CE6003 Etivity 1

- Name : Martin Power
- ID : 9939245

"Region Growing" and "Superpixels"

Recap

This is the Lab on Region Growing and Superpixels for Classical Image Segmention in CE6003. You should complete the tasks in this lab as part of the Region Growing section of the lesson.

Please remember this lab must be completed before taking the quiz at the end of this lesson.

First, if we haven't already done so, we need to clone the various images and resources needed to run these labs into our workspace.

In [1]:

```
#!git clone https://github.com/EmdaloTechnologies/CE6003.git
```

Introduction

In this lab you will complete your third image segmentation project where you will use the watershed algorithm to segment a relatively complex image containing objects which are touching each other.

You will also complete a short super-pixel based segmentation example.

Please work through these projects using the image processing techniques from the previous labs and then segment the example images, once using watershed and once using super-pixels.

At the end of the lab we'll review the work we've done and assess what types of images and projects these approaches are effective for.

Goal

In this lab, you will:

- learn about one introductary 'region growing' image segmentation techique - 'watershed';
- revise deblurring, thresholding, dilation, opening, and the distance transformation;
- use cv2.watershed() to segment an image;
- gain an insight into superpixels. This is a very important topic in neural network based image processing!
- segment the tomatoes image into superpixels
- use the SLIC algorithm

Background

Image segmentation is the process of partitioning a digital image into multiple segments to make the image easier to analyze. Often we are looking to locate objects and boundaries in the original image. Another way of looking at it is image segmentation's goal is to assign a label to every pixel in an image such that pixels with the same label share certain characteristics. Like many elements of computer vision, I find an example is often more useful than precise text.

For example, these images show a typical road scene on the left and a segmented version of the image on the right.

Region Growing - 'Watershed'

The simplest image segmentation techniques, like the thresholding we used in Lab 1, start to show their limitations pretty quickly - requirements like segmenting beyond two objects, touching objects or overlapping objects, etc quickly lead us to techniques such as Watershed,

'Watershed' is a (mostly) unsuperised algorithm used to isolate portions of an image from each other - in effect image segmentation. Its worth noting, however, that - similar to most classical image processing 'watershed' works by identifying elements of the image such as color intensity. i.e. 'watershed' has no semantic understanding of the contents of the image.

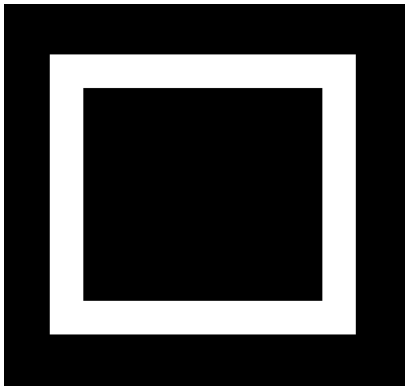
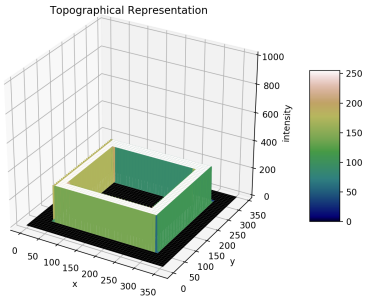
The watershed algorithm is useful for segmentation and is relatively strong at detecting touching or overlapping objects in images such as the image of tomatoes below.

So to summarise, algorithms like 'watershed' have applications where simpler methods such as thresholding and contour detection fail.

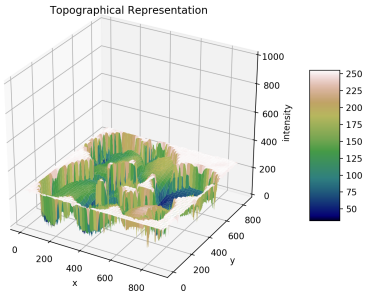
Theory behind Watershed

After converting an image to greyscale, we can imagine that greyscale image as being a topographic surface (a height map) where high intensity denotes 'hills' and low intensity denotes 'valleys'. Conceptually, we start by assigning every isolated 'valley' with different colour liquid. These correspond to our labels. As the liquids rise, the various liquids from different valleys start to merge. We want to avoid these mergers. To avoid mergers, we build barriers at the merger locations. We continue to fill the liquids until all the hills are under liquid.

We can visually grasp the concept using the simple square image on the left and its topographic representation on the right. We can imagine filling the topographic representation of our square with two liquids, one inside the square and one outside the square. As each liquid meets each other we create a 'watershed', and we can imagine this representing the square.

Original	Topography
	

Similarly, we can imagine the tomatoes below on the left, being transformed into a 3d shape being filled with a liquid until the individual tomatoes emerge.

Original	Topography
	

Watershed in Practice

The watershed algorithm tends to give us over-segmented results due to noise etc in the image. So its typical to simplify the image before sending it to the Watershed algorithm, and as you'll see below, there's almost invariably a denoising step.

OpenCV Watershed

A big improvement of the watershed transformation consists of flooding the topographic surface from a previously defined set of markers. This effectively automates the process of starting the watershed algorithm off by giving it hints where to start filling from.

The OpenCV watershed algorithm has two parameters, the image being worked on and a second image. This second image contains 'markers'. These markers must be 'user-defined'.

We won't manually define them (e.g. using point-and-click; instead we'll heuristically define them instead using thresholding and/or morphological operations.

Then we'll apply the watershed algorithm to our image using that marker.

Our Technique

In this technique, we will see how to use the Distance Transformation along with the watershed algorithm to segment some objects.

Let"s remind ourselves of our image of tomatoes. As you can see, this is an image where several of the objects are touching or overlapping. We could tune the thresholding algorithm we used in Lab 1 or the clustering algorithm we used in Lab 2 to detect multiple objects and segment the image into those objects but its likely that they will be unable to distinguish between the two red tomatoes on the left hand side of the image as we look at it.

Instead, we'll start by preparing the image such that the OpenCV 'waterfall' can operate on this image.

Our overall goal here is to construct a 'marker' image for OpenCV's 'waterfall' routine. The marker area is, effectively, an initial area of the image which guides the 'waterfall' algorithm to help with over-segmenting.

Initially we will create an object mask, a background mask and then subtract one from the other to create a mask of the 'unknown' part of the image.

So, in this example, we will 'semi-automate' the process of providing marker to watershed algo

1. decide what is clearly background
2. decide what is clearly foreground
3. create a region that's neither clearly background nor foreground
4. set up 'marker' around this area
5. hand that off to watermark

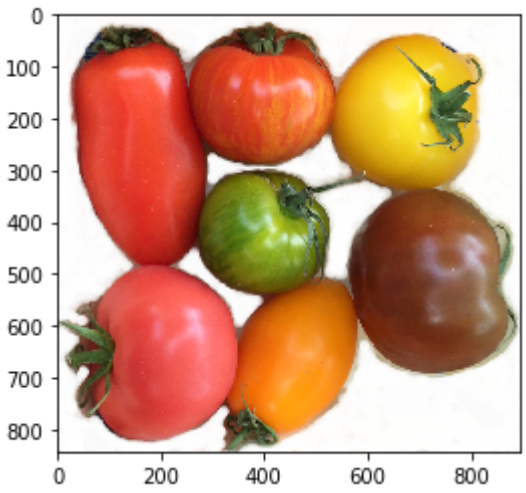
We start by converting the tomatoes image to greyscale, applying a little Gaussian blur and then using a relatively high threshold to to simplify the image.

In [2]:

```
# Similar to previous labs, we're going to use OpenCV, NumPY and MatPlotLib
# so import them here
import cv2
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# Load an image
#img = cv2.imread("/content/CE6003/images/lab2/tomatoes.png")
img = cv2.imread("./images/lab2/tomatoes.png")
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
```

Out[2]:

```
<matplotlib.image.AxesImage at 0x1a4a3a19c88>
```

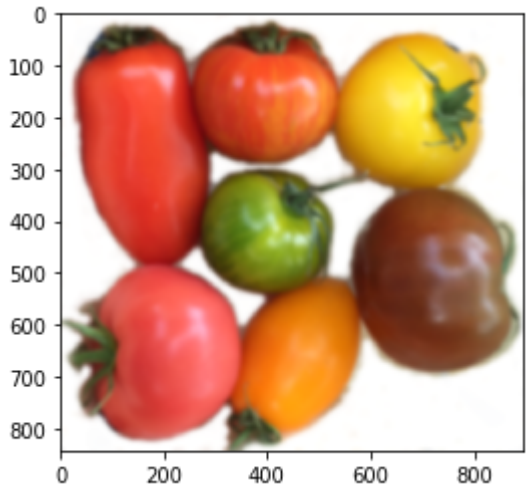


In [3]:

```
# YOUR CODE GOES HERE
# The degree of blurring is strongly coupled with the performance of the watershed.
# It can be quite interesting to experiment with adjusting the amount of blurring.
# Exercise: Try using cv2.GaussianBlur() until your image is satisfactory.
# You have a reference image below to help you if you want it.
# Store the output from cv2.GaussianBlur() in a variable called blur

# cv2.GaussianBlur(src, ksize, sigmaX[, dst[, sigmaY[, borderType]]]) → dst
kernel_dim = 25
blur = cv2.GaussianBlur(img,(kernel_dim,kernel_dim),sigmaX=0,sigmaY=0)
plt.imshow(cv2.cvtColor(blur, cv2.COLOR_BGR2RGB))
# END YOUR CODE HERE
```

Out[3]:

```
<matplotlib.image.AxesImage at 0x1a4a3491f08>
```

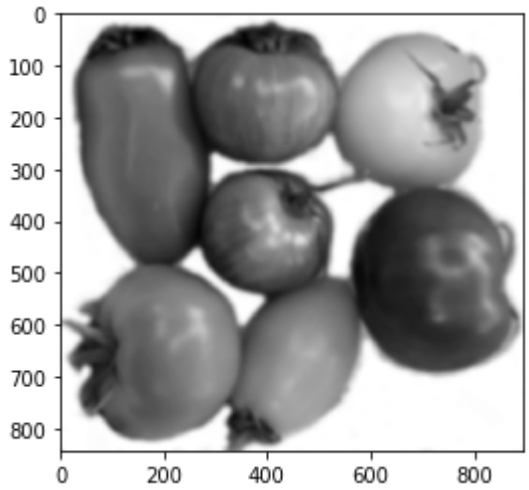


In [4]:

```
# Convert to grayscale so we can threshold it
gray = cv2.cvtColor(blur, cv2.COLOR_BGR2GRAY)

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

Out[4]:

```
<matplotlib.image.AxesImage at 0x1a4a350b408>
```

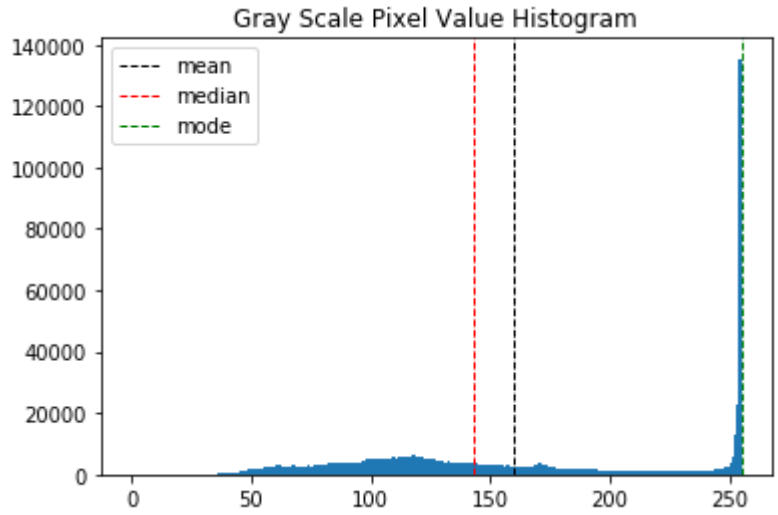


So, after these operations, you should be starting to see the tomatoes emerge from the background.

Your blurred grayscale image should be similar to the image here.

In [5]:

```
# Histogram of Gray Scale Pixel Values
from scipy import stats
plt.hist(gray.ravel(), bins=256, range=(0.0, 255.0))
plt.axvline(gray.ravel().mean(), color='k', linestyle='dashed', linewidth=1, label='mea
n')
plt.axvline(np.median(gray.ravel()), color='r', linestyle='dashed', linewidth=1, label=
'median')
plt.axvline(int(stats.mode(gray.ravel())[0]), color='g', linestyle='dashed', linewidth=
1, label='mode')
plt.legend()
plt.title('Gray Scale Pixel Value Histogram');
plt.show()
```



In [6]:

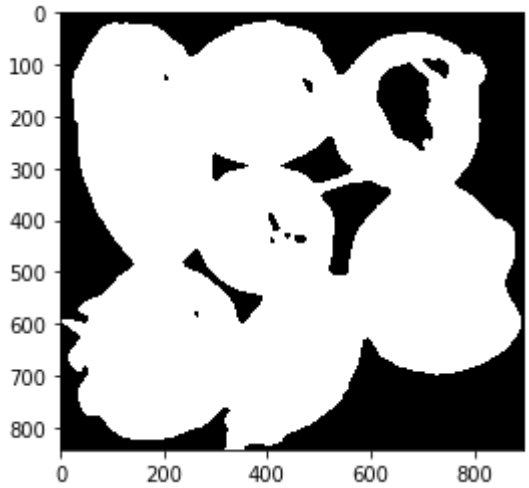
```
# YOUR CODE GOES HERE
# Experiment: So, create a matrix thresh which is thresholded by OpenCV's threshold() o
peration.
# HINT: I did not using OTSU here as I got better results from
# weighting the threshold high. You can experiment here with thresholding to
# improve on the final segmentation.
#
# Use the method cv2.threshold() which returns two values. The second value is the
# value you are interested in, you can ignore the first for now...
#
# Store the value in a variable called thres
ret, thresh = cv2.threshold(gray,200,255,cv2.THRESH_BINARY_INV)
#ret, thresh = cv2.threshold(gray,250,255,cv2.THRESH_BINARY_INV)
# END YOUR CODE GOES HERE
```

In [7]:

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

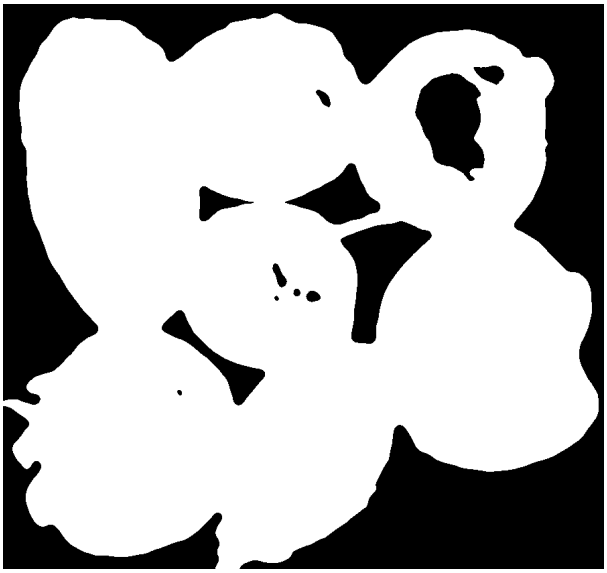
Out[7]:

```
<matplotlib.image.AxesImage at 0x1a4a6041848>
```



Now, you should have a matrix (or image if you prefer) which is similar to this image.

After these operations, we're starting to see the tomatoes emerge from the background. Note in this example that, its a little more complex than the starfish example, in that the objects of interest are touching and/or overlapping.



Prepare the marker image

Now you need to prepare the foreground, background, and unknown regions of the image to create a marker for OpenCV's watershed().

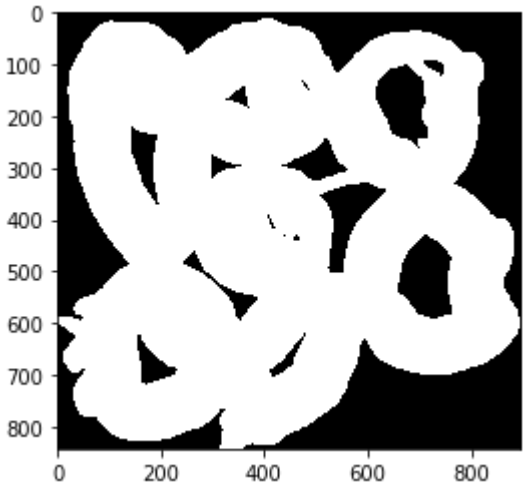
At this stage, we can start preparing the foreground, background, and 'unknown' regions of the image. For this image, I used a closing morphological operation followed by a dilation to generate the background image and I used a distance transform to generate the foreground image. Finally, I subtracted those two images from each other to generate the unknown area of the image.

In [8]:

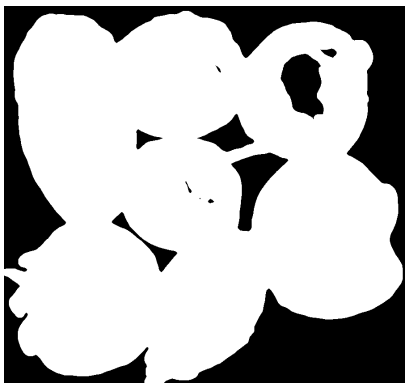
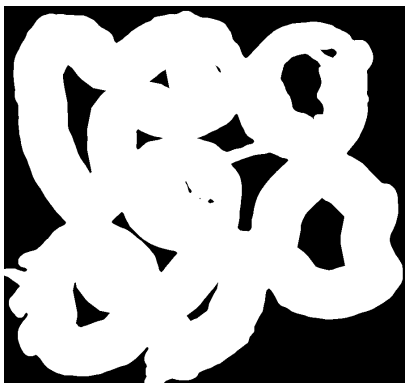
```
#  Close the image a little to fill in a few small holes in it
closingKernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3,3))

# Create a matrix closed that is generated from thresh by a closing
# operation using the kernel above.
closed = cv2.morphologyEx(thresh, cv2.MORPH_CLOSE, closingKernel)

# Use just enough dilate to get some clearly identifiable background
dilationKernel = np.ones((3,3), np.uint8)

# create a matrix 'bg' from OpenCV's dilate() function
# using the dilation kernel above
bg = cv2.dilate(closed, dilationKernel, iterations=3)

# Now use a distance transform to extract is clearly foreground

# Create a matrix 'dist_transform' using OpenCV's distanceTransform
# method on the 'closed' matrix.
dist_transform = cv2.distanceTransform(closed, cv2.DIST_L2, 5)

# Threshold the distance transformation
ret, fg = cv2.threshold(dist_transform,0.7*dist_transform.max(), 255, 0)

# Now find the unknown region by subtracting one from the other
fg = np.uint8(fg)
unknown = cv2.subtract(bg, fg)

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

Out[8]:

```
<matplotlib.image.AxesImage at 0x1a4a60b1088>
```



At this point your unknown region should be substantially similar to the image on the right here. I've also included a reference background image and foreground (or object) image in case you lose your way a little.

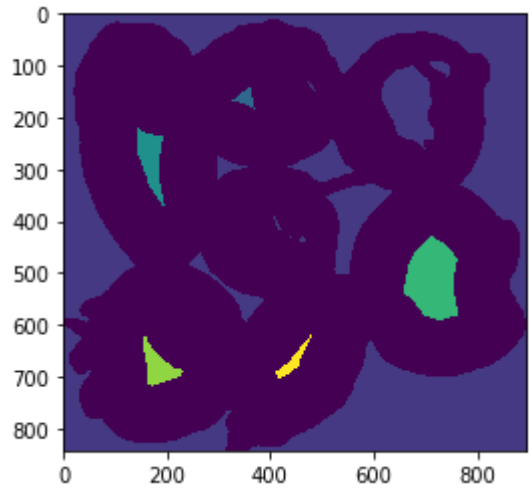
Background	Foreground	Unknown
		

Now, we're ready to do a little data manipulation before calling OpenCV's watershed.

Right now what we have in marker is a set of pixel values, with 0 representing the background and 255 representing the unknown region and what OpenCV's watershed expects to see in marker is the unknown region set to 0. So, you need to run the following code to prepare for watershed.

In [9]:

```
# Marker labelling
ret, marker = cv2.connectedComponents(fg)

# Add one to all labels so that bg is not  0, but 1
marker = marker+1

# Now, mark the region of unknown with 0
marker[unknown==255] = 0;

plt.imshow(marker)
plt.show()
print("Number of Unique Markers:",len(np.unique(marker)),"Markers : ",np.unique(marker
))
```



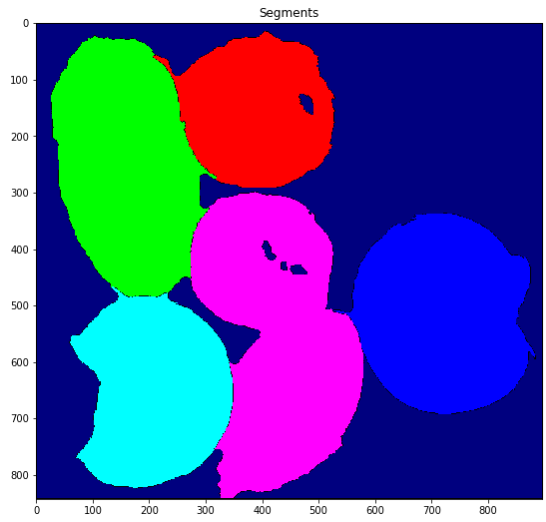

```
Number of Unique Markers: 7 Markers :  [0 1 2 3 4 5 6]
```

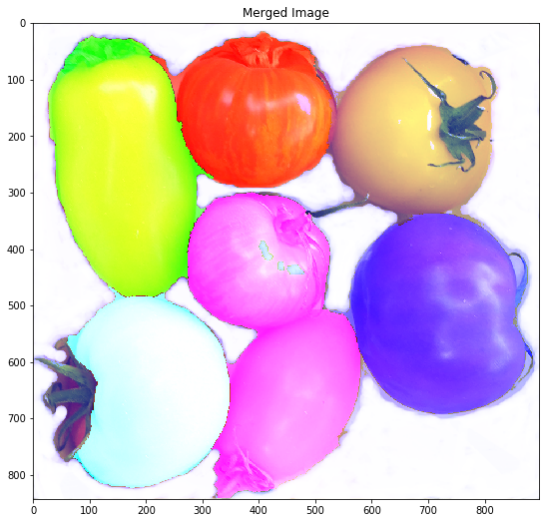
At this stage you should be seeing an image substantively similar to one shown here.

Now we can run watershed over our image. Then we'll colour the segmentation boundaries the watershed found in green and save it.

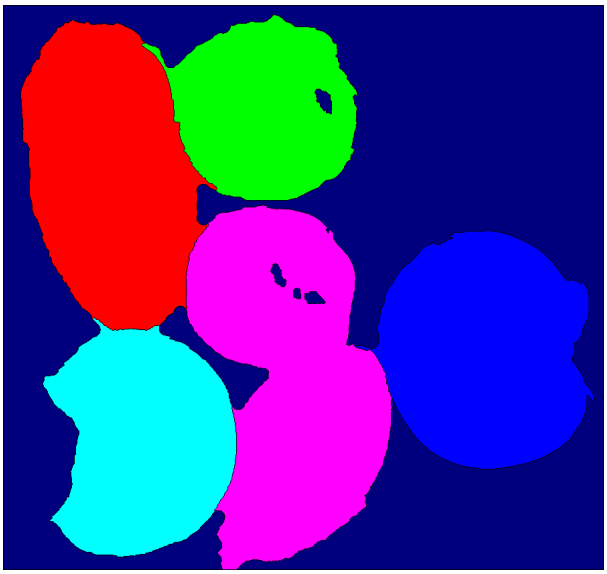
In [10]:

```
# Now marker is ready.  It is time for last step
cv2.watershed(img, marker)

# Create a new empty image with the same shape
# as the original image.
h, w, num_c = img.shape
seg = np.zeros((h, w, num_c), np.uint8)

# Watershed has replaced the pixel
# values in marker with integers representing
# the segments it has found in the original
# image.
# Color in these segments
#
maxMarker = np.max(marker)
minMarker = np.min(marker)

colorMap =  [ \
         [0,0,0], \
         [255,255,255], \
         [127,0,0], \
         [0,0,255], \
         [0,255,0], \
         [255,0,0], \
         [255,255,0], \
         [255,0,255], \
         [0,255,255], \
         [0,127,0], \
         [0,0,127] \
            ]

for region in range(minMarker, maxMarker+1):
    seg[marker==region] = colorMap[region+1]


plt.figure(figsize=(20,10))
plt.subplot(1,2,1)
plt.imshow(cv2.cvtColor(seg, cv2.COLOR_BGR2RGB))
plt.title("Segments")

plt.subplot(1,2,2)
merge = cv2.bitwise_or(seg, img)
plt.imshow(cv2.cvtColor(merge, cv2.COLOR_BGR2RGB))
plt.title("Merged Image")

plt.show()
```





Now your image should be similar to the one I've shown here.



There are a few items I want to highlight at this stage - most notably we've missed the yellow tomato on top t the right and two of our tomatoes have 'merged'.

We can continue to tweak to improve the segmentation for this image, but its probably more productive to just note, for now, that - with classical algorithms - we can often end up hand-crafting our features.

SLIC

We'll be talking about super-pixels at a later stage in this module. We're going to do a short lab with super-pixels here to introduce them.

You'll find a more detailed discussion of super-pixels and the SLIC algorithm in the accompanying lesson for this lab.

In summary, an isolated pixel does conveys extremely little information about the image it resided within. There would appear to be good benefits from gathering pixels together to gain some semantic information - i.e. to create a 'thing' which is capable of carrying some 'meaning'.

If we call out the work of Dr. Xiaofeng Ren's work here super-pizels would ideally contain perceptual meaningfulness - for instance colour, texture, etc (and ideally semantic value). This property leads towards their use in neural networks.

As you will see, super-pixels also offer computational efficiency - the reduce the complexity of images from hundreds of thoushands of pixels to hundreds of pixels.

Super-pixelation tends to lead to the detection of the boundaries we desire while having a side-effect of over-segmenting and introducing boundaries we don't want. Pleae keep this insight as you work through this lab and retain it as you work through later lessons.

For this algorithm, you'll need to import the 'segmentation' package

In [11]:

```
from skimage import segmentation, color
from skimage.io import imread
from skimage.io import imsave

from skimage.segmentation import mark_boundaries
```

Now, load your image. We"re going to use the relatively complex tomatoes image again. However, this time, please don"t blur or grayscale the image.

In [12]:

```
# Load an image
#img = imread("/content/CE6003/images/lab2/tomatoes.png")
img = imread("./images/lab2/tomatoes.png")
```

Run the SLIC super-pixel algorithm over the image, varying the compactness and the numsegments until the image you produce resembles the image below.

Watershed Function

This function captures the functionality require to run the watershed algorithm. This was encapsulated in a function to allow for easier experimentation and the calling of this function from within a loop

In [13]:

```
def run_watershed(img, add_blur=False, kernel_dim=25, sigma=0, gray_conv_flag=cv2.COLOR
_RGB2GRAY, plot=False):

    if(plot==True):
        plt.imshow(img)
        plt.title("Superpixel Image")
        plt.show()


    if(add_blur==True):
        blur = cv2.GaussianBlur(img,(kernel_dim,kernel_dim),sigmaX=sigma,sigmaY=sigma)
        if(plot==True):
            plt.imshow(blur)
            plt.title("Blurred Image")
            plt.show()
        gray = cv2.cvtColor(blur, cv2.COLOR_RGB2GRAY)
    else:
        gray = cv2.cvtColor(img, gray_conv_flag)

    if(plot==True):
        plt.imshow(gray, cmap='gray')
        plt.title("Gray Scale Image")
        plt.show()

    ret, thresh = cv2.threshold(gray,200,255,cv2.THRESH_BINARY_INV)

    if(plot==True):
        plt.imshow(thresh, cmap='gray')
        plt.title("Threshold Image")
        plt.show()

    #  Close the image a little to fill in a few small holes in it
    closingKernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3,3))

    # Create a matrix closed that is generated from thresh by a closing
    # operation using the kernel above.
    closed = cv2.morphologyEx(thresh, cv2.MORPH_CLOSE, closingKernel)

    # Use just enough dilate to get some clearly identifiable background
    dilationKernel = np.ones((3,3), np.uint8)

    # create a matrix 'bg' from OpenCV's dilate() function
    # using the dilation kernel above
    bg = cv2.dilate(closed, dilationKernel, iterations=3)

    # Now use a distance transform to extract is clearly foreground

    # Create a matrix 'dist_transform' using OpenCV's distanceTransform
    # method on the 'closed' matrix.
    dist_transform = cv2.distanceTransform(closed, cv2.DIST_L2, 5)

    # Threshold the distance transformation
    ret, fg = cv2.threshold(dist_transform,0.7*dist_transform.max(), 255, 0)

    # Now find the unknown region by subtracting one from the other
    fg = np.uint8(fg)
    unknown = cv2.subtract(bg, fg)

    if(plot==True):
```

```
        plt.imshow(unknown, cmap='gray')
        plt.title("Unknown Region Image")
        plt.show()

    # Marker labelling
    ret, marker = cv2.connectedComponents(fg)

    # Add one to all labels so that bg is not  0, but 1
    marker = marker+1

    # Now, mark the region of unknown with 0
    marker[unknown==255] = 0;

    if(plot==True):
        print("Number of Unique Markers:",len(np.unique(marker)),"Markers : ",np.unique
(marker))
        plt.imshow(marker)
        plt.title("Markers Image")
        plt.show()

    # Now marker is ready.  It is time for last step
    #cv2.watershed(img, marker)
    cv2.watershed(cv2.cvtColor(img,cv2.COLOR_RGB2BGR), marker)

    # Create a new empty image with the same shape
    # as the original image.
    h, w, num_c = img.shape
    seg = np.zeros((h, w, num_c), np.uint8)

    # Watershed has replaced the pixel
    # values in marker with integers representing
    # the segments it has found in the original
    # image.
    # Color in these segments
    #
    maxMarker = np.max(marker)
    minMarker = np.min(marker)

    colorMap =  [ \
             [0,0,0], \
             [255,255,255], \
             [127,0,0], \
             [0,0,255], \
             [0,255,0], \
             [255,0,0], \
             [255,255,0], \
             [255,0,255], \
             [0,255,255], \
             [0,127,0], \
             [0,0,127] \
            ]

    for region in range(minMarker, maxMarker+1):
        seg[marker==region] = colorMap[region+1]

    if(plot==True):
        plt.imshow(cv2.cvtColor(seg, cv2.COLOR_BGR2RGB))
        plt.title("Segmented Image")
        plt.show()

    return seg
```

Print SLIC Plots

Function to print the following plots

- Superpixel Boundaries
- Superpixel image
- Segments from Watershed Algorithm
- Merged Image from Original with Segments

In [14]:

```
def print_slic_plot():
    plt.figure(figsize=(20,10))

    plt.subplot(1,4,1)
    plt.imshow(mark_boundaries(superpixels, img_segments))
    plt.title("Boundaries")

    plt.subplot(1,4,2)
    plt.imshow(superpixels)
    plt.title("Superpixels")

    plt.subplot(1,4,3)
    plt.imshow(cv2.cvtColor(seg, cv2.COLOR_BGR2RGB))
    plt.title("Segments")

    plt.subplot(1,4,4)
    merge = cv2.bitwise_or(seg, img)
    plt.imshow(cv2.cvtColor(merge, cv2.COLOR_BGR2RGB))
    plt.title("Merged Image")

    plt.show()
```

Setting numSegments and compactFactor

The following cells contains list of values that were used to run loops to experiment with optimal settings for the number of segments and the compact factor

Segments

The documentation stated that the number of segments should approximately correspond to the number of labelled objects. Therefore, I expected a value of approximately 7 would be sufficient. However, very poor results were obtained. In the end I need to use much larger values and settled upon a value of 800

Compactness

The notebook had compactness set at 20. The documentation recommended using log values (0.01, 0.1, 1, 10, etc) initially to tune the algorithm. I tried this and the best values where in the 10s range. From there I tried values of 20,30,40,50,etc. In the end, using a compactness value of 20 was delivering the best results

In [15]:

```
# YOUR CODE HERE
# Experiment: Vary numSegments to find a reasonable value for it for this image
# Important Note: You are manually selecting features and feature properties
# (e.g. size of super-pixels) here

#numSegments = [1,2,7,10,100,300,500]
#numSegments = [1000]
#numSegments = [500,600,700,750,800,900,950,1000]
#numSegments = [700,750,800,850,900,950,1000]
#numSegments = [800,850,875,900]
numSegments = [800]

# Exercise: set the variable numSegments to a suitable value
# END YOUR CODE HERE
```

In [16]:

```
#compactFactor = [0.01, 0.1, 1, 10, 100]
#compactFactor = [40]
#compactFactor = [10,20,30,40]
#compactFactor = [20,30,40,50]
#compactFactor = [30,35,40,45,50]
#compactFactor = [20,25,30,35,40,45,50]
#compactFactor = [20,30,40]
compactFactor = [20]
```

Running SLIC

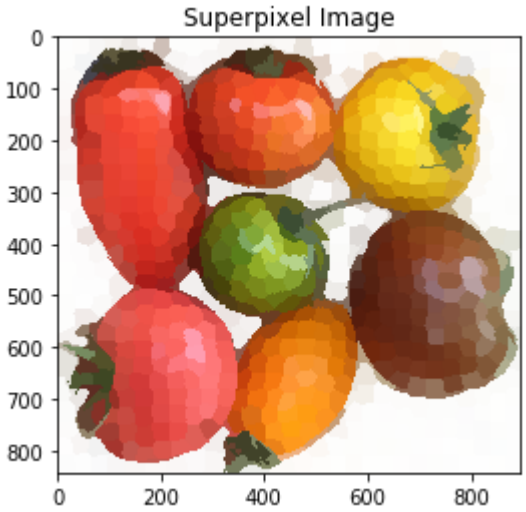
After running experiments were I looped through various combinations of n_segments and compactness using the lists above, I settled on the following as the parameters that best approximated the reference image.

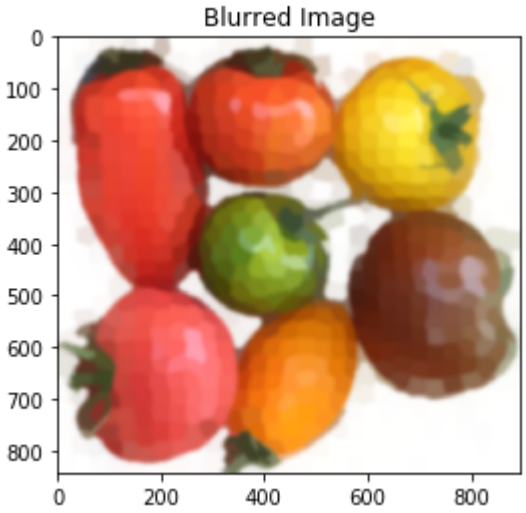
- n_segments = 800
- compactness = 20

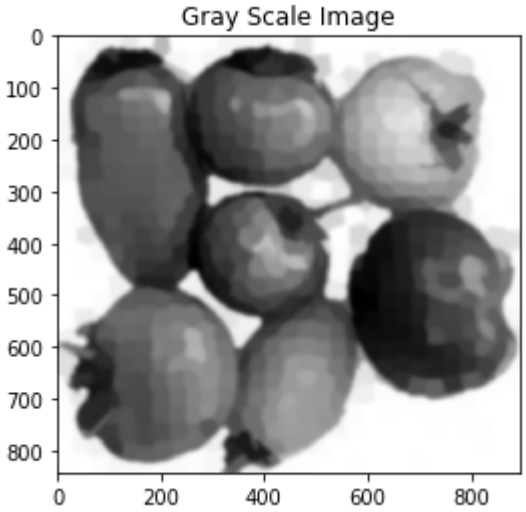
In [17]:

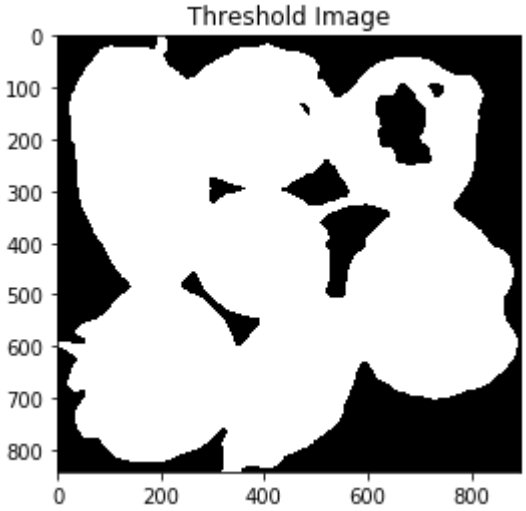
```
for i in numSegments:
    for j in compactFactor:
        print("n_segments:",i,"compactness:",j)
        img = imread("./images/lab2/tomatoes.png")
        img_segments = segmentation.slic(img, compactness=j, n_segments=i)
        superpixels = color.label2rgb(img_segments, img, kind='avg')

        seg = run_watershed(superpixels, add_blur=True, kernel_dim=25, plot=True)
        print_slic_plot()
```

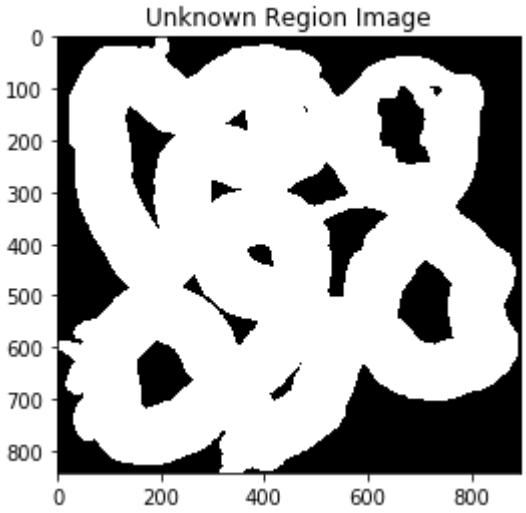
```
n_segments: 800 compactness: 20
```



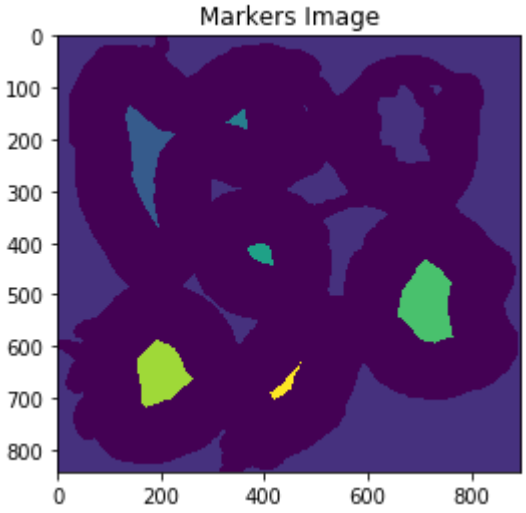


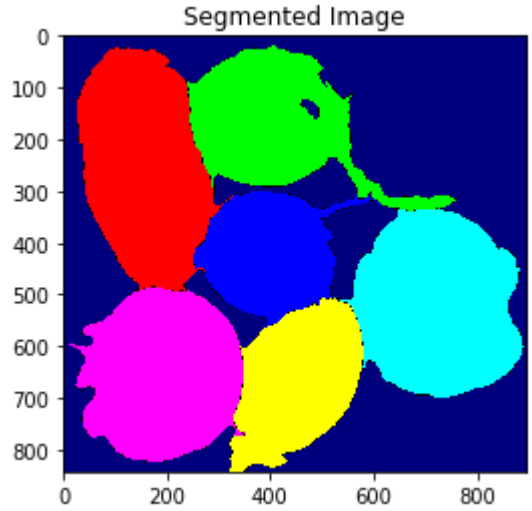


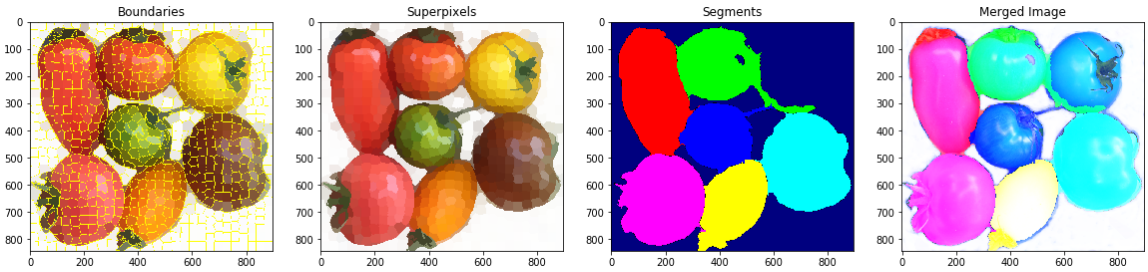





```
Number of Unique Markers: 8 Markers :  [0 1 2 3 4 5 6 7]
```



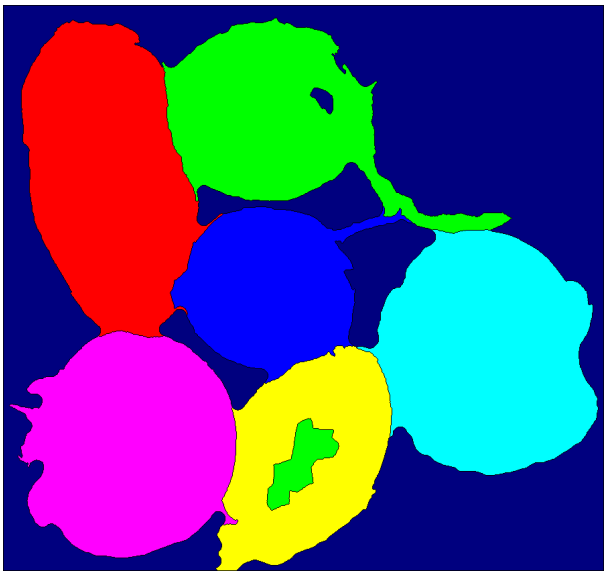




Now your image should be similar to the one shown here.

To complete the assignment, you should run the watershed algorithm over the image you created in the last step.

Your aim is to produce an image similar to the one below.



Conclusion

So, that completes the third of the four labs in this lesson.

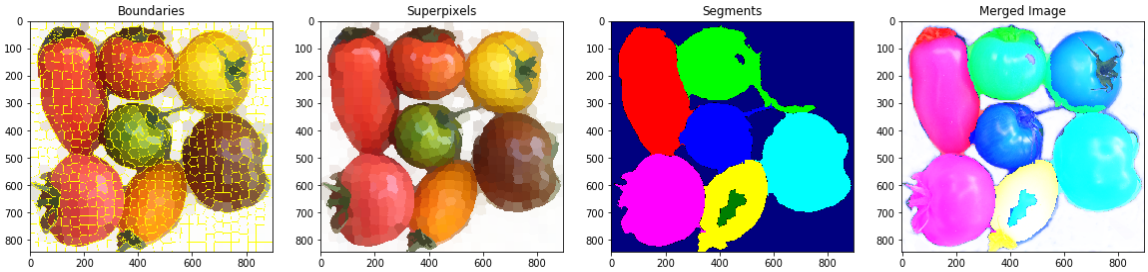
You've learned how to use erosion and dilation You've learned how to use the watershed algorithm to segment an image with touching or overlapping regions. You've been introduced to super-pixels and the SLIC algorithm You've gained an insight that classical techniques involve manually tuning features like super-pixel size for individual images or classes of images and you should be asking yourself "surely, there's a better way?"

Trying to Match Segment Exactly

Based on content on the forum and GitLab from Rick Qui and Victor Amaya, the following settings appear to get almost the exact image as the reference image (with the artifact on the orange tomato)

In [18]:

```
numSegments = 790
compactFactor = 20
kernel_dim = 15
sigma=17

img = imread("./images/lab2/tomatoes.png")
img_segments = segmentation.slic(img, compactness=compactFactor, n_segments=numSegments
)
superpixels = color.label2rgb(img_segments, img, kind='avg')

seg = run_watershed(superpixels, add_blur=True, kernel_dim=kernel_dim, sigma=sigma, plo
t=False)
print_slic_plot()
```



Trying to Identify Yellow Tomato in Top Left Corner

I ran multiple experiments trying to get the SLIC and watershed algoritms to correctly identify yellow tomato in the top left corner.

Initially I tried varying the compactness and the number of segments and iterating through combinations in a loop. However, this did not yield the results I was looking for.

I tried varying other parts of the code and found that with blurring disabled, results seemed to be better.

Finally, after some more edits I got a result that segmented all seven tomatoes. However, on further inspection there was an unitended typo in the code that was using cv2.COLOR_BGR2GRAY instead of cv2.COLOR_RGB2GRAY when converting the superpixel image to grayscale. When I corrected this typo, the segment for the yellow tomato disappeared and two of the tomatoes were appearing as one segment.

The code below shoes a comparison between using cv2.COLOR_RGB2GRAY and cv2.COLOR_BGR2GRAY when covnerting the image to grayscale. Even though the image is RBG, using cv2.COLOR_BGR2GRAY produces an image where the tomatoes are darker in colour and this additional darkness/definition appears to help the watershed algorithm to segment the superpixel image. This is some accidental hand crafting.

Why the difference?

Why is there a difference and how did I end up introducing this change in behaviour? At the beginning of the notebook, the image is loaded using the OpenCV imread() function:

```
img = cv2.imread("{PATH}tomatoes.png")
```

The documentation for this function states that *In the case of color images, the decoded images will have the channels stored in **B G R** order.* which means that `img` in this case is a BGR image in memory.

However, in the SLIC part of the notebook, the image is loaded using the skimage.io imread() function:

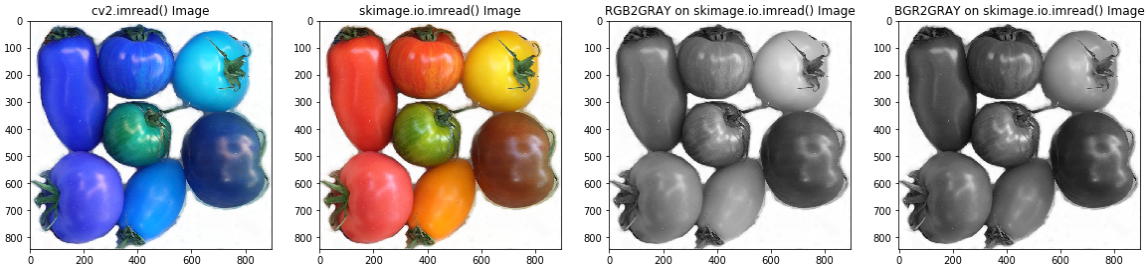
```
img = imread("{PATH}tomatoes.png")
```

The skimage.io.imread() function behaves differently to the cv2.imread() function. The documentation for the skimage.io.imread() function states that for the return image *The different color bands/channels are stored in the third dimension, such that a gray-image is MxN, an **RGB-image MxNx3** and an RGBA-image MxNx4*. As a result, when loaded using this function, `img` is an RGB image in memory.

Therefore care needs to be taken with the function used to load the image and the manipulation of the image subsequently.

In [19]:

```
cv2_img = cv2.imread("./images/lab2/tomatoes.png")
img = imread("./images/lab2/tomatoes.png")
plt.figure(figsize=(20,10))

plt.subplot(1,4,1)
plt.imshow(cv2_img)
plt.title("cv2.imread() Image")

plt.subplot(1,4,2)
plt.imshow(img)
plt.title("skimage.io.imread() Image")

plt.subplot(1,4,3)
plt.imshow(cv2.cvtColor(img, cv2.COLOR_RGB2GRAY),cmap='gray')
plt.title("RGB2GRAY on skimage.io.imread() Image")

plt.subplot(1,4,4)
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2GRAY),cmap='gray')
plt.title("BGR2GRAY on skimage.io.imread() Image")

plt.show()
```



The following code was used to find the ideal number of segment and compactness

The settings used were

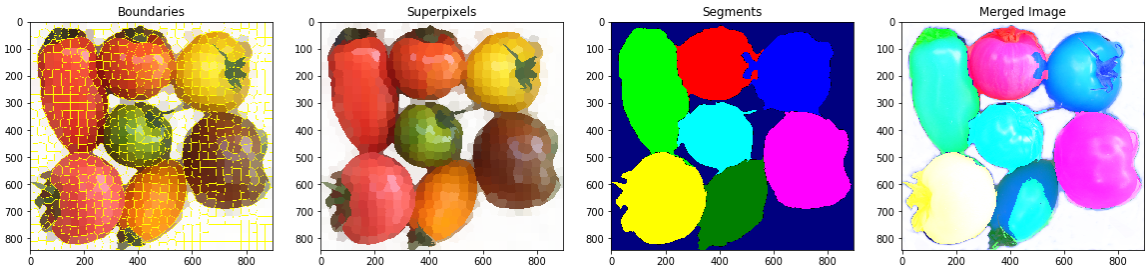
- n_segments=850
- compactness = 30
- Gray Scale Conversion Flag : cv2.COLOR_BGR2GRAY
- Blurring Disabled

For comparison, the loop is also exectured with the gray scale coversion flag set to cv2.COLOR.RGB2GRAY and this shows how worse performance is obtained even if all other parameters are kept the same

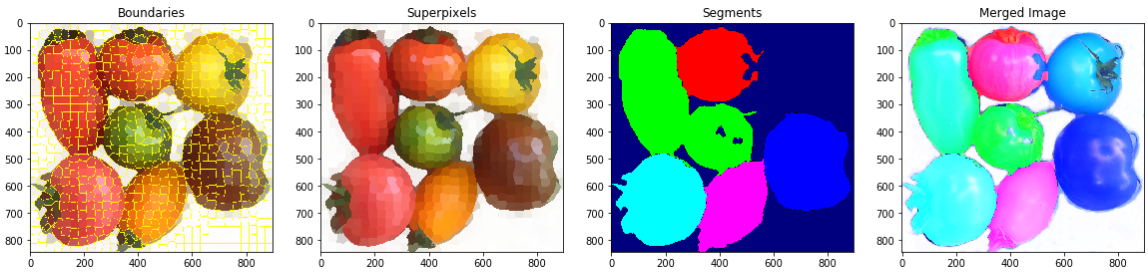
In [20]:

```
#numSegments = [700,725,750,775,800,850,900,950,1000]
#compactFactor = [20,30,40]

# Through experimemtation, numSegments=850 and compactFactor=30
# appears to be the sweetspot where all tomatoes get segmented
numSegments = [850]
compactFactor = [30]

# Even though image is RGB, using the BGR2GRAY color flag helps
# the algorithm to better identify the tomatoes. This was an
# accidental discovery due to code typ
gray_conv_flag=[cv2.COLOR_BGR2GRAY,cv2.COLOR_RGB2GRAY]

# Turn off blur. Better results observerd without blurring
add_blur=False

for i in numSegments:
    for j in compactFactor:
        for k in gray_conv_flag:
            if(k==cv2.COLOR_BGR2GRAY):
                print("n_segments:",i,"compactness:",j, "gray scale conversion flag: cv
2.COLOR_BGR2GRAY")
            else:
                print("n_segments:",i,"compactness:",j, "gray scale conversion flag: cv
2.COLOR_RGB2GRAY")
            img = imread("./images/lab2/tomatoes.png")
            img_segments = segmentation.slic(img, compactness=j, n_segments=i)
            superpixels = color.label2rgb(img_segments, img, kind='avg')

            seg = run_watershed(superpixels, add_blur=add_blur, gray_conv_flag=k,plot=F
alse)
            print_slic_plot()
```

```
n_segments: 850 compactness: 30 gray scale conversion flag: cv2.COLOR_BGR2
GRAY
```




```
n_segments: 850 compactness: 30 gray scale conversion flag: cv2.COLOR_RGB2
GRAY
```



Comments on Results Above

- With the gray scale coversion set to COLOR_BGR2GRAY, even though image was actually RGB, the watershed algorithm manges to segement all 7 tomoateooes correctly
- With the gray scale coversion set to COLOR_RGB2GRAY, on an RGB image, the watershed algorithm does not manage to segment the yellow tomato and a red and a green tomoato are actually segmented as a single object
- This result would seem counterintuitive but in my opinion shows how hand tweaking and manipulation of the images is required to get optimal performance

In [ ]: