Introduction to Image Processing and Computer Vision

Laboratory Project 1:

Image segmentation and labeling

Contents:

Bartłomiej Żyła 5.01.2020

1. Introduction

In computer vision, image segmentation is the process of partitioning a digital image into multiple segments. The purpose of segmentation is to simplify and change the representation of an image into something that is more meaningful and easier to analyze. More precisely, image segmentation is the process of assigning a label to every pixel in an image such that pixels with the same label share certain characteristics[1].

In this case our goal is to select plant from the image (and optionally mark it with rectangle – bounding box) and mark (by painting on different colors) each plant leaf. The images vary slightly in background (i.e. a cup or a laptop in a corner) but significantly in plant size and number of leaves due to the fact that the observation sample show the growing process of number of plants.

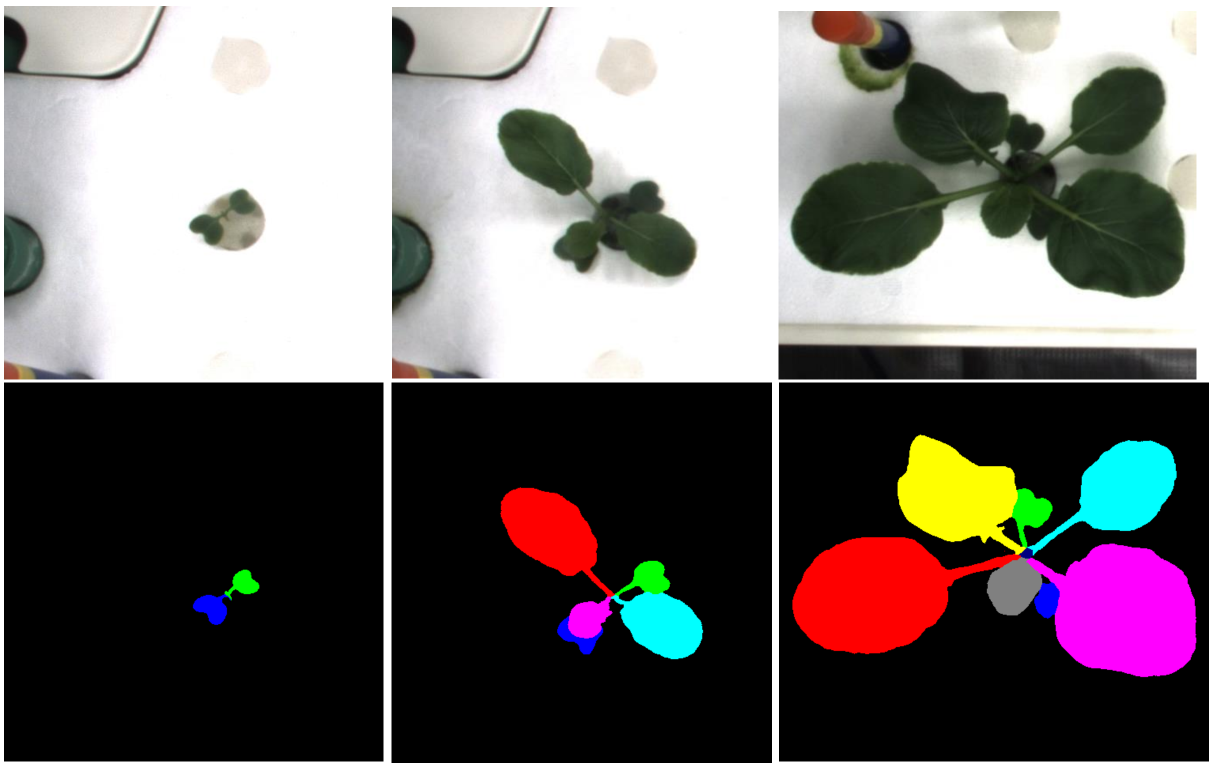
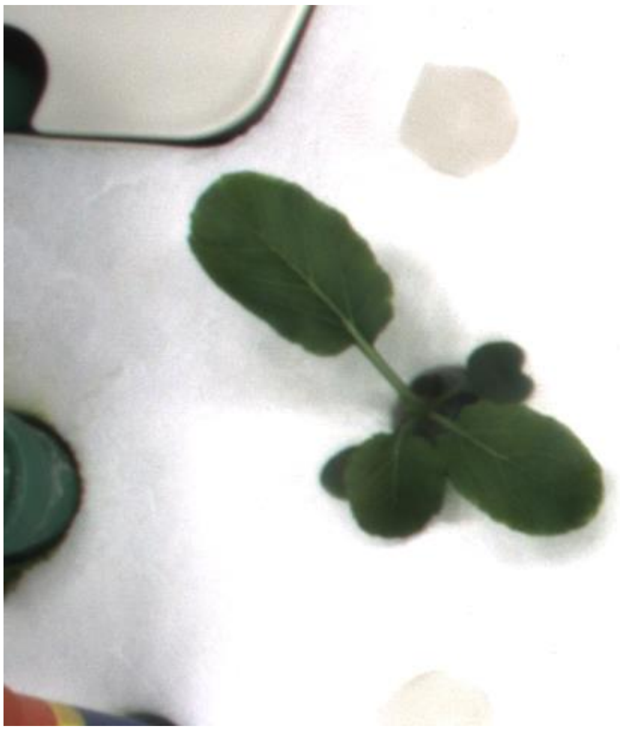


Figure 1: Sample of input images and desired output

1.1 Description of algorythm

This script is divided into two main parts. The first part is responsible for segmenting the whole plant from the image by creating a mask of the plant and putting a bounding box on the original image. The second part goal is to label each leaf as a separate object by painting it on a different color.

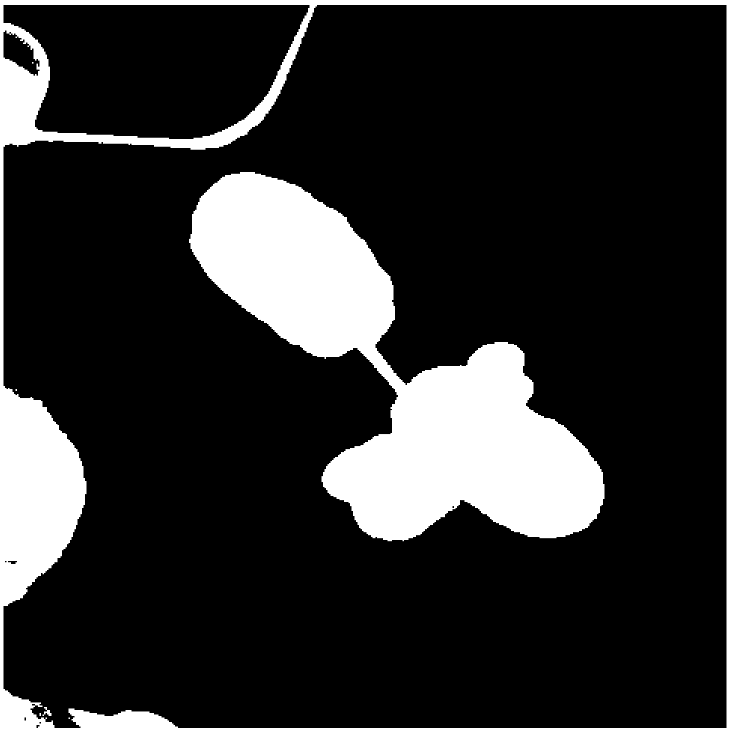


a) Segmentation of the whole plant

The main idea of obtaining the plant's mask is based on searching the image for the plant's color and removing everything, which is not the plant.

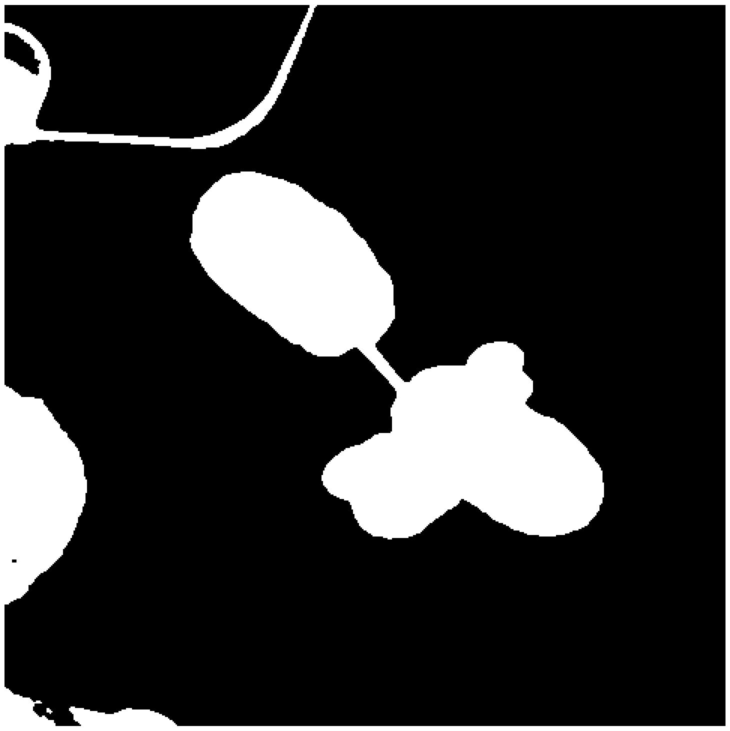
Process of obtaining the mask is started by removing each element which is not in range of particular shades of green color – for the first time the range is wide and BGR palette is used to make sure that entire plant was selected (noise is not relevant yet). The range was determined by selection of color of different parts of plants using Paint.NET software.

```
#Remove every color which is not in range
greenMin = np.array([10 ,20 ,10])
greenMax = np.array([130 ,255 ,150])
mask = cv2.inRange(image, greenMin, greenMax)
```



As we can see, the whole plant was selected as planned but also a lot of background objects got cought into range due to beeing in similar colors. To get rid of them, the best approach (other ideas will be discussed later) is to select the center-most object with restriction that it is big enough to be a plant (to avoid picking up a random single pixel). To do that we first apply *median blur*[2] to reduce the number of objects by blending them together.

```
#Apply first blur to eliminate small noise for contour finding
mask = cv2.medianBlur(mask , 3)
```



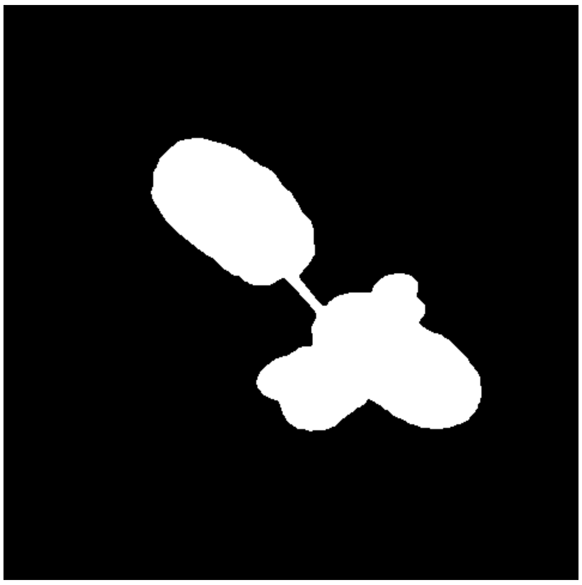
The differences may not seem to be big but our next step is countour detection, thus having greatly reduced number of different objects make it easier to detect the one we are looking for.
We now detect *contours*[4] of all objects in the mask and sort them by the distance from the center of the image using *moments*[5] to find center of mass of each object.

```
#Find contours
ret,thresh = cv2.threshold(mask, 127, 255, 0)
contours, hierarchy = cv2.findContours(thresh,
      cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

#Sort contours and find the closest to the center of the image
using moments
plant_cont = contours[0]
min=image.shape[0]
for i in range(len(contours)):
   if cv2.contourArea(contours[i]) > 500:
       M = cv2.moments(contours[i])
       cX = int(M["m10"] / M["m00"])
       cY = int(M["m01"] / M["m00"])
       Z = sqrt((cX - image.shape[1]/2)**2 + (cY -
            image.shape[0]/2)**2)
       if min > Z:
           min=Z
           plant_cont=contours[i]

#Draw contours and their insides on the empty image to obtain
only plant on the mask
mask = cv2.drawContours(np.zeros((image.shape[0],
      image.shape[1], 3), np.uint8), [plant_cont], 0,
      (255,255,255),  cv2.FILLED)
mask = cv2.cvtColor(mask, cv2.COLOR_BGR2GRAY)
```

After finding our plant among all objects and drawing everything inside it's contours we obtain a more precise mask.

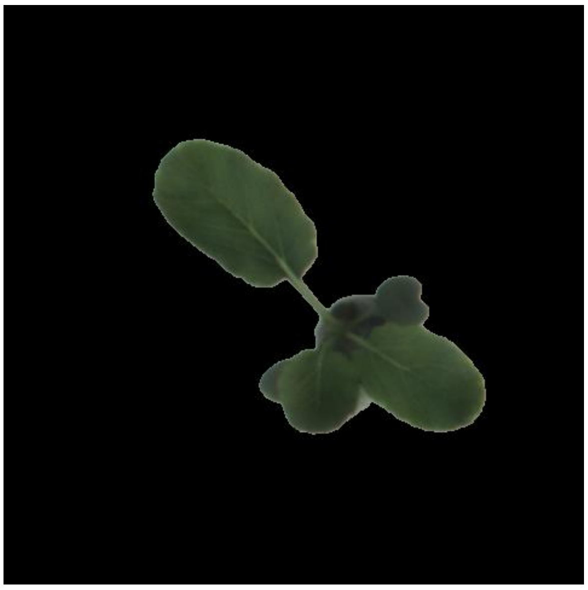
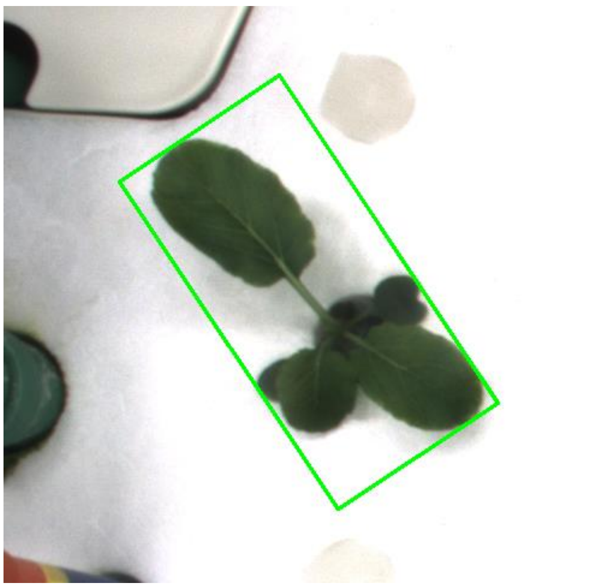


We also use bitwise AND operation to show result of segmentation as well as draw bounding box around the plant on the copy of the original image.

```
#Compute the rotated bounding box of the found contour
rect = cv2.minAreaRect(plant_cont)
box = cv2.boxPoints(rect)
box = np.int0(box)

#Draw bounding box
image_box = image.copy()
cv2.drawContours(image_box, [box], -1, (0, 255, 0), 2)

segmented = cv2.bitwise_and(image , image , mask=mask)
```

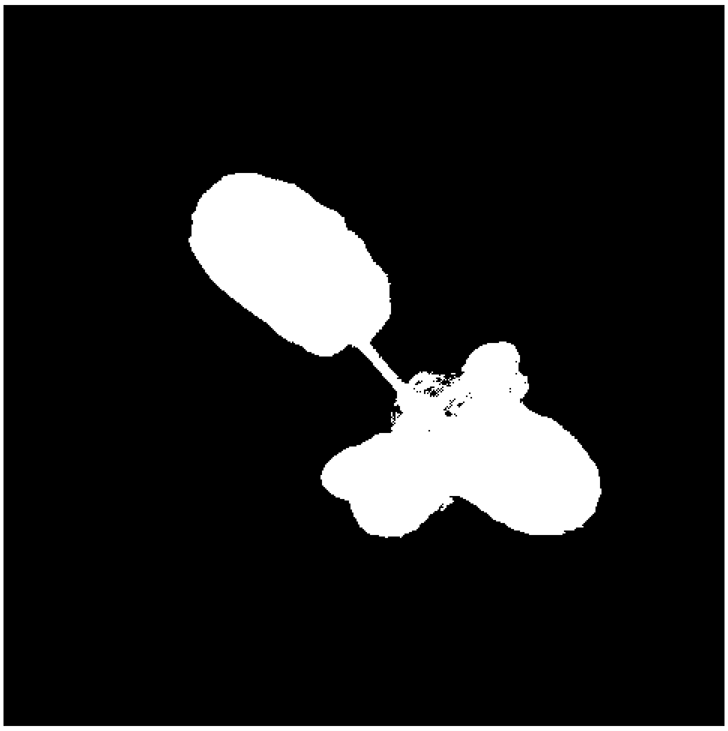



What we are interested in now is to eliminate remaining non-plant objects like shadows and plant's pot. To do that we use narrower range of green color on our segmented image - again with Paint.NET, but this time using HSV palette.

```
#Use narrower range to eliminate non-plant objects like pot
hsv = cv2.cvtColor(segmented , cv2.COLOR_BGR2HSV)
potMin = np.array([35, 30, 30],np.uint8)
potMax = np.array([90, 255, 135],np.uint8)
mask = cv2.inRange(hsv, potMin , potMax)
```

We also use morphology again – although now with *Opening and Closing*[3] to slightly fill holes in our plant and remove single pixels left after applying previous step.

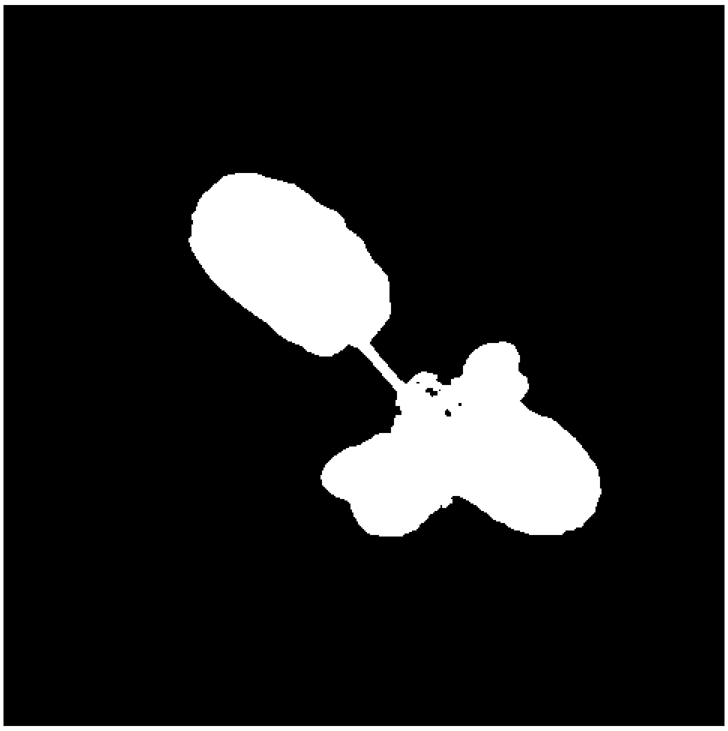
```
#Apply morphology such as opening and closure
kernel = np.ones((1,1),np.uint8)
mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN , kernel)
mask = cv2.morphologyEx(mask ,cv2.MORPH_CLOSE ,kernel)
```



Our final step to further eliminate unwanted elements is to use blur again and also series of *erosions and dilations*[3], where number of iterations is dependent on the size of the plant in order to preserve things like stalk in small plants while also eliminate greater shadows in bigger plants.

```
#Apply second blur and series of erosion and dilation depending on
plant area
    iter = 1
    if cv2.contourArea(plant_cont) > 30000:
        iter = 3
    mask = cv2.medianBlur(mask , 3)
    mask = cv2.erode(mask, None, iterations = int(iter))
    mask = cv2.dilate(mask, None, iterations = int(iter))
```

All those steps gives us our final mask which on average looks like this:



b) Segmentation of each individual leaf

The algorythm of painting each individual leaf on different color is based on the idea of getting positions of centers of every leaf and assigning each pixel to the closest leaf center.

It starts by thresholding the mask, getting coordinates of all white pixels in mask and then turning them into list of tuples.

```
    ret,label_thresh = cv2.threshold(label, 127, 255, 0)

    indices = np.where(label==[255])
    coords = zip(indices[1], indices[0])
    coords = list(coords)
```

Then we apply delicate *dilation*[3] onto the mask to ensure that all leaves are connected and next, we find *contours*[4] of each object in mask (ideally there should be only one) and sort them to get the largest one.

```
    label_dilate = cv2.dilate(label, None, iterations=2)
    contours_label, hierarchy = cv2.findContours(label_thresh,
                    cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
   max_label = sorted(contours_label, key = cv2.contourArea,
                reverse = True)[0]
```

Next step is to compute the center of our biggest object using *moments*[5].

```
    M_label = cv2.moments(max_label)
    cX_label = int(M_label["m10"] / M_label["m00"])
    cY_label = int(M_label["m01"] / M_label["m00"])
```

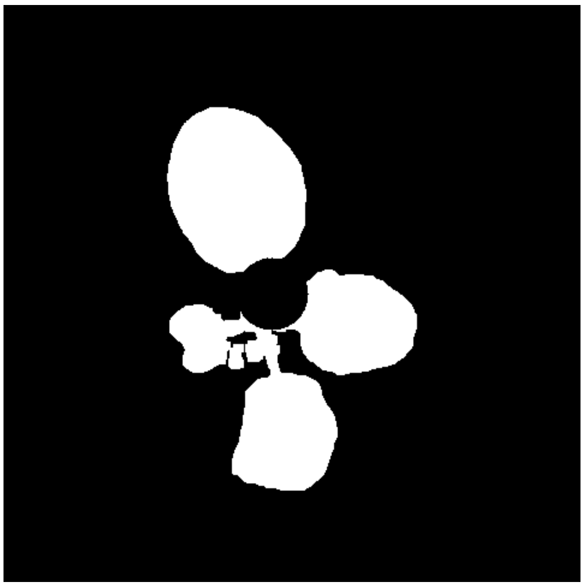
Now we denote 2 special variables depending on the area of the object.

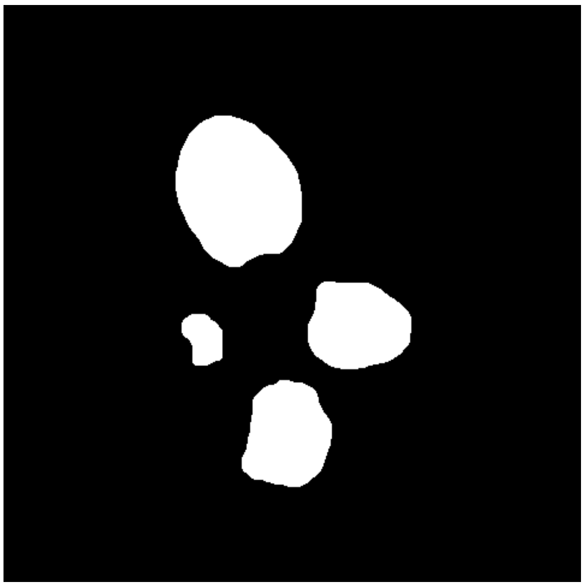
```
    #Adjust variables depending on the size of object
    area = cv2.contourArea(max_label)
    if area > 1000 and area < 4000:
         radius = 10
         scale = floor(area/2000)
    else:
          if area < 10000:
                   radius = 20
                   scale = floor(area/5000)
          else:
                   radius = 30
                   scale = floor(area/8000)
    if area > 40000:
         scale = floor(scale/2)
```

Next, we draw a black circle, with computed radius, at the center of the object followed by the series of *erosions and openings*[3], to hopefully get rid of stalks, pot and shadow, since they are the biggest defect in our mask.

```
    cv2.circle(label, (cX_label,cY_label), radius, color=(0,0,0),
                thickness=-1)

    kernel = np.ones((7+scale,7+scale),np.uint8)
    eroded = cv2.erode(label, kernel, iterations = 1)
    eroded = cv2.morphologyEx(eroded, cv2.MORPH_OPEN, kernel/2,
                iterations=1)
    eroded = cv2.medianBlur(eroded, 5)
```





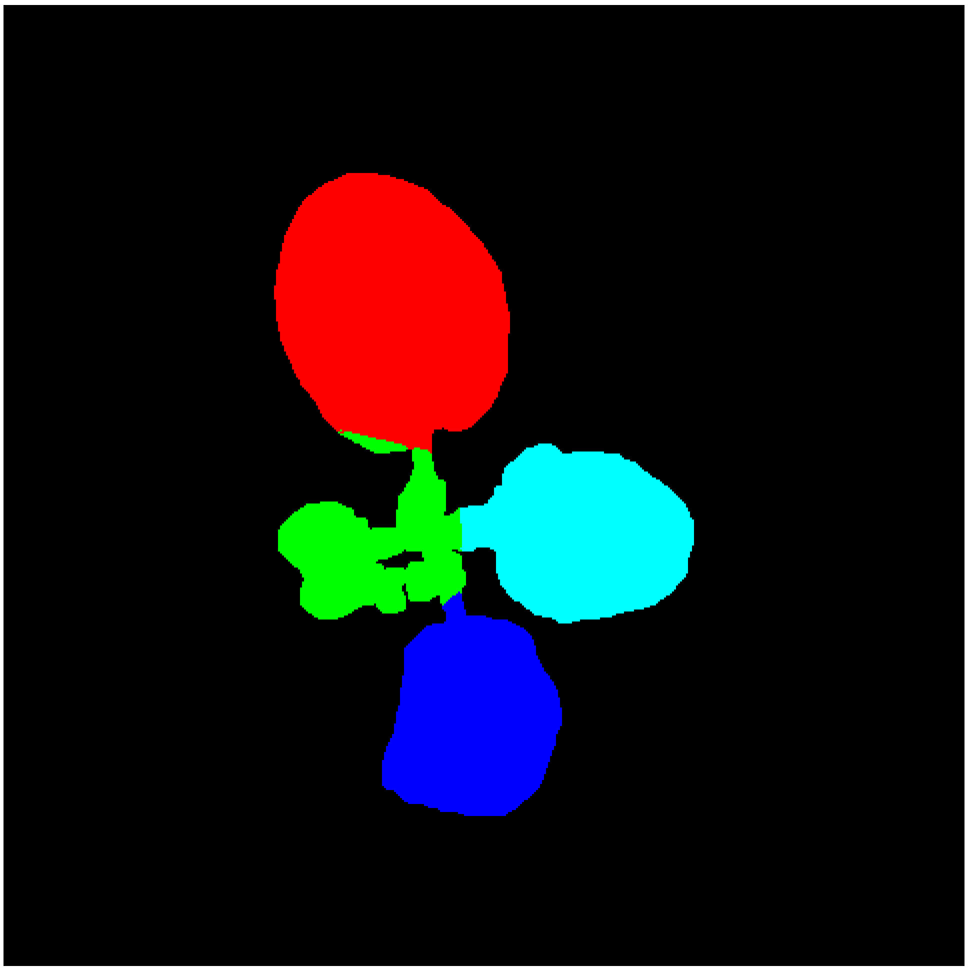
Finally, the last step left is to find the center of each leaf and then by computing and sorting the distance of each white pixel in the mask from every leaf center, assign every pixel into one (the closest in theory) leaf by painting it (i.e. putting small circle of appropriate color in it's place) with the color of this leaf. (It may not and most certainly isn't the most efficient way of painting the mask, but it was the only one that worked – more about this in section 2.2 - failed ideas)

```
ret,thresh = cv2.threshold(eroded, 127, 255, 0)
           contours, hierarchy = cv2.findContours(thresh,
           cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

zeros = np.zeros((label.shape[0], label.shape[1], 3),
     dtype=np.uint8)
colors=[(255,0,0),(0,255,0),(255,255,0),(0,0,255),(255,0,255),(
     0,255,255),(128,128,128),(128,0,0)]

cX = []
cY = []
for i in range(len(contours)):
   M = cv2.moments(contours[i])
   cX.append(int(M["m10"] / M["m00"]))
   cY.append(int(M["m01"] / M["m00"]))

for xy in coords:
   min = label.shape[0]
   index = 0
   for i in range(len(contours)):
       dist = sqrt(((xy[0] - cX[i])**2) + ((xy[1] - cY[i])**2))
       if dist < min:
           min = dist
           index = i
   cv2.circle(zeros, xy, 1, color=colors[index])
```



2. Results

2.1 Results, tests and conclusions

The assesment criteria was the Intersection over Union metric and the Dice coefficient.

For the first part (masks), the results are comparatively good – the mean score for IoU was 88.33% while the mean Dice coefficient was 93.65%.

When we look at the highest scores: IoU – 98.27% & Dice – 99.13% we can observe that the algorythm works best when the leaves are far from both the plant’s center and themselves, which prevents them from overlapping and creating shadows.

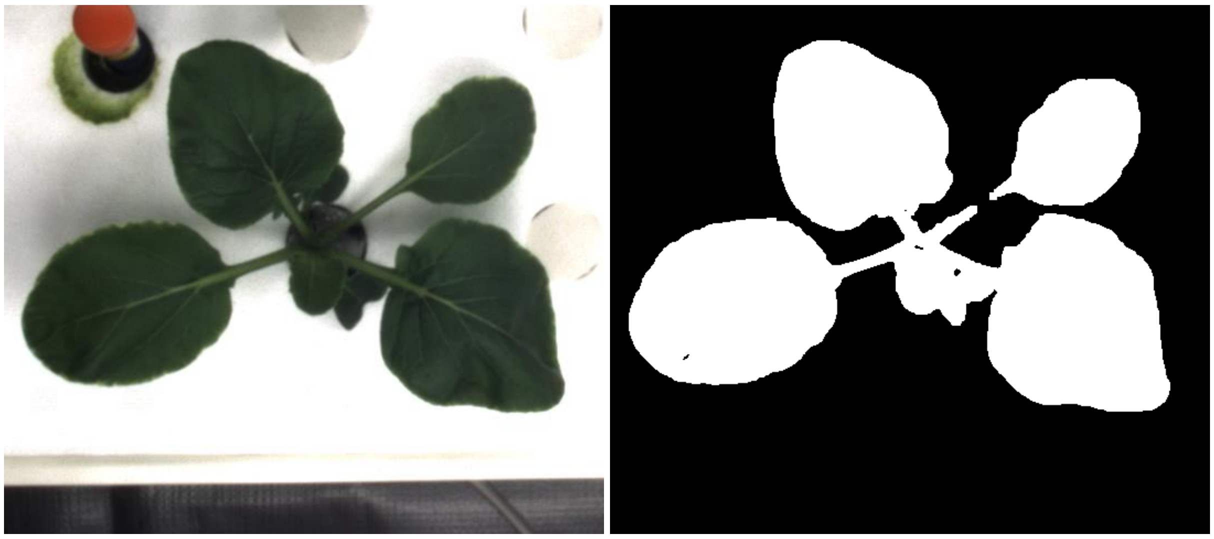


Figure 2: Image with best percentage result (rgb_02_04_009_05.png)

We also have to mention that masks for smaller plant were on average also very accurate, but due to their smaller size the score was visibly lower (i.e. IoU – 87.92% & Dice – 93.57% for the plant below) because for them, all imperfections mean a lot more.

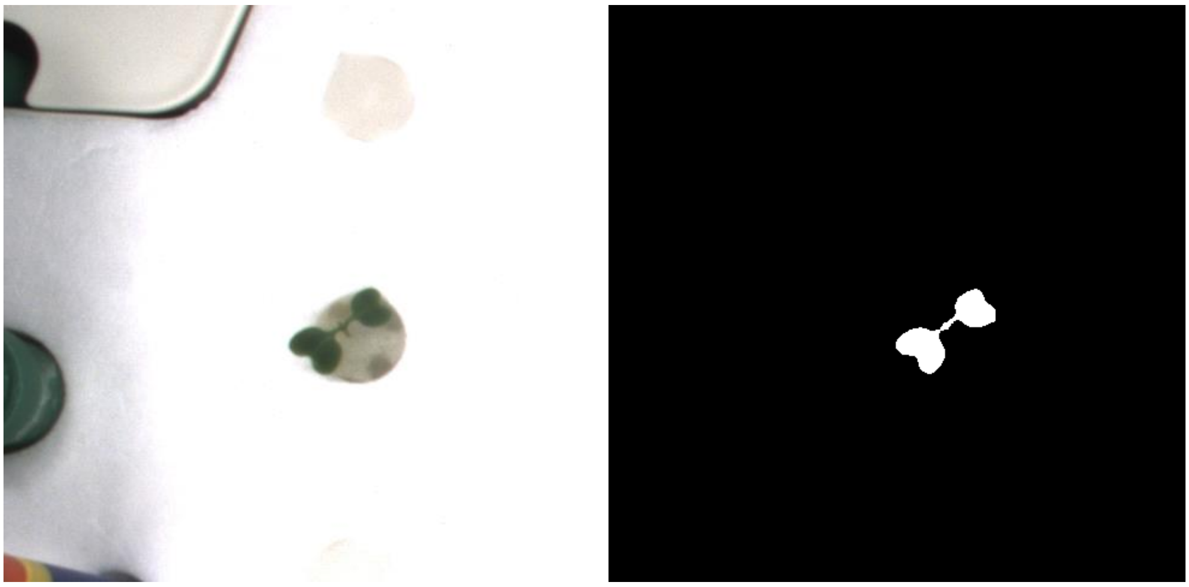


Figure 3: Image with good result but with lower percentage score (rgb_00_00_000_00.png)

From the mask with the lowest scores: IoU – 68.71% & Dice: 81.46% we can draw the conclusion that the biggest obsctacle on the way was the plants's pot combined with the shadow that the leaves left on the table. Due to the fact that their colors was both in BGR and HSV color palettes very similar to the colors of some part of plant, it was almost impossible to get rid of them, while simultanously not erasing part of the plant. When we consider that the algorythm was to be used on all plants, not on the individual one, then idea of getting slightly lower accuracy on number of plants was the better alternative than erasing half of the plant on the some images to keep the others more accurate.

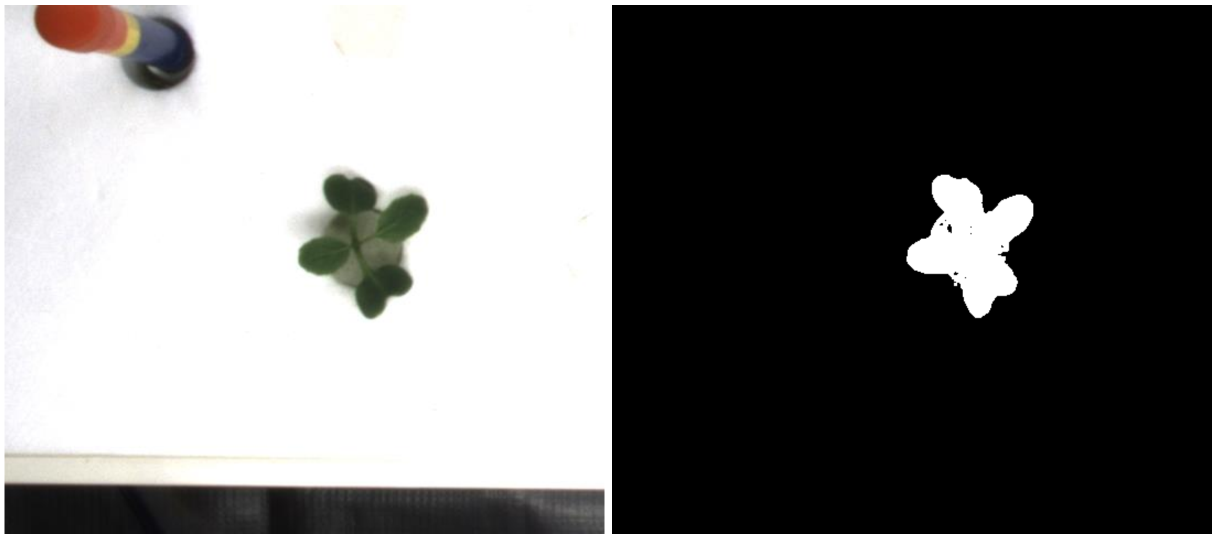
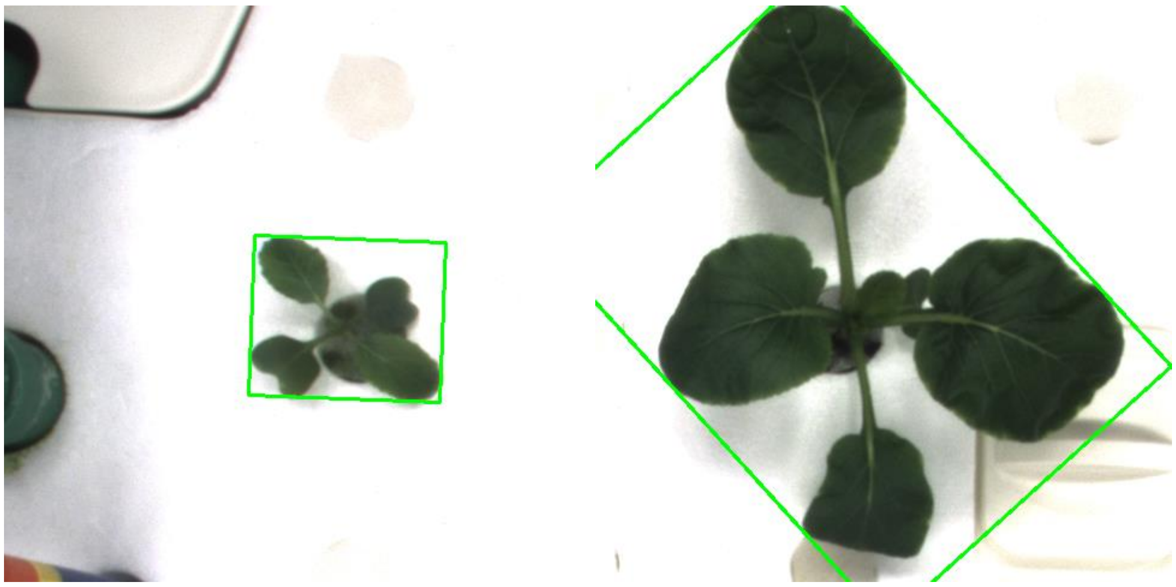


Figure 4: Image with worst percentage result (rgb_01_04_002_03.png)

The additional part which was putting the bounding boxes on the plants was almost complete success exept for some individual cases where plant was mixed with some object in the background such as laptop or a cup, since they had the same color as plant and overlapped the leaves.



As for the second part (labels), the results were significantly worse: IoU – 39.98% & Dice – 47.33%.

While there still were plants with such a high score that it is difficult to tell which one is which, with the highest of IoU – 93.27% & Dice – 96.52% presented below, they were rare cases.

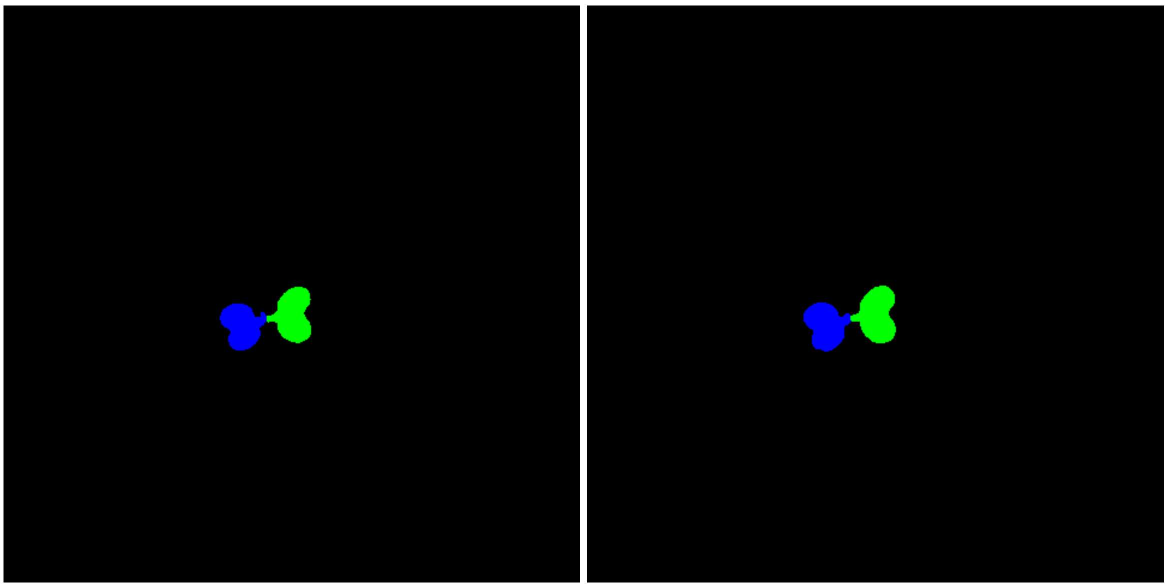
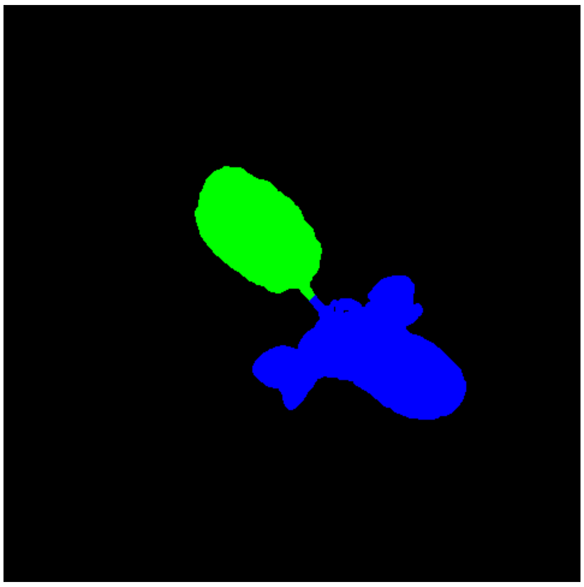
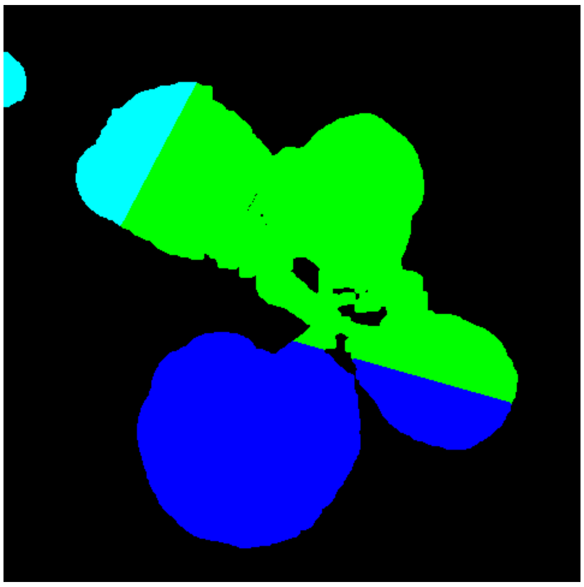


Figure 5: Label with best percentage result (rgb_00_01_000_04.png)

The reasons for such situation are clearly visible after closer inspection of the problem. There are 2 factors which caused the results to be poor.

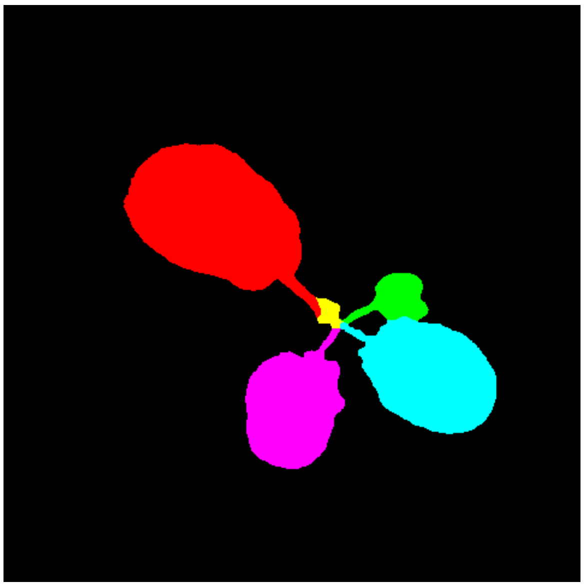
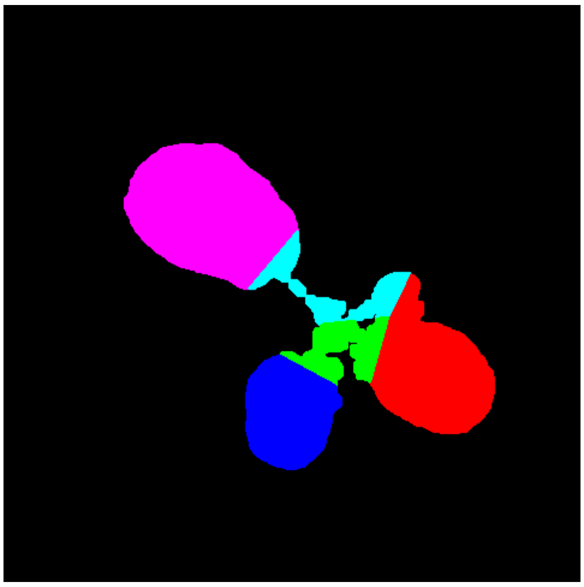
1) Imperfect masks

Just as discussed in previous section, the choice was made to pick lower decrease in leaf mass rather than getting rid of pot and shadows. While this solution made the masks to be more accurate, it also made labeling significantly more difficult, due to merging of pot and shadows with smaller leaves or blending some leaves together by shadow between them. Above situation resulted is such cases:

2) Color order

At first glance, the order at which leaves were colored in the sample may not look really important, but it sadly was the most significant factor which caused huge differences between algorithm-colored masks and sample labels. In the sample the leaves were colored in such a way that the blue color (0, 0, 255) was assigned to the oldest leaf. The second oldest leaf got the green color (0, 255, 0) and so on. While for human, who is looking at the set of images, which presents plant's growing process from the beginning to the fully grown state, it is very easy to determine leaves seniority, for algorithm which is processing each image individually this tasks is almost impossible. The algorithm assigns the colors in the same order as in the sample but which leaf is given which color is determined by the order of the contours which was found. This single fact causes the situations, where during comparison, the blue leaf segmented by algorithm is totally different than the blue leaf from labels sample which in result have around 0-1% of both IoU & Dice score.

2.2 Failed ideas

During the process of creating the algorythms, a lot of ideas were tested and here are some promising examples, which eventually didn't work or had to be abandoned.

1) More specific masks for removing non-plant objects

At the beginning the idea of using many masks containing specific ranges of colours to get rid of green-like objects from the main mask using bitwise operations seemed like a good idea. However during the algorythm development it became clear that those objects vary too much and too many masks would have to be made (all color ranges picked manually) for this idea to be efficient, especially when after each 2 masks additions, the series of *closing*[3] had to be made to compensate for loss of leaf mass.

```
hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
badmask = cv2.inRange(hsv, (70,50,40), (90,255,255))
badmask2 = cv2.inRange(hsv, (30,40,100), (50,70,150))
cv2.imshow("pot", badmask2);
cv2.imshow("preMask", mask)
badmask2 = cv2.bitwise_not(badmask2)
mask = cv2.bitwise_xor(mask, badmask)
mask = cv2.bitwise_and(mask, badmask2)
```

2) Idea of searching for the biggest object

This rather short-living idea was about finding a plant for mask creation by searching the image for the biggest object. It's flaw laid in a fact that in early stages of plant growth, other objects such as cup in a image corner had bigger area than the small plant which resulted in wrong object selection. As discussed before, this idea contradicts the chosen startegy of optimal mask detection for every plant, rather than perfect result for some of them.

```
contours = cv2.findContours(mask,
            cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIMPLE)
c = sorted(contours, key = cv2.contourArea, reverse = True)[0]
mask = cv2.drawContours(np.zeros((image.shape[0],
      image.shape[1], 3), np.uint8),[c], 0, (255,255,255),
      cv2.FILLED)
```

3) Other painting options

While it may seem to be much better in case of efficiency, to paint the whole leaf at once, it came with 2 big disadvantages.
Firstly, OpenCV functions for painting objects like fillPoly() were likely to throw an error which made it difficult to have one algorythm working for each image.
Secondly, the approach for treating each pixel separately benefited in making the score much higher than having few objects as complete leaves. Hence painting the whole leaf at once results in much lower accuracy.

The idea of painting a pixel by putting a very small pixel on it may seem a bit abstract and not straight-forward, but another approach like simply changing each pixel color to the desired one, came with unexpected problem. Pixels at the image edges, that were to be painted, threw en error of out of bound index. To solve this issue we can either make a huge if/else block checking if the pixel is on the image's edge, which greatly decreases perfomance due to the number of comparisons for each single pixel,

```
if(xy[0] >= label.shape[0] and xy[1] >= label.shape[1]):
        zeros[label.shape[0]-1][label.shape[1]-1] =
        colors[index]
   else:
        if xy[0] >= label.shape[0]:
            zeros[label.shape[0]-1][xy[1]] = colors[index]
        else:
            if xy[1] >= label.shape[1]:
                zeros[xy[0]][label.shape[1]-1] = colors[index]
            else:
                zeros[xy[0]][xy[1]] = colors[index]
```

or make the output image's size 1 pixel bigger than the original which then causes problems with image comparison since their sizes don't match.

```
zeros = np.zeros((label.shape[0]+1, label.shape[1]+1, 3),
          dtype=np.uint8)
zeros[xy[0]][xy[1]] = colors[index]
```

3. References

[1] https://en.wikipedia.org/wiki/Image_segmentation

[2] https://docs.opencv.org/2.4/doc/tutorials/imgproc/gausian_median_blur_bilateral_filter/gausian_median_blur_bilateral_filter.html

[3] https://docs.opencv.org/3.4/d9/d61/tutorial_py_morphological_ops.html

[4] https://docs.opencv.org/3.4/dd/d49/tutorial_py_contour_features.html

[5] https://www.pyimagesearch.com/2016/02/01/opencv-center-of-contour/