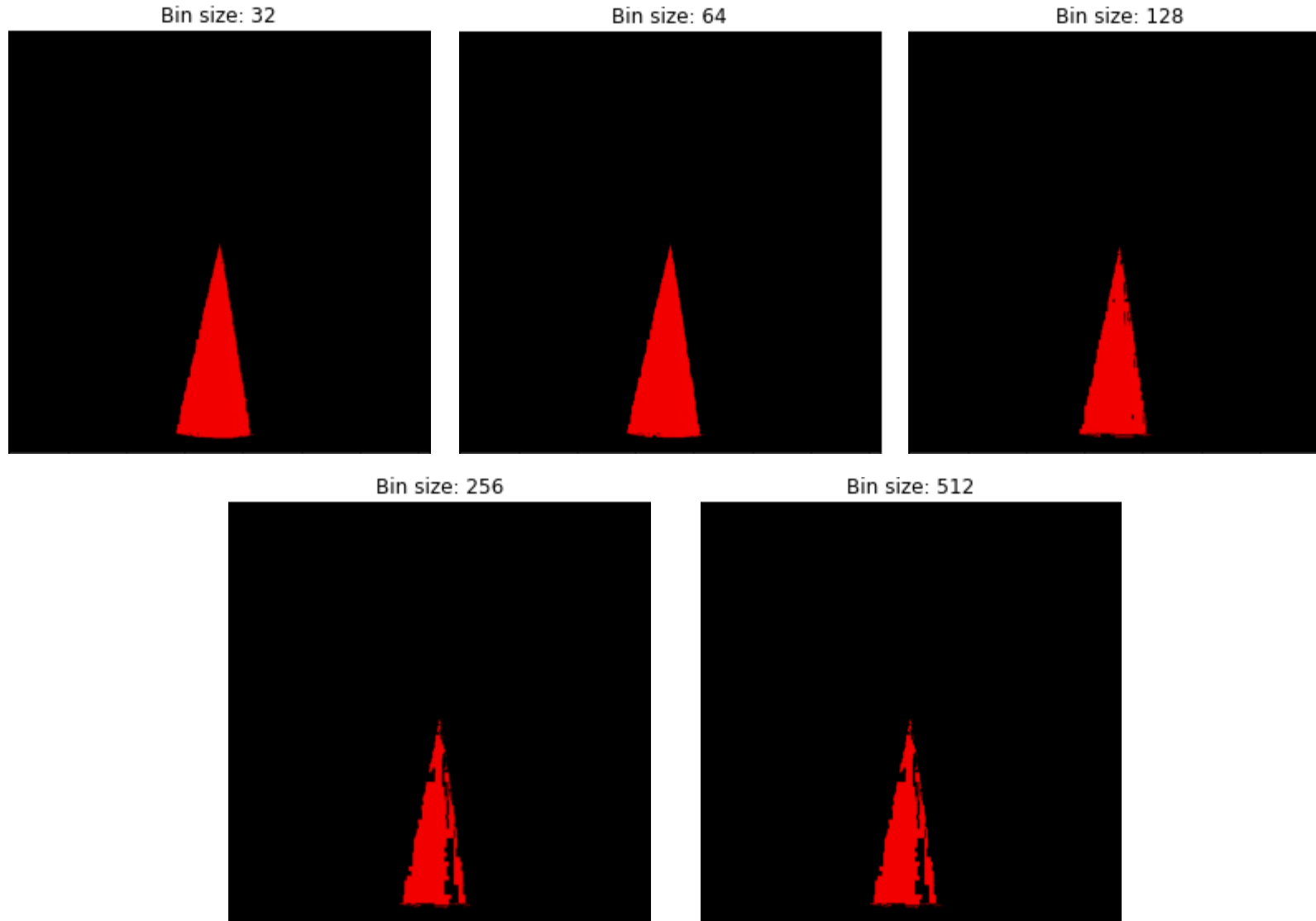
Comparison 2: ROI Size (80x80)



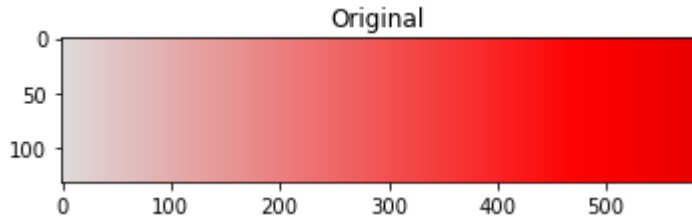
Comparison 2: ROI Size

As the ROI pixel count increases, the parametric segmentation algorithm becomes more robust and accurate, as shown by the increase in the angular width of the green hue. Parametric segmentation works best with larger ROI sizes due to its statistical nature (think of the sample size as the pixel count of the ROI).

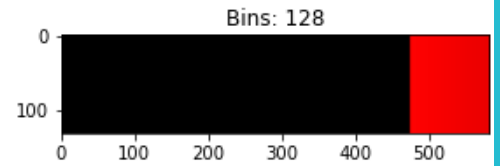
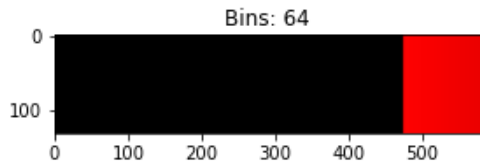
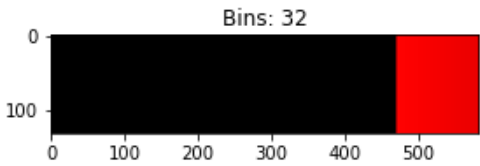
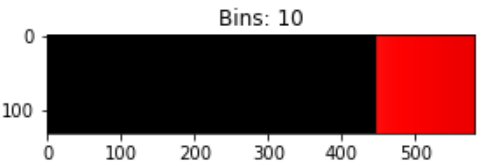
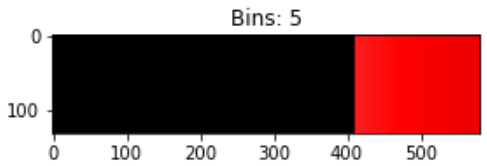
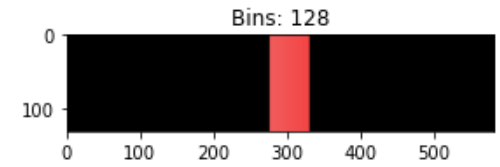
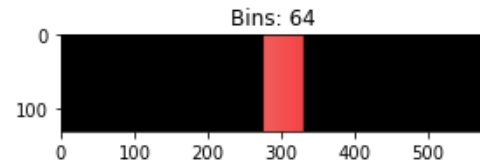
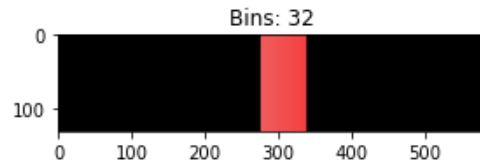
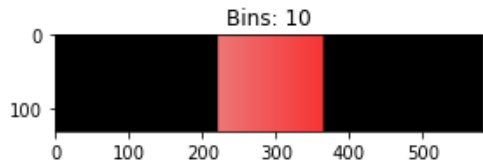
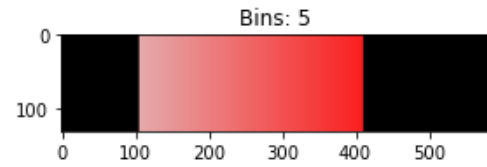
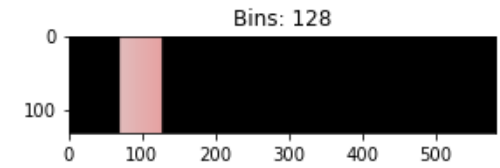
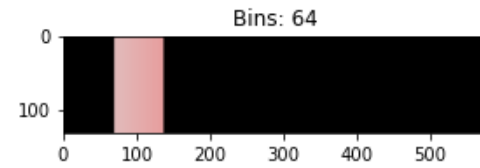
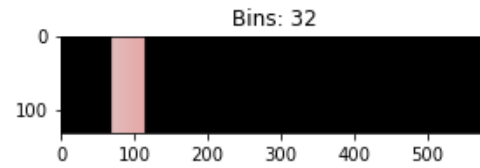
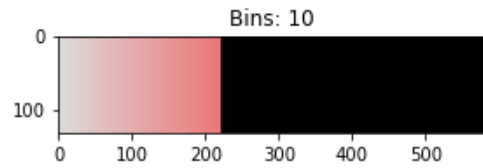
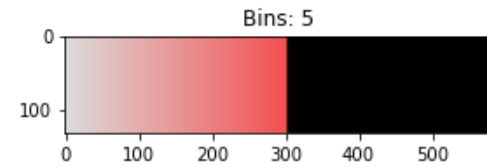
Comparison 3: Bin Size (Non-parametric)



Comparison 3: Bin Counts and Saturation Range (Non-parametric)



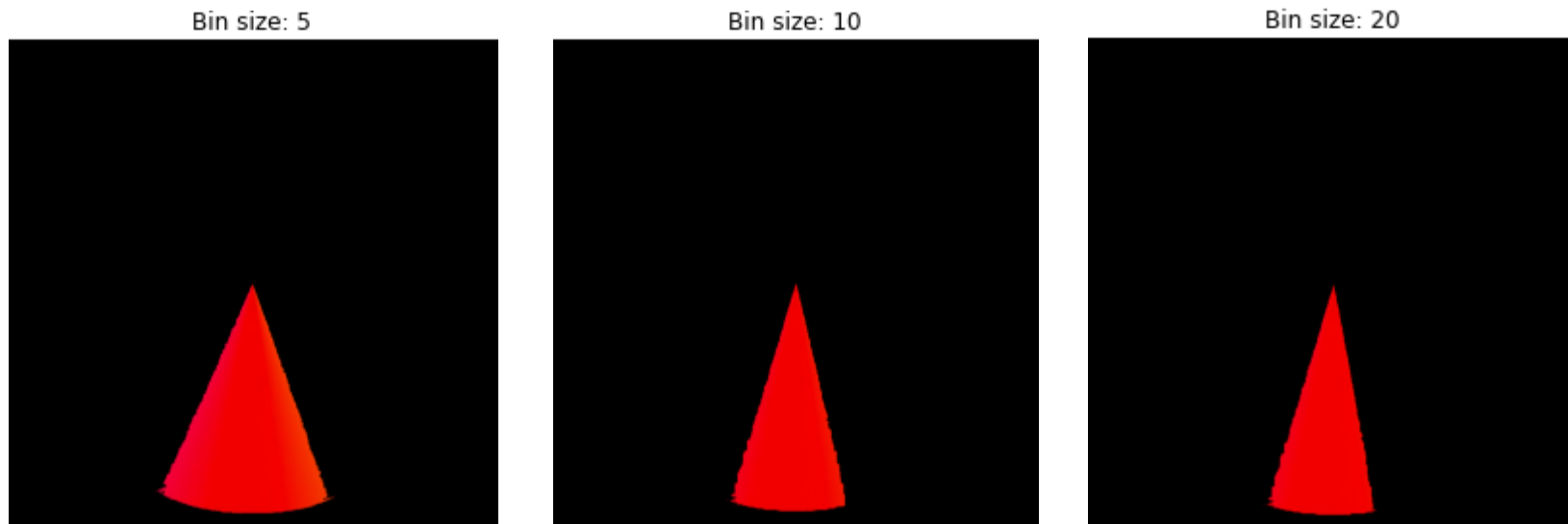
`ROI_images=[image[50:100, 75:125], image[50:100, 275:325], image[50:100, 475:525]]`



Comparison 3: Bin Size (Non-parametric)

As the number of bins increases, the segmentation algorithm becomes more “selective” i.e., the bin widths become smaller, so pixel membership becomes increasingly harder. This is why the segmented image appears more granulated.

However, if the bin size is too low, it might be easier for noise or unwanted pixels to “become members” of the ROI. Granted, such choice may be better if the ROI is a solid, uniform color.

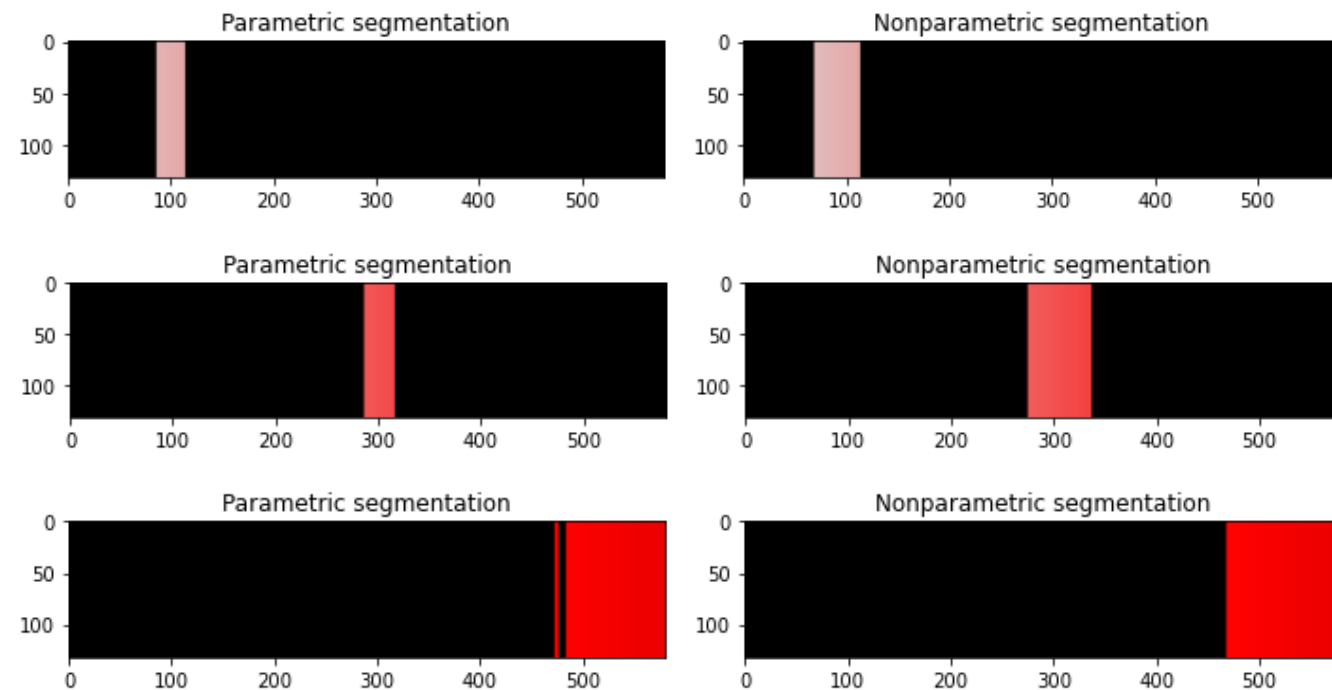


Comparison 4: Saturation Range

```
ROI_images=[image[50:100, 75:125], image[50:100, 275:325], image[50:100, 475:525]]
```

Strips of sizes 50x50 were selected across the range of red saturations. Bin counts is 32.

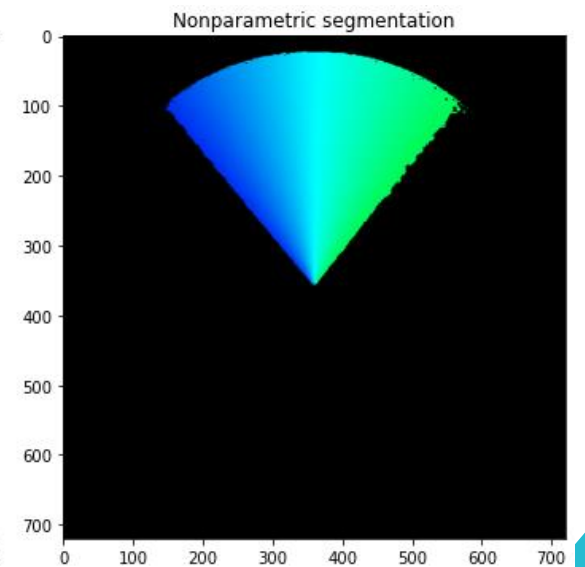
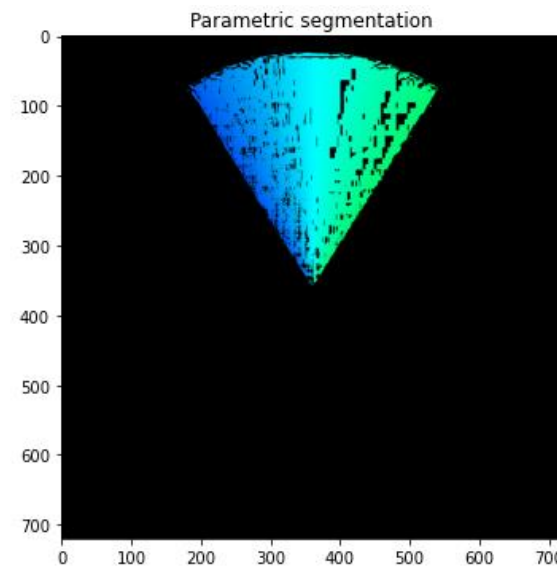
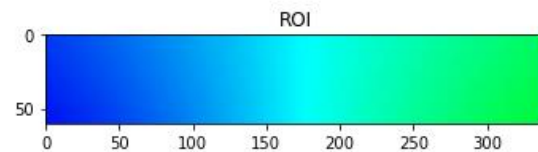
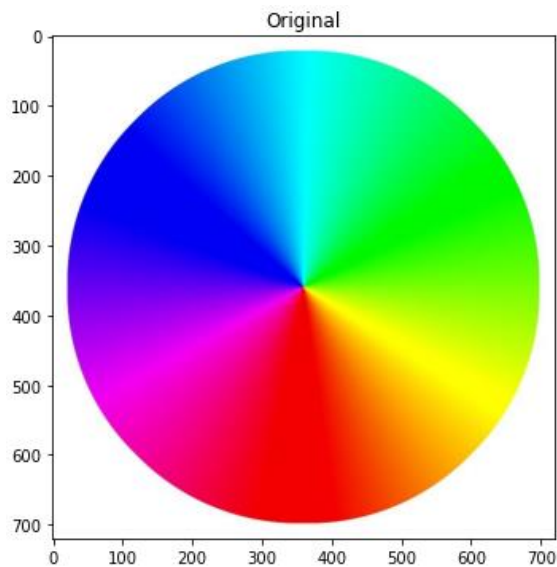
For this particular image, non-parametric segmentation is slightly less selective than parametric segmentation.



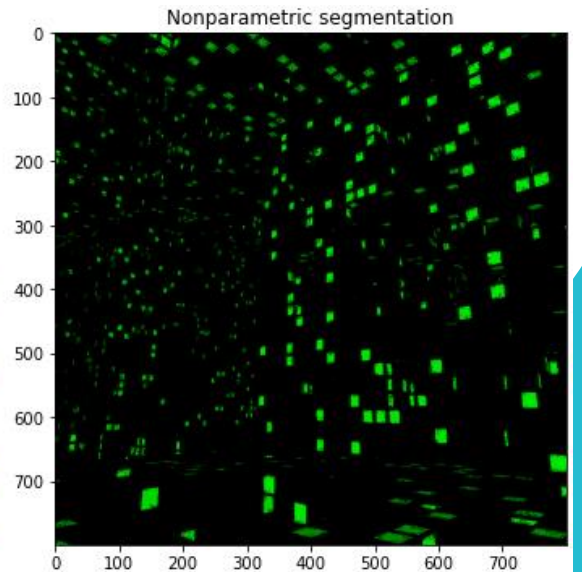
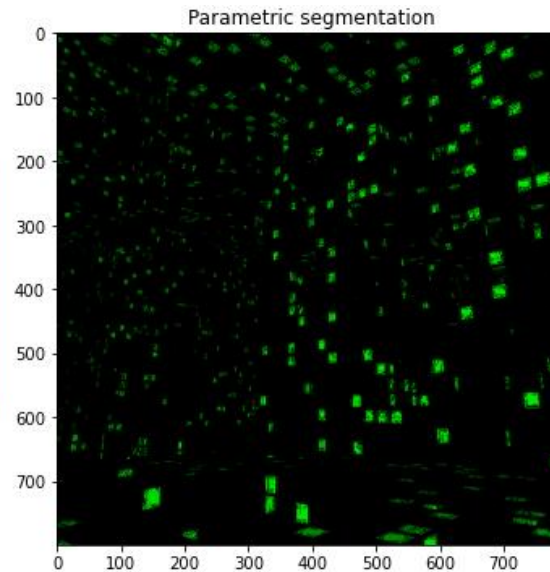
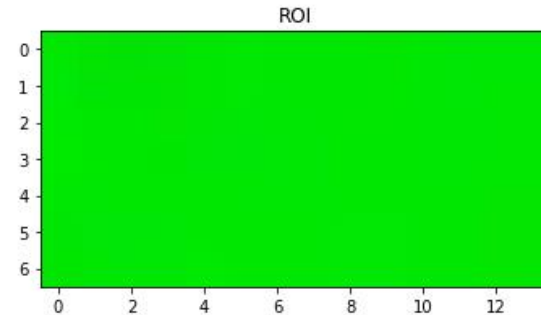
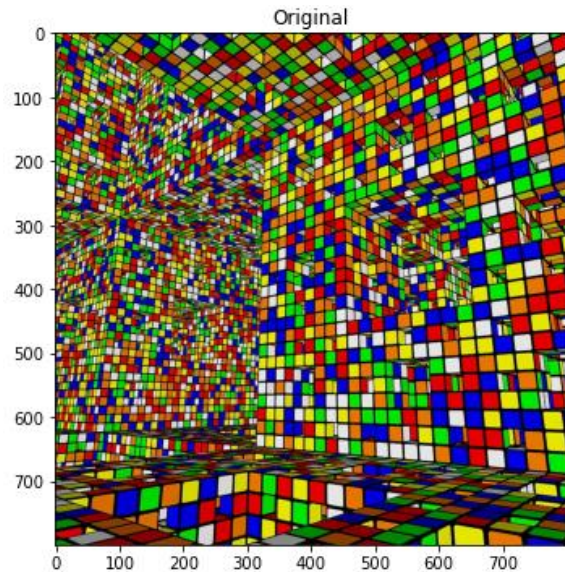
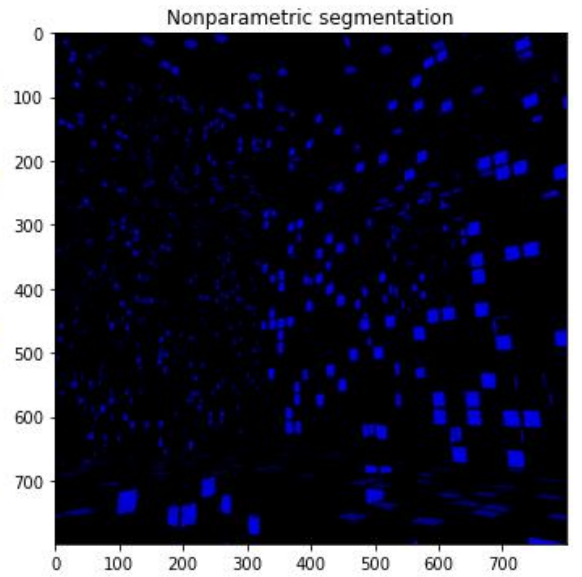
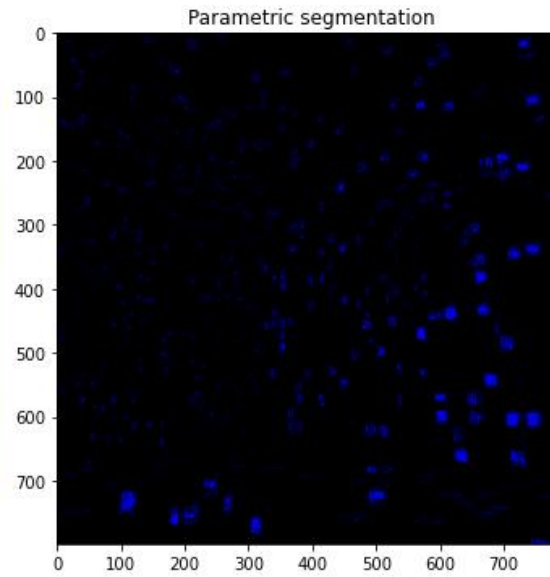
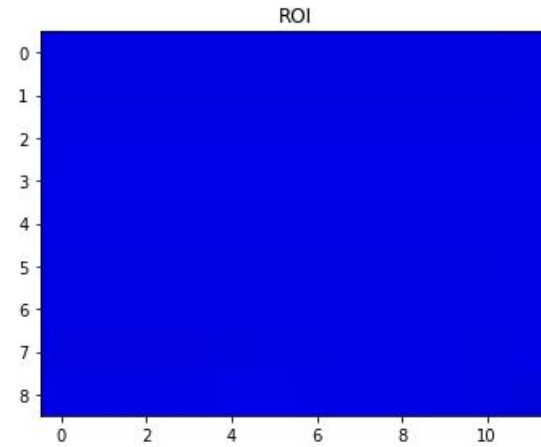
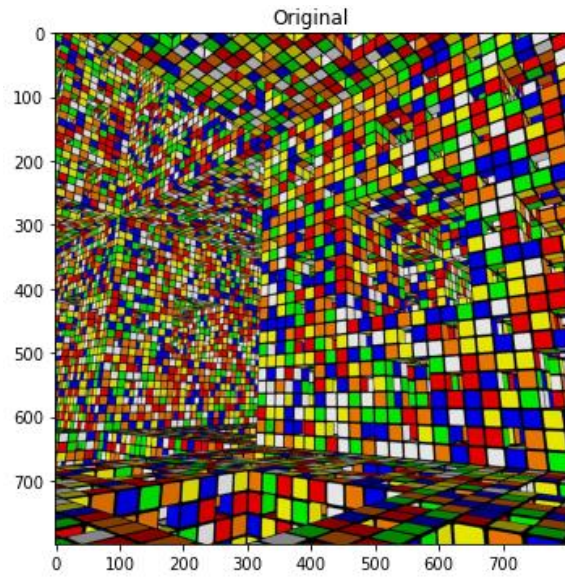
Comparison 5: Color Gradient

For color gradients, the non-parametric algorithm fares much better than the parametric algorithm.

Possible reasons: A color gradient means a higher standard deviation or spread in the color distribution, which means the uncertainty in the determination of pixel membership is higher.

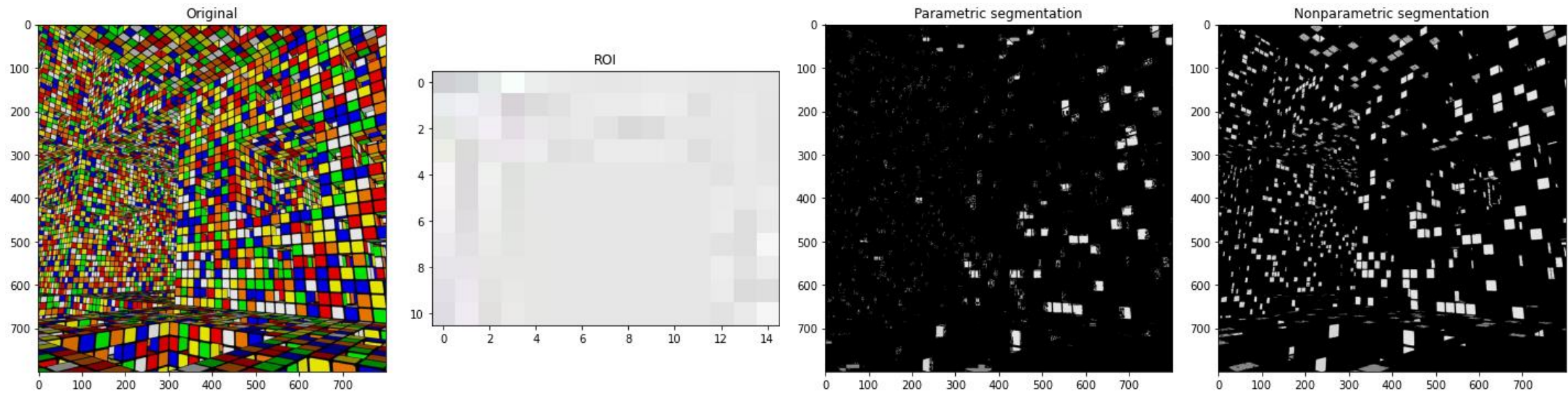


Comparison 6: Resolving Power

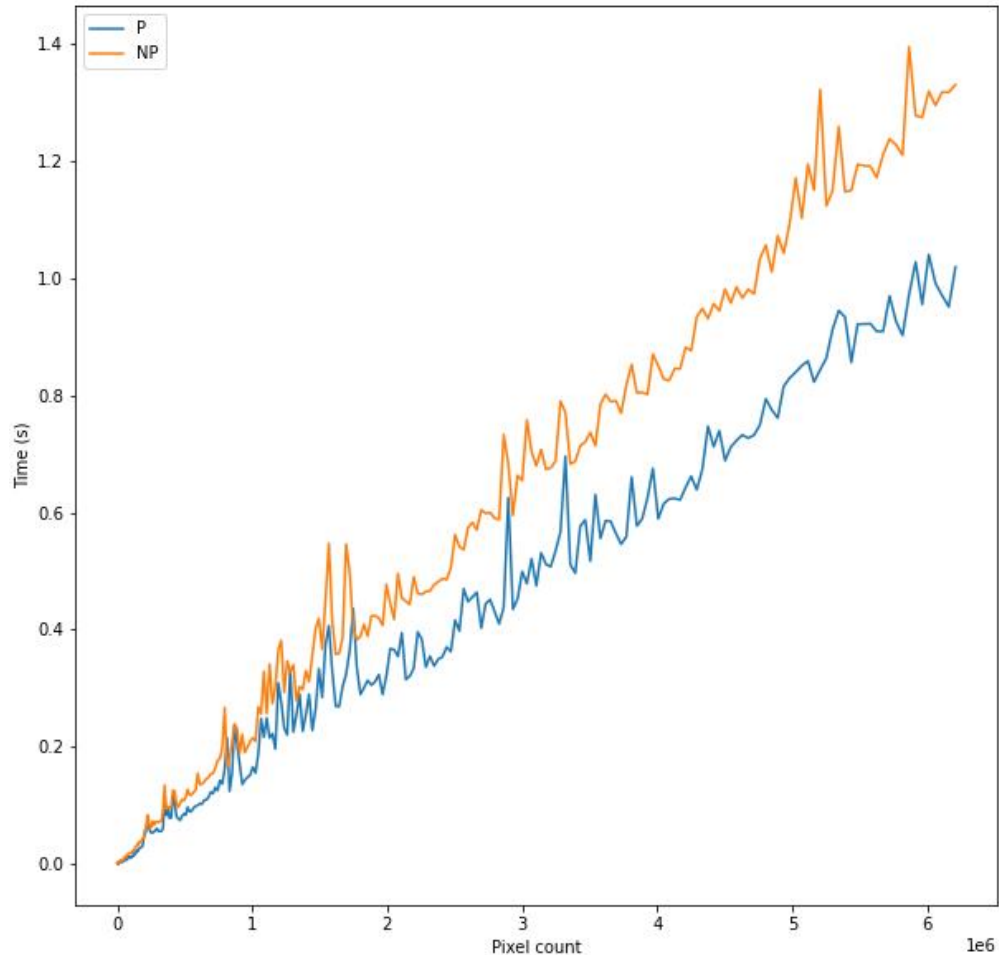


Comparison 6: Resolving Power

Parametric segmentation is less susceptible to noise, but this comes at the cost of potentially segmenting small, colored regions. They also work better at larger ROI sizes. Non-parametric segmentation is often better when segmenting colored images especially when the ROI is not that large.



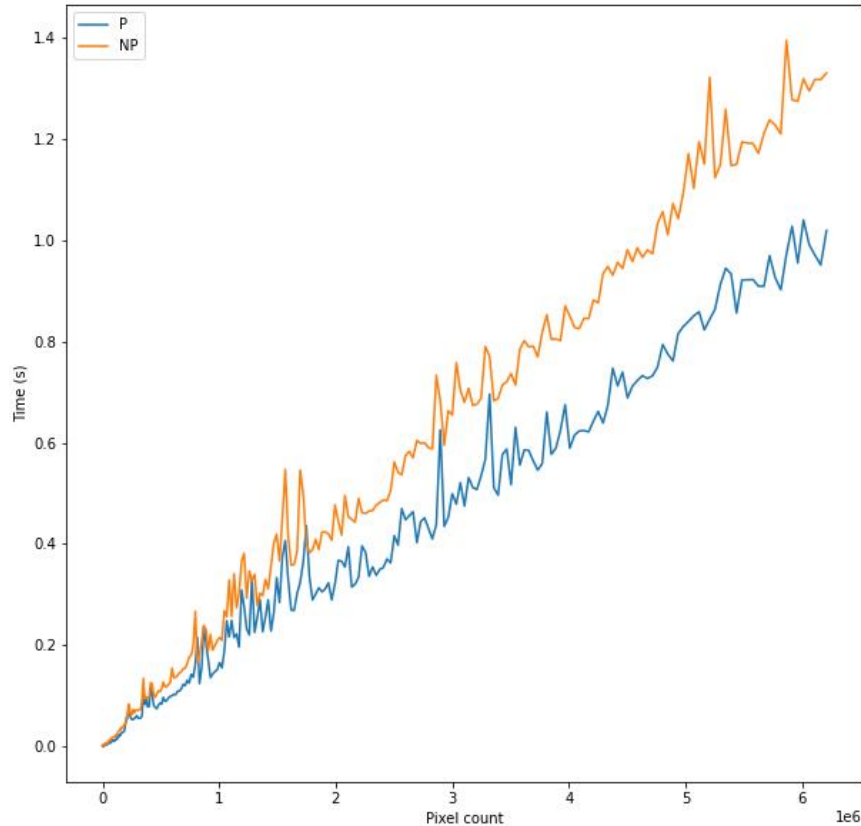
Comparison 7: Runtime/Time Complexity



Note: Both implementations are vectorized.

Here, we can see that both algorithms have linear time complexity. However, the parametric algorithm is generally faster than the non-parametric algorithm.

Comparison 7: Runtime/Time Complexity



```
#Test for a square, randomly-generated RGB image array.
sizes = np.array([10*i+2 for i in range(250)])
times_parametric= []
times_nonparametric = []
for N in sizes:
    image = np.random.random_integers(0,255, size=(N,N,3))
    ROI_image = image[1:51, 1:51]
    start = time.time()
    gaussian_segementer(image, ROI_image)
    end1 = time.time()
    nonparametric_segementer(image, ROI_image)
    end2 = time.time()
    times_parametric.append(end1-start)
    times_nonparametric.append(end2-end1)
```

```
plt.figure(figsize=(10,10))
plt.plot(sizes**2, times_parametric, label='P')
plt.plot(sizes**2, times_nonparametric, label='NP')
plt.legend(loc='best')
plt.xlabel('Pixel count')
plt.ylabel('Time (s)')
```

Summary of Comparisons

Parametric segmentation is faster but requires a higher pixel count for the ROI to be robust and less granular because of its statistical nature. It is less susceptible to noise, especially at higher ROI pixel counts, but this makes them a bit more restrictive.

Non-parametric segmentation is a little bit slower but works quite well even for small ROI, especially solid/uniform colors. Despite its name, it actually requires the bin size as an input. Increasing the bin size makes them more selective at the risk of becoming more granular.

Self-Reflection

I took my time writing code and experimenting with the strengths and limitations of both algorithms using the color wheel and other pictures. I think I was able to explain and demonstrate their pros and cons quite well.

I made a vectorized implementation of both algorithms and avoided the use of for loops which bottleneck the speed of the code. I also compared the non-vectorized implementation in the module and a more efficient vectorized implementation and argued that the latter is roughly two orders of magnitude faster.

I compared the time complexities of both algorithms. This demands that both implementations are vectorized, otherwise, the time plot will take a **very** long time to create.

Self-score: 110/100

Sources

https://scipy-lectures.org/packages/scikit-image/auto_examples/plot_threshold.html (Otsu thresholding for image binarization)

Billiard balls, cheque, and Macbeth Color checker from AP157 Module, Automated Feature Extraction Part I

Color wheel: <https://pixabay.com/illustrations/colour-wheel-spectrum-rainbow-1740381/>

M&Ms candies: <https://www.sfgate.com/politics/article/wokeness-m-and-ms-defeated-17736299.php>

Rubiks' cube room: <https://www.pinterest.ph/pin/260716265899920795/>