20171097_assignment3

March 18, 2020

[1]:
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from sklearn.mixture import GaussianMixture
from sklearn.cluster import KMeans
import math
import maxflow
```

1 Grab Cut - Ajay Shrihari

1.1 Method

- We use the Grab cut algorithm for foreground segmentation. In the implementation, there are four classes - 0 -> background, 2 -> foreground, 3 -> probably background, 4 -> probably background. The bounding box given in the dataset/selected by the programmed interactive tool gives a rectangle, which is the probable foreground class. We then learn the GMM models for the foreground and background, and weight the neighbouring nodes based on the class labels edges from source and sink. Based on these weights, a graph is constructed. Edge weights are assigned with likelihoods based on the class. A mincut breaks the graph, and we iteratively repeat this process until we get convergence.
- The library PyMaxflow was used for finding the mincut/maxflow.

[151]:
```
def get_mask(img, rect):
    '''
    Create mask based on bounding box
    '''
    mask = np.zeros(img.shape[:2])
    mask[rect[1]:rect[3],rect[0]:rect[2]] = 3
    return mask
```

[152]:
```
def GMM(img, mask, bkgGMM, fgGMM):
    '''
    Assign pixels to foreground and background. Fit GMM's
    '''
    fg = []
    bkg = []
```

```
    x = img.shape[0]
    y = img.shape[1]
    for i in range(x):
        for j in range(y):
            if (int(mask[i][j]) == 0 or int(mask[i][j]) == 2):
                bkg.append(img[i][j])
            else:
                fg.append(img[i][j])
    return bkgGMM.fit(np.array(bkg)), fgGMM.fit(np.array(fg))
```

[153]:
```
def find_beta(img, num_neighbour = 8):
    '''
    Function to find beta. 8 neighbours and 4 neighbours.
    4 neighbours - left and up
    8 neighbours - left, upper left, up, upper right
    Beta value used to find edge weights of neighbours in the making of graph
    '''
    if num_neighbour == 8:
        beta = 0
        x = img.shape[0]
        y = img.shape[1]
        for i in range(x):
            for j in range(y):
                pix = img[i][j]
                if (j>0):
                    d = pix - img[i][j-1]
                    beta = beta + np.dot(d,d)
                if (i>0 and j>0):
                    d = pix - img[i-1][j-1]
                    beta = beta + np.dot(d, d)
                if (i>0):
                    d = pix - img[i-1][j]
                    beta = beta + np.dot(d, d)
                if (i>0 and j<y-1):
                    d = pix - img[i-1][j+1]
                    beta = beta + np.dot(d, d)
        num_edge = (4*y*x - 3*y - 3*x +2)
        return 1.00/(2*beta/num_edge)
    if num_neighbour == 4:
        beta = 0
        x = img.shape[0]
        y = img.shape[1]
        for i in range(x):
            for j in range(y):
                pix = img[i][j]
                if (j>0):
```

```
                    d = pix - img[i][j-1]
                    beta = beta + np.dot(d,d)

                if (i>0):
                    d = pix - img[i-1][j]
                    beta = beta + np.dot(d, d)

        num_edge = (2*y*x - y - x +2)
        return 1.00/(2*beta/num_edge)
```

[155]:
```
def weights(img, beta, gamma, num_neighbour = 8):
    """
    Calculate weights for vertices that are not the source and sink
    Beta and gamma - inputs, calculated in different functions
    """
    if num_neighbour == 8:
        x = img.shape[0]
        y = img.shape[1]
        gamma2 = gamma/math.sqrt(2.00)
        lw, ulw, uw, upw  = np.zeros((x,y)), np.zeros((x,y)), np.zeros((x,y)),␣
 ↪np.zeros((x,y))
        for i in range(x):
            for j in range(y):
                pix = img[i][j]
                if (j>=1):
                    d = pix - img[i][j-1]
                    lw[i][j] = gamma * math.exp(-beta*np.dot(d,d))
                if (j>=1 and i>=1):
                    d = pix - img[i-1][j-1]
                    ulw[i][j] = gamma2 * math.exp(-beta*np.dot(d,d))
                if (i>=1):
                    d = pix - img[i-1][j]
                    uw[i][j] = gamma * math.exp(-beta*np.dot(d,d))
                if (j<y-1 and i>=1):
                    d = pix - img[i-1][j+1]
                    upw[i][j] = gamma2 * math.exp(-beta*np.dot(d,d))
        return lw, ulw, uw, upw
    if num_neighbour == 4:
        x = img.shape[0]
        y = img.shape[1]
#         gamma2 = gamma/math.sqrt(2.00)
        lw, uw  = np.zeros((x,y)), np.zeros((x,y))
        for i in range(x):
            for j in range(y):
                pix = img[i][j]
                if (j>=1):
```

```
                    d = pix - img[i][j-1]
                    lw[i][j] = gamma * math.exp(-beta*np.dot(d,d))
                if (i>=1):
                    d = pix - img[i-1][j]
                    uw[i][j] = gamma * math.exp(-beta*np.dot(d,d))

        return lw, uw
```

[157]:
```
def make_graph(img, mask, bkgGMM, fgGMM, lmbda, lw,ulw,  uw, upw, num_neighbour
↪= 8):
    """
    Build graph and add edges
    """
    if num_neighbour == 8:
        x = img.shape[0]
        y = img.shape[1]
        num_edge = 2* (4*y*x - 3*y - 3*x +2)
        num_vertex = x * y
        graph = maxflow.Graph[float](num_vertex, num_edge)
        graph.add_nodes(num_vertex)
        src = -bkgGMM.score_samples(img.reshape(x*y,3))
        snk = -fgGMM.score_samples(img.reshape(x*y,3))
        for i in range(x):
            for j in range(y):
                v_id = i*img.shape[1] + j
                pix = img[i][j]
                source = None
                sink = None
                if (mask[i][j] == 2 or mask[i][j] == 3):
                    source = src[v_id]
                    sink = snk[v_id]
                elif (mask[i][j] == 0):
                    source = 0
                    sink = lmbda
                else:
                    source = lmbda
                    sink = 0
                graph.add_tedge(v_id, source, sink)
                if (j>0):
                    graph.add_edge(v_id, v_id-1, lw[i][j], lw[i][j])
                if (j>0 and i>0):
                    graph.add_edge(v_id, v_id-y-1, ulw[i][j], ulw[i][j])
                if (i>0):
                    graph.add_edge(v_id, v_id-y, uw[i][j], uw[i][j])
                if (j<y-1 and i>0):
                    graph.add_edge(v_id, v_id -y+1, upw[i][j], upw[i][j])
```

```
        return graph
        if num_neighbour == 4:
            x = img.shape[0]
            y = img.shape[1]
            num_edge = 2* (4*y*x - 3*y - 3*x +2)
            num_vertex = x * y
            graph = maxflow.Graph[float](num_vertex, num_edge)
            graph.add_nodes(num_vertex)
            src = -bkgGMM.score_samples(img.reshape(x*y,3))
            snk = -fgGMM.score_samples(img.reshape(x*y,3))
            for i in range(x):
                for j in range(y):
                    v_id = i*img.shape[1] + j
                    pix = img[i][j]
                    source = None
                    sink = None
                    if (mask[i][j] == 2 or mask[i][j] == 3):
                        source = src[v_id]
                        sink = snk[v_id]
                    elif (mask[i][j] == 0):
                        source = 0
                        sink = lmbda
                    else:
                        source = lmbda
                        sink = 0
                    graph.add_tedge(v_id, source, sink)
                    if (j>0):
                        graph.add_edge(v_id, v_id-1, lw[i][j], lw[i][j])

                    if (i>0):
                        graph.add_edge(v_id, v_id-y, uw[i][j], uw[i][j])

            return graph
```

[158]:
```
def mincut_maxflow(graph, mask):
    """
    Use PyMaxflow to calculate the mincut of the graph
    """
    x = mask.shape[0]
    y = mask.shape[1]
    minimum = graph.maxflow()
    partition = graph.get_grid_segments(np.arange(mask.shape[0]*mask.shape[1])).
↪reshape(mask.shape).astype('int')
    partition[partition==1] = 2
    partition[partition==0] = 3
```

```
    mask = (mask>0)*partition
    return mask
```

[159]:
```
def grabcut(img, rect, num_iter = 2, num_components = 3, num_neighbour= 8, gamma␣
↪= 50):
    """
    Function to tie everything together
    """
    bkgGMM = GaussianMixture(n_components = num_components, covariance_type =␣
↪'full')
    fgGMM = GaussianMixture(n_components = num_components, covariance_type =␣
↪'full')
    mask = get_mask(img, rect)
    gamma = gamma
    lmbda = 9 * gamma
    beta = find_beta(img, num_neighbour = num_neighbour)
    if num_neighbour == 8:
        lw, ulw, uw, upw = weights(img, beta, gamma, num_neighbour =␣
↪num_neighbour)
    if num_neighbour == 4:
        lw, uw = weights(img, beta, gamma, num_neighbour = num_neighbour)
        ulw = []
        upw = []

    for i in range(num_iter):
        bkgGMM, fgGMM = GMM(img, mask, bkgGMM, fgGMM)
        graph = make_graph(img, mask, bkgGMM, fgGMM, lmbda, lw, ulw, uw, upw,␣
↪num_neighbour = num_neighbour)
        mask = mincut_maxflow(graph, mask)
        mask_corr = np.where((mask == 1)+(mask == 3), 255, 0).astype('uint8')
        out = cv2.bitwise_and(img, img, mask = mask_corr)
#         cv2.imwrite()
    return out
```

[13]:
```
def image_list(name):
    dir_im = './images'
    path = dir_im + '/' + name + '.jpg'
    return plt.imread(path)
```

[14]:
```
def bbox_list(name):
    dir_box = './bboxes/'
    with open(dir_box + '/' + name + '.txt', "r") as f:
        for line in f:
            x = line.split(' ')
    return tuple(map(int, x))
```

[160]:
```
def interactive(img):
    """
    Function to make interactive GUI application for selecting foreground␣
 ↪bounding box
    """
    r = cv2.selectROI(img,False,False)
    cv2.destroyAllWindows()
    return [r[0],r[1],r[0]+r[2],r[1]+r[3]]
```

1.2 1. Testing output for different images

- Flower for 2 iterations.
- Banana1 for 3 iterations.
- Accurate segmentation received.

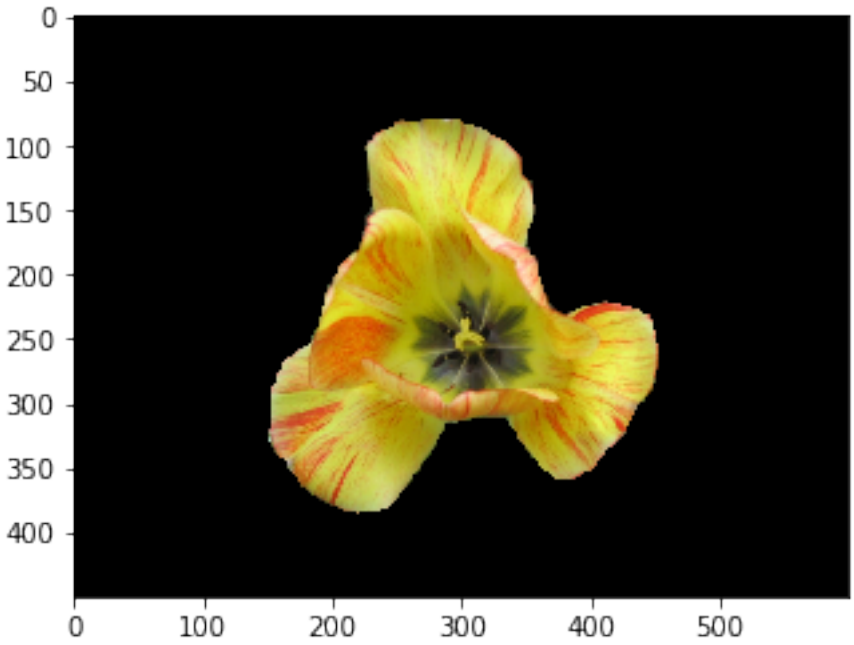
[145]:
```
name = 'flower'
img = image_list(name)
out = grabcut(image_list(name), bbox_list(name),2)
```

[146]:
```
plt.imshow(img)
```

[146]:
```
<matplotlib.image.AxesImage at 0x7fa189778748>
```

[147]:
```
plt.imshow(out)
```

[147]: <matplotlib.image.AxesImage at 0x7fa1896db8d0>

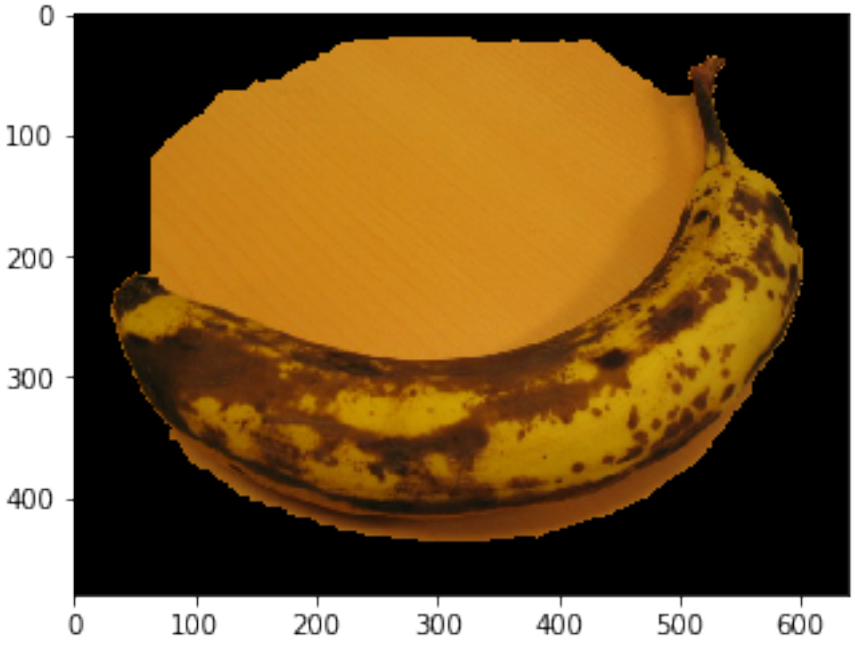


[148]:
```
name = 'banana1'
img = image_list(name)
out = grabcut(image_list(name), bbox_list(name),3)
plt.imshow(img)
```

[148]: <matplotlib.image.AxesImage at 0x7fa1896c3da0>

[149]:
```
plt.imshow(out)
```

[149]:
```
<matplotlib.image.AxesImage at 0x7fa189621b00>
```



1.3 2. Interactive output

cv2.SelectROI was used for this process. The output of the flower with the bounding box selected by the user is given below.

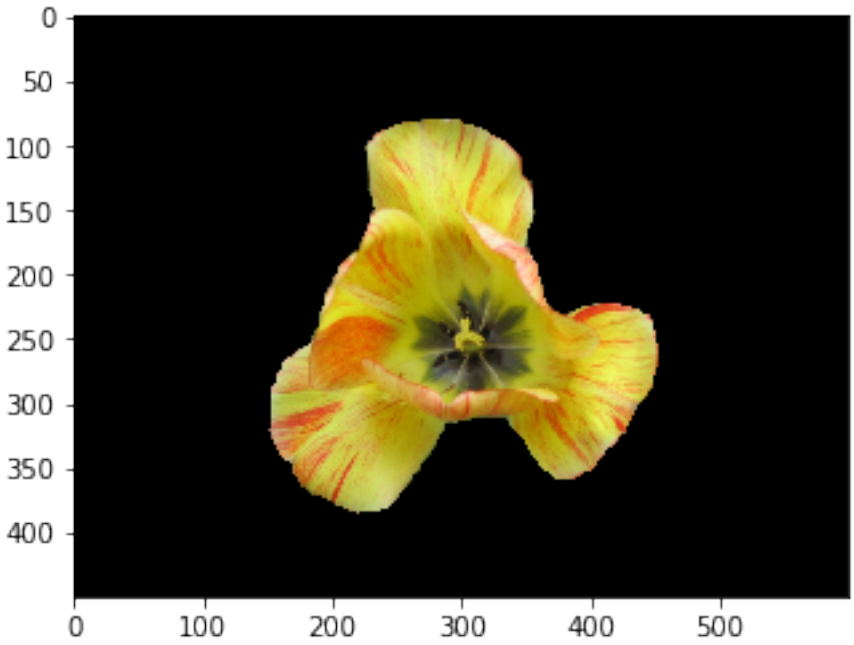
[16]:
```
name = 'flower'
img = image_list(name)
bbox = interactive(img)
out = grabcut(image_list(name), bbox,3)
```

[41]:
```
plt.imshow(img)
```

[41]:
```
<matplotlib.image.AxesImage at 0x7fa1955b6898>
```

[17]:
```
plt.imshow(out)
```

[17]:
```
<matplotlib.image.AxesImage at 0x27fabc1f708>
```



1.4 3.Varying number of iterations

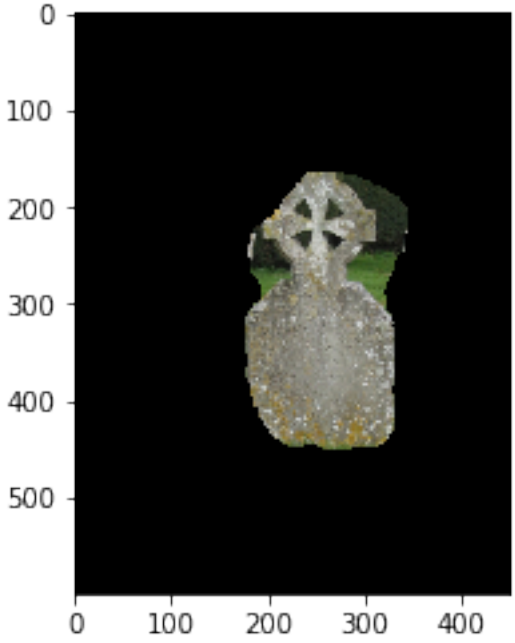
- Grave picture used, with iterations 1-10 tested
- We can see from this that with increased number of iterations, the segmentation gets better

[49]:
```
## Varying number of iterations
name = 'grave'
img = image_list(name)
out1 = grabcut(image_list(name), bbox_list(name),1)
out2 = grabcut(image_list(name), bbox_list(name),2)
out3 = grabcut(image_list(name), bbox_list(name),3)
out3 = grabcut(image_list(name), bbox_list(name),4)
out4 = grabcut(image_list(name), bbox_list(name),4)
out5 = grabcut(image_list(name), bbox_list(name),5)
out6 = grabcut(image_list(name), bbox_list(name),6)
out7 = grabcut(image_list(name), bbox_list(name),7)
out8 = grabcut(image_list(name), bbox_list(name),8)
out9 = grabcut(image_list(name), bbox_list(name),9)
out10 = grabcut(image_list(name), bbox_list(name),10)
plt.imshow(img)
```

[49]:
```
<matplotlib.image.AxesImage at 0x7fa194eb60f0>
```

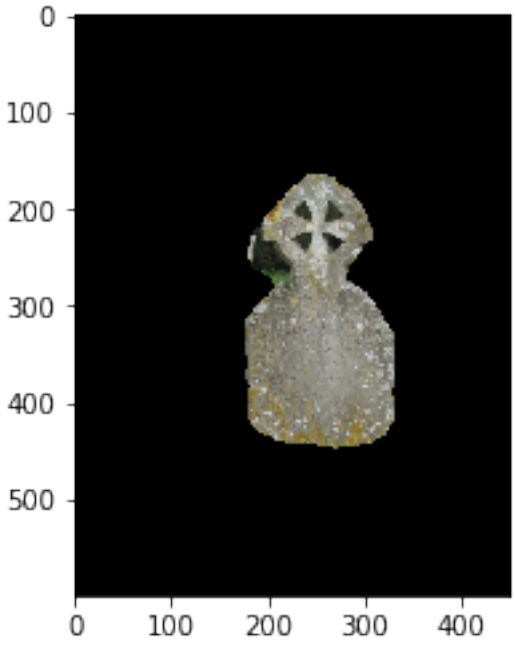
[51]:
```
plt.imshow(out1)
```

[51]: <matplotlib.image.AxesImage at 0x7fa194ebc438>



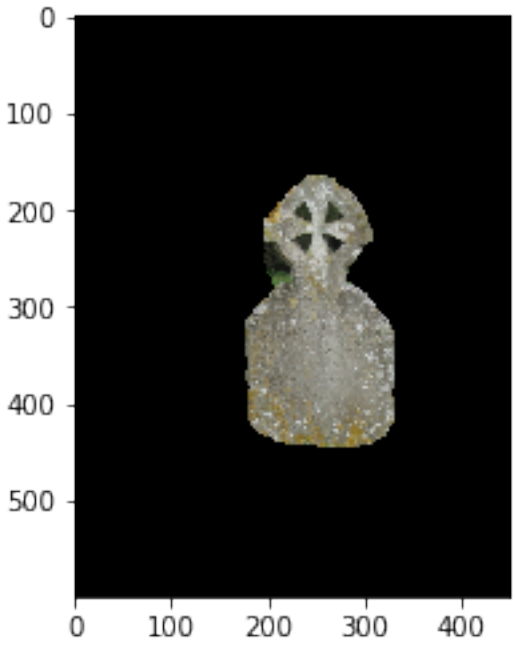
[52]:
```
plt.imshow(out2)
```

[52]:
```
<matplotlib.image.AxesImage at 0x7fa1954db588>
```



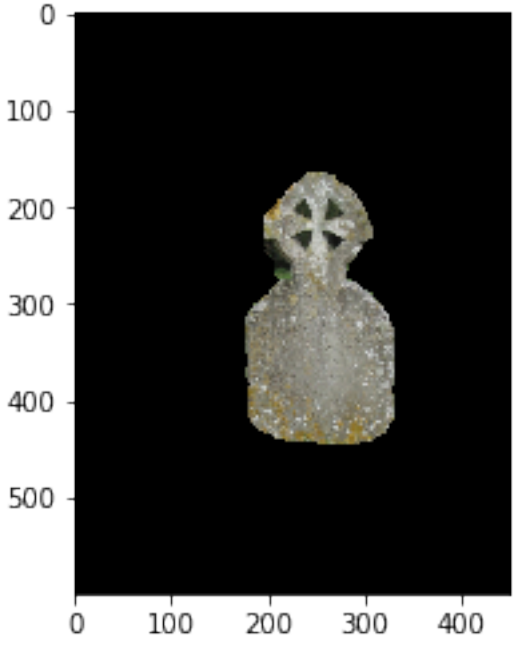
[53]:
```
plt.imshow(out3)
```

[53]:
```
<matplotlib.image.AxesImage at 0x7fa194f2d5c0>
```



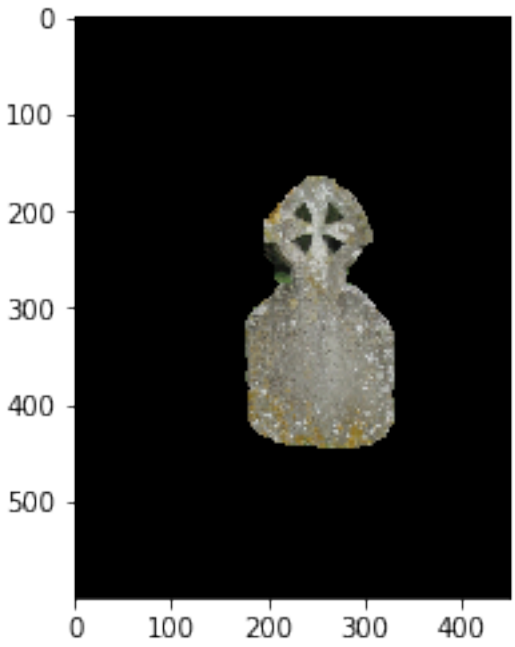
[54]:
```
plt.imshow(out4)
```

[54]: <matplotlib.image.AxesImage at 0x7fa195247198>



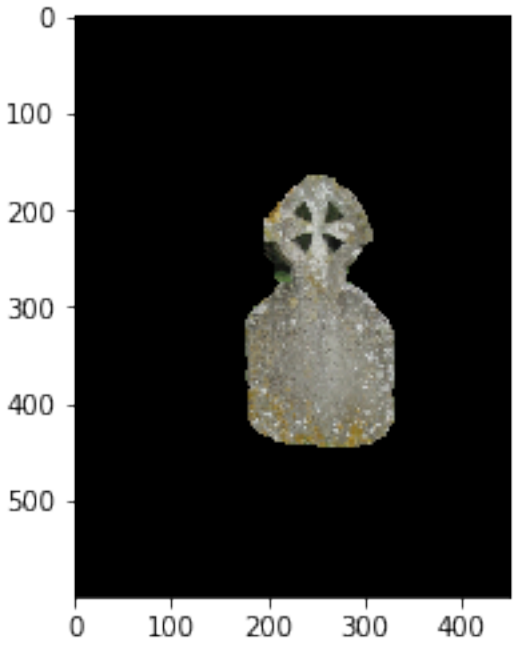
[55]:
```
plt.imshow(out5)
```

[55]: <matplotlib.image.AxesImage at 0x7fa1954e6d30>



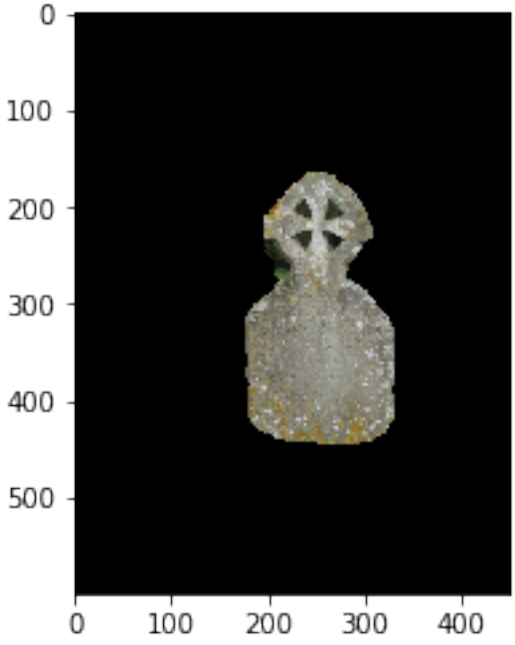
[56]:
```
plt.imshow(out6)
```

[56]: <matplotlib.image.AxesImage at 0x7fa19527d908>



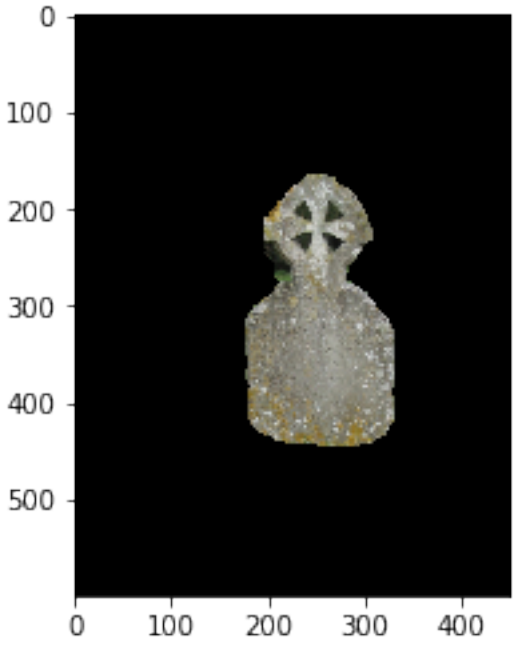
[57]:
```
plt.imshow(out7)
```

[57]: <matplotlib.image.AxesImage at 0x7fa194f5b4e0>



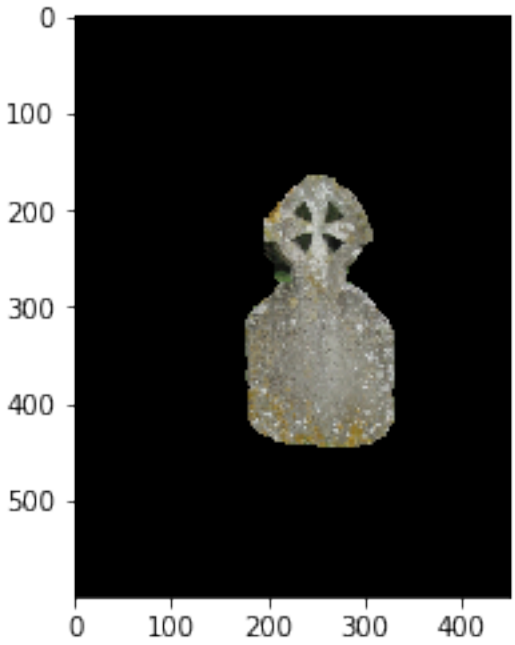
[58]:
```
plt.imshow(out8)
```

[58]: <matplotlib.image.AxesImage at 0x7fa194fc00b8>



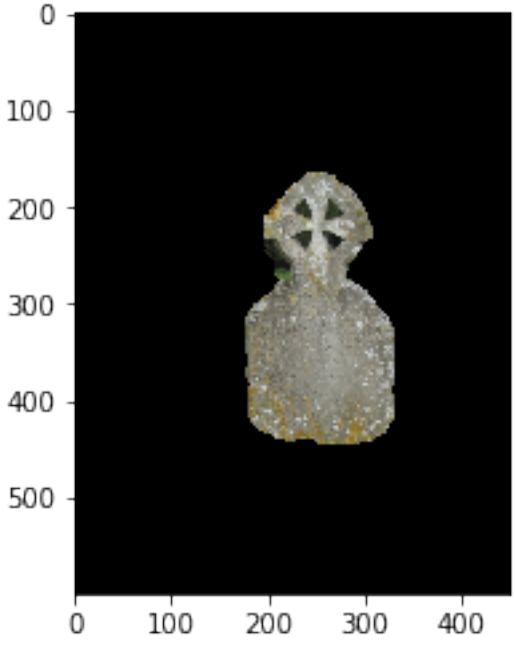
[59]:
```
plt.imshow(out9)
```

[59]: <matplotlib.image.AxesImage at 0x7fa1953a0c50>



[60]:
```
plt.imshow(out10)
```

[60]: <matplotlib.image.AxesImage at 0x7fa194fe3828>

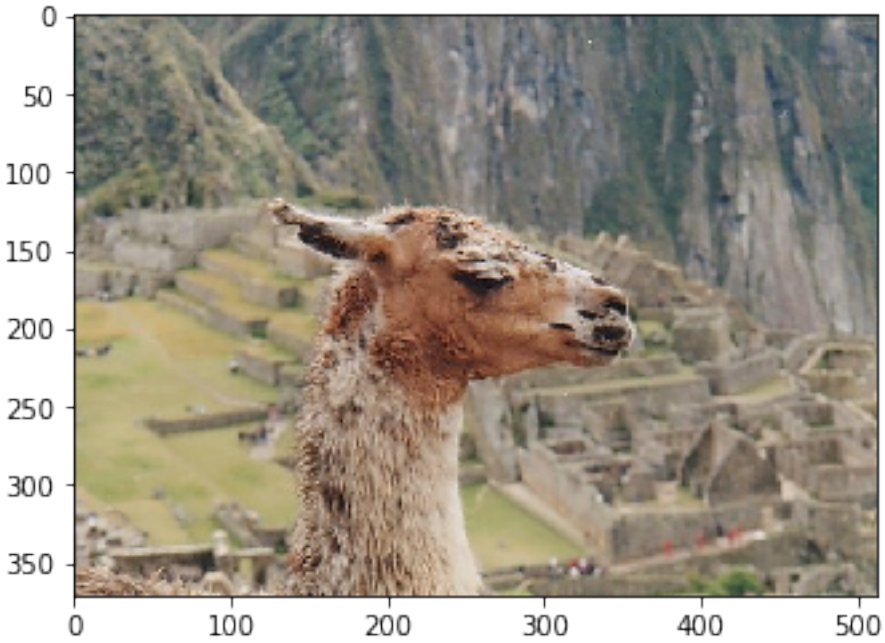


1.5 4. Varying number of components

- We use the llama picture in the dataset increase the number of components = [1,2,3,5,7]in the GMM, and observe the results ,while keeping the number of iterations constant.
- We can see that the image is best modelled by a gaussian with number of components = 5
- This will vary from image to image

[61]:
```
name = 'llama'
img = image_list(name)
out1 = grabcut(image_list(name), bbox_list(name),2, num_components = 1)
out2 = grabcut(image_list(name), bbox_list(name),2, num_components = 3)
out3 = grabcut(image_list(name), bbox_list(name),2, num_components = 5)
out4 = grabcut(image_list(name), bbox_list(name),2, num_components = 7)
plt.imshow(img)
```

[61]:
```
<matplotlib.image.AxesImage at 0x7fa195090160>
```

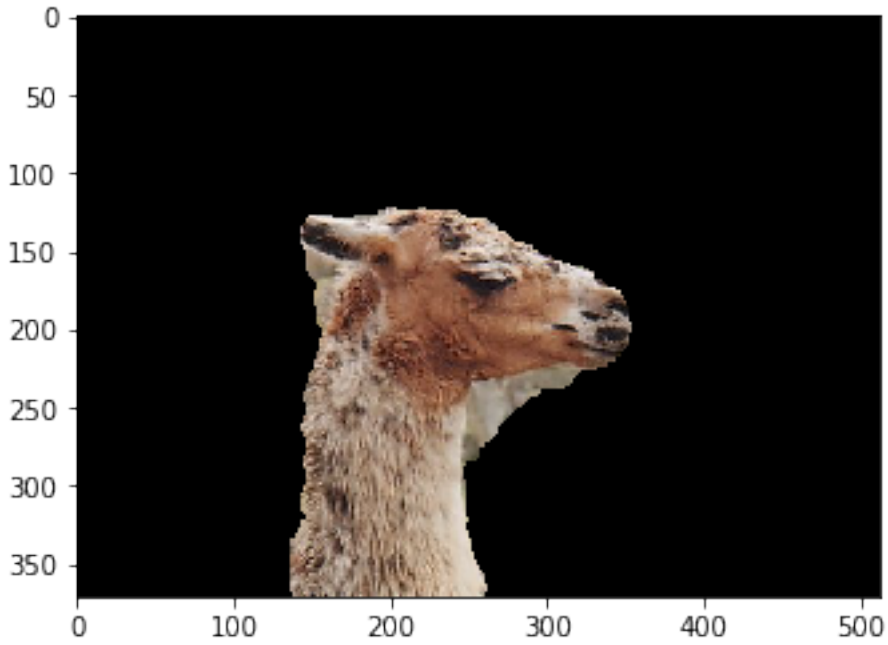


[66]:
```
out5 = grabcut(image_list(name), bbox_list(name),2, num_components = 2)
```

[62]:
```
print ("Num_components = 1")
plt.imshow(out1)
```

```
Num_components = 1
```

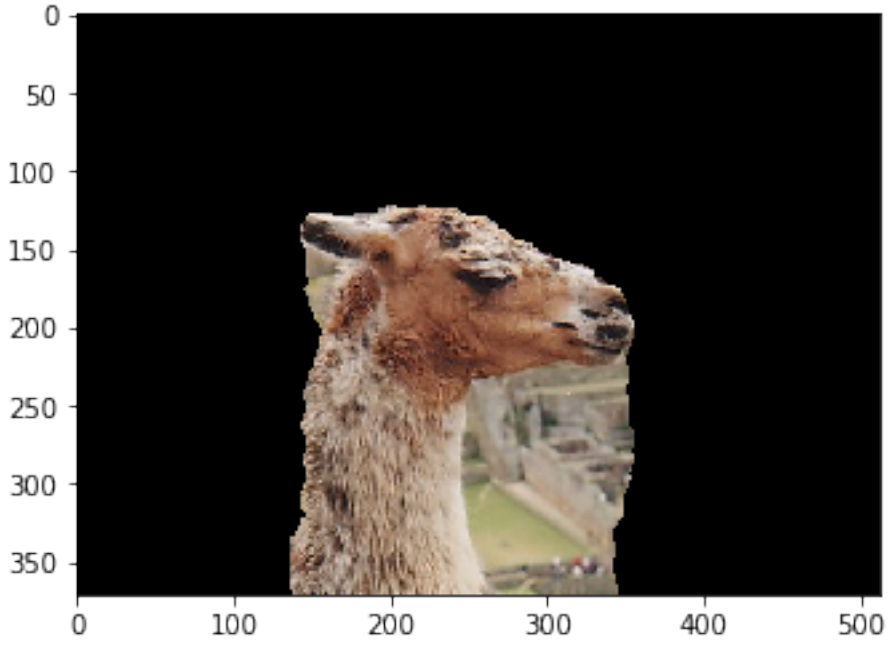
[62]:
```
<matplotlib.image.AxesImage at 0x7fa195476128>
```



[67]:
```
print ("Num_components = 2")
plt.imshow(out5)
```

```
Num_components = 2
```

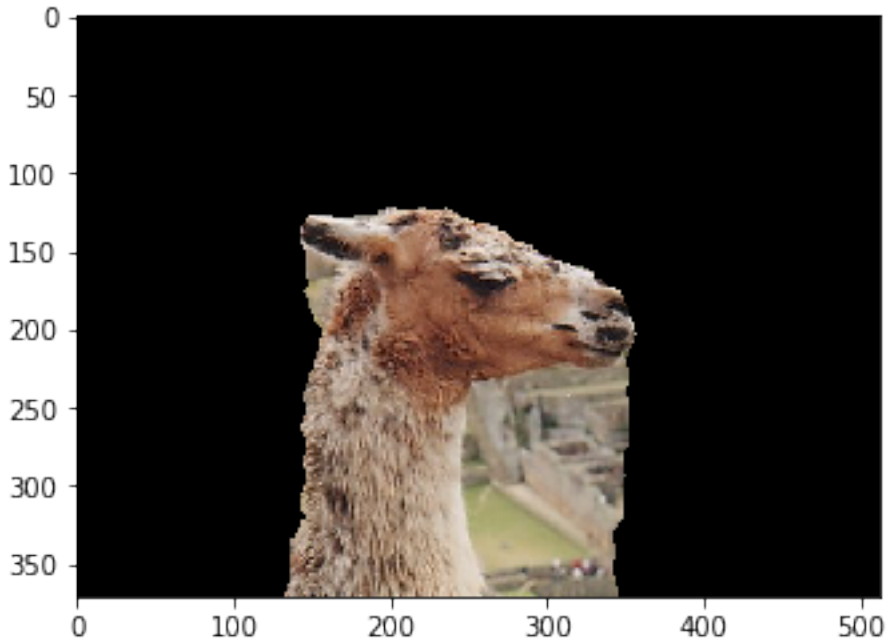
[67]: <matplotlib.image.AxesImage at 0x7fa195071fd0>



[63]:
```
print ("Num_components = 3")
plt.imshow(out2)
```

```
Num_components = 3
```

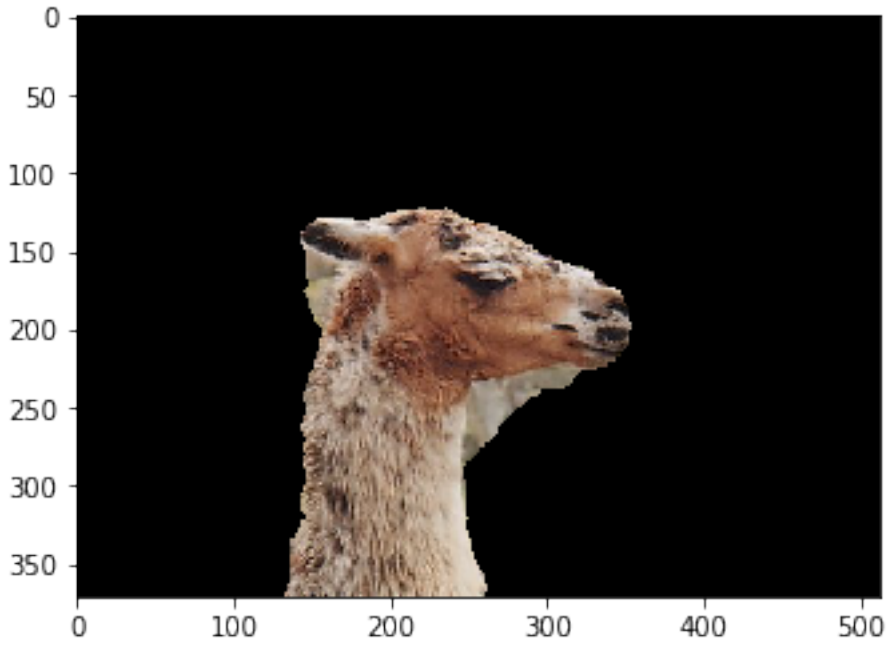
[63]: <matplotlib.image.AxesImage at 0x7fa19538fcc0>



[64]:
```
print ("Num_components = 5")
plt.imshow(out3)
```

```
Num_components = 5
```

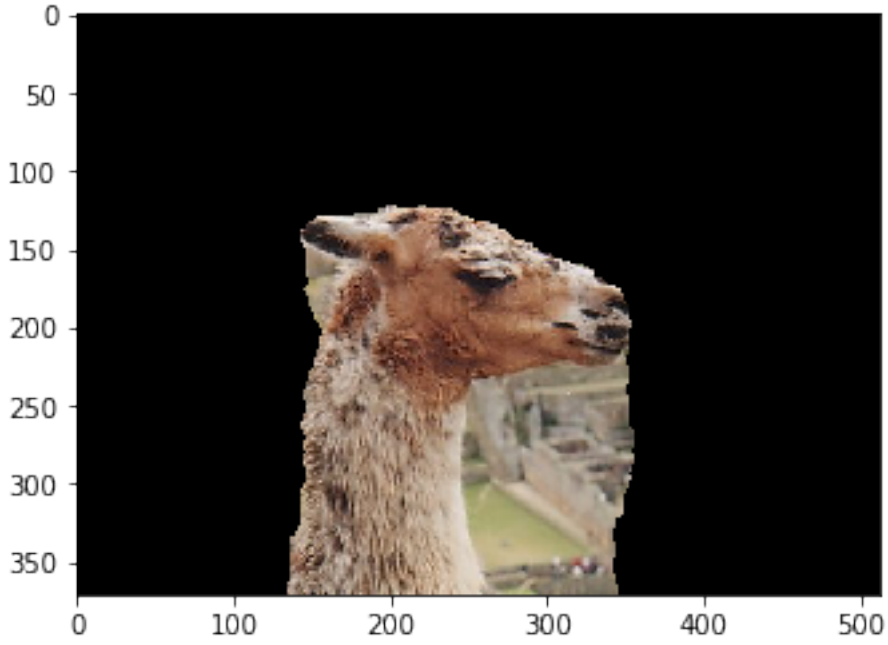
[64]: <matplotlib.image.AxesImage at 0x7fa1952f7898>



[65]:
```
print ("Num_components = 7")
plt.imshow(out4)
```

```
Num_components = 7
```

[65]: <matplotlib.image.AxesImage at 0x7fa1950dd470>

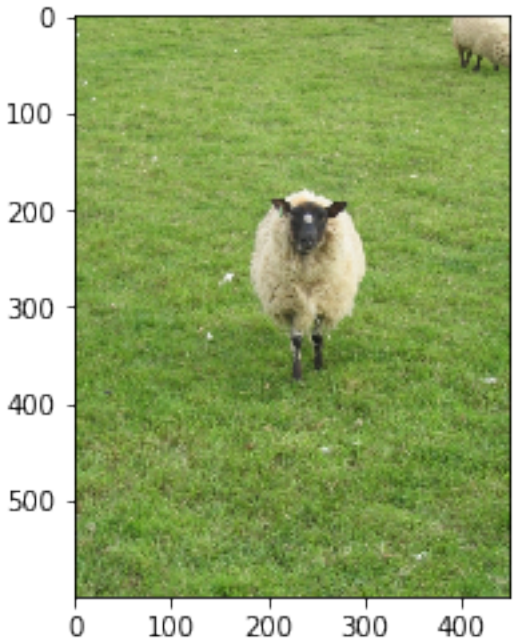


1.6 5. Varying number of neighbours

- We test with 4 neighbours and 8 neighbours and compare the difference.
- In this case, we did not observe much difference.
- Ideally, we are supposed to observe that the foreground segmentaion is much more uniform across the diagonals as we increase the number of neighbours. A different set of images would show the same

[110]:
```
name = 'sheep'
img = image_list(name)
out1 = grabcut(image_list(name), bbox_list(name),2)
out2 = grabcut(image_list(name), bbox_list(name),2, num_neighbour = 4)
plt.imshow(img)
```

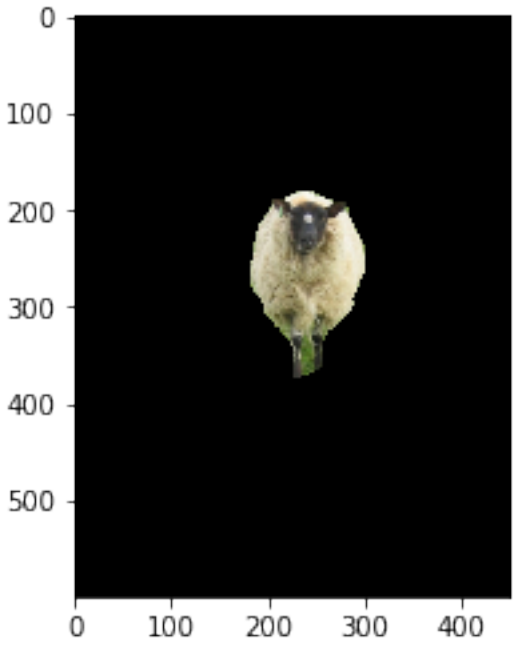
[110]: <matplotlib.image.AxesImage at 0x7fa190db50b8>



[112]:
```
print ('Number of neighbours = 8')
plt.imshow(out1)
```

```
Number of neighbours = 8
```

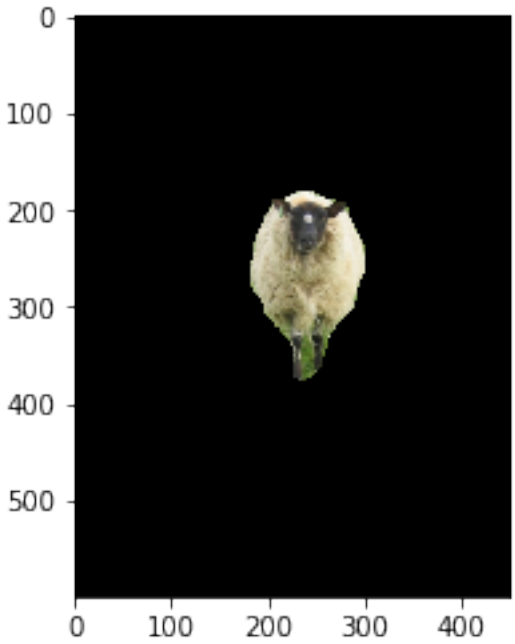
[112]: <matplotlib.image.AxesImage at 0x7fa190d65198>



[113]:
```
print ('Number of neighbours = 4')
plt.imshow(out2)
```

```
Number of neighbours = 4
```

[113]: <matplotlib.image.AxesImage at 0x7fa1922b8d30>



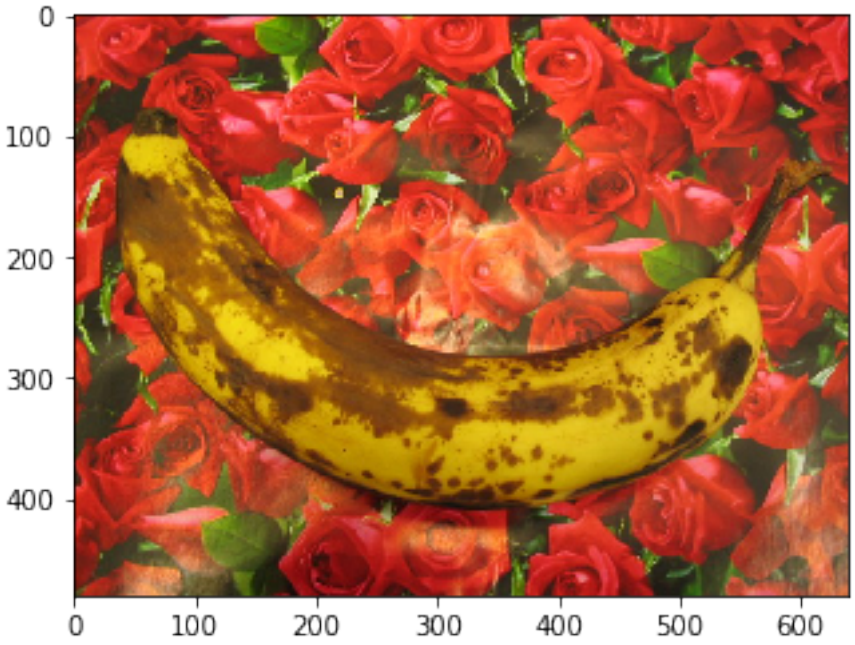
1.7 6. Variation of gamma

- Gamma is the gets multiplied with a certain value to be initialized to be source and sink depending on the case.
- Hence, when we increase gamma, there is a more stress for the neighbourhood nodes to have the same class.
- In this case, we get a better foreground segmentation.

[115]:
```
name = 'banana3'
img = image_list(name)
out1 = grabcut(image_list(name), bbox_list(name), gamma = 30)
out2 = grabcut(image_list(name),vg bbox_list(name), gamma = 40)
out3 = grabcut(image_list(name), bbox_list(name), gamma = 50)
out4 = grabcut(image_list(name), bbox_list(name), gamma = 60)
out5 = grabcut(image_list(name), bbox_list(name), gamma = 70)
print ("Original image")
plt.imshow(img)
```

```
Original image
```

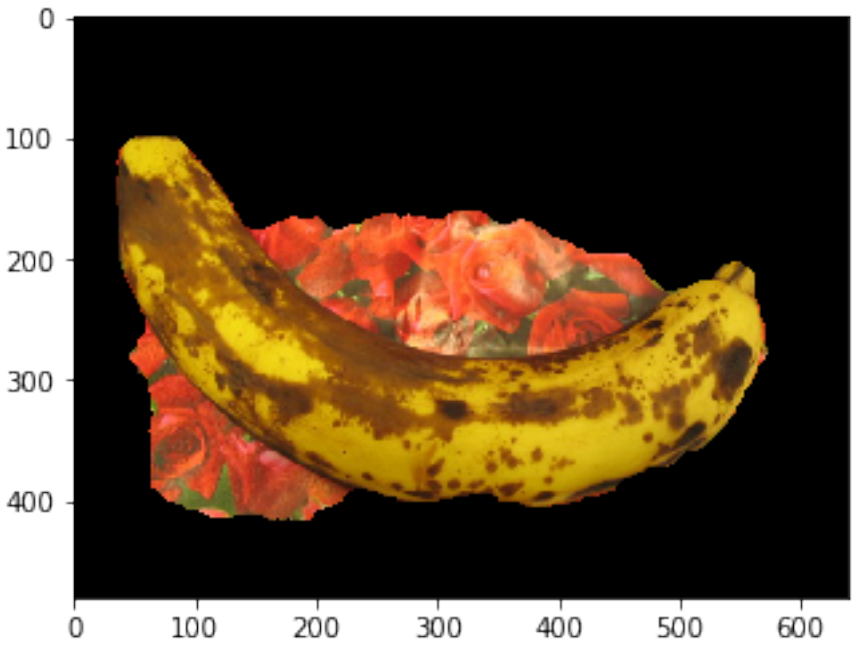
[115]:
```
<matplotlib.image.AxesImage at 0x7fa192295860>
```



[116]:
```
print ("Gamma = 30")
plt.imshow(out1)
```

```
Gamma = 30
```

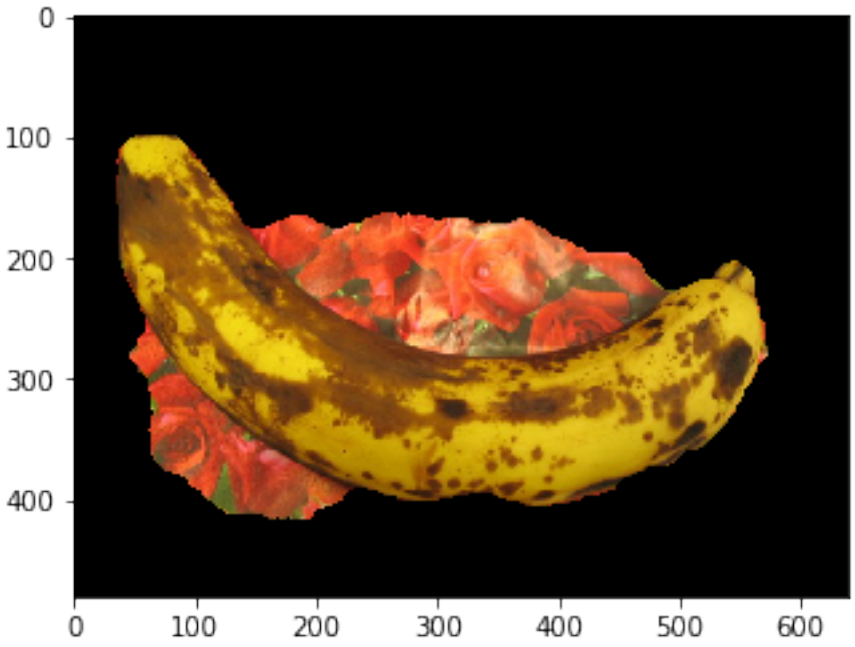
[116]: <matplotlib.image.AxesImage at 0x7fa1921ebac8>



[117]:
```
print ("Gamma = 40")
plt.imshow(out2)
```

```
Gamma = 40
```

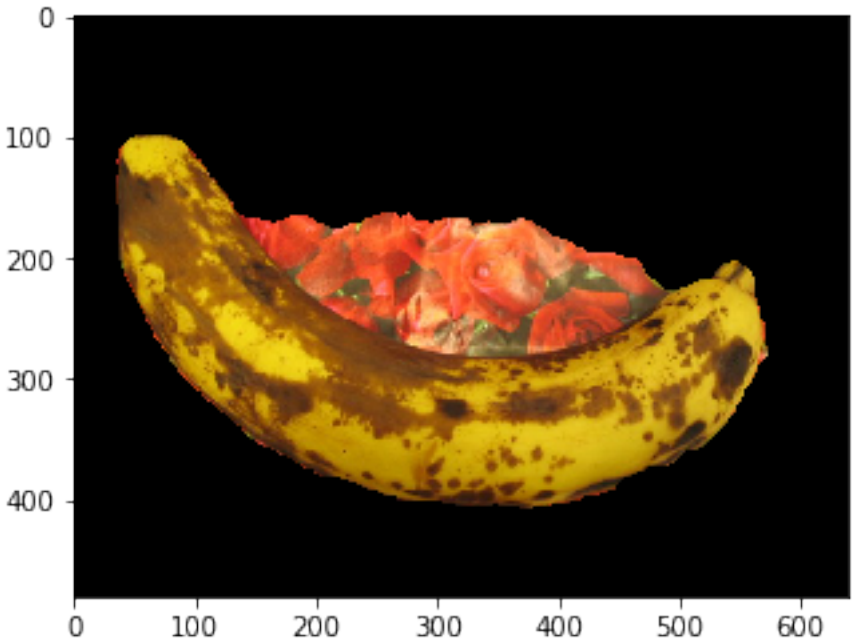
[117]: <matplotlib.image.AxesImage at 0x7fa1921c6be0>



[118]:
```
print ("Gamma = 50")
plt.imshow(out3)
```

```
Gamma = 50
```

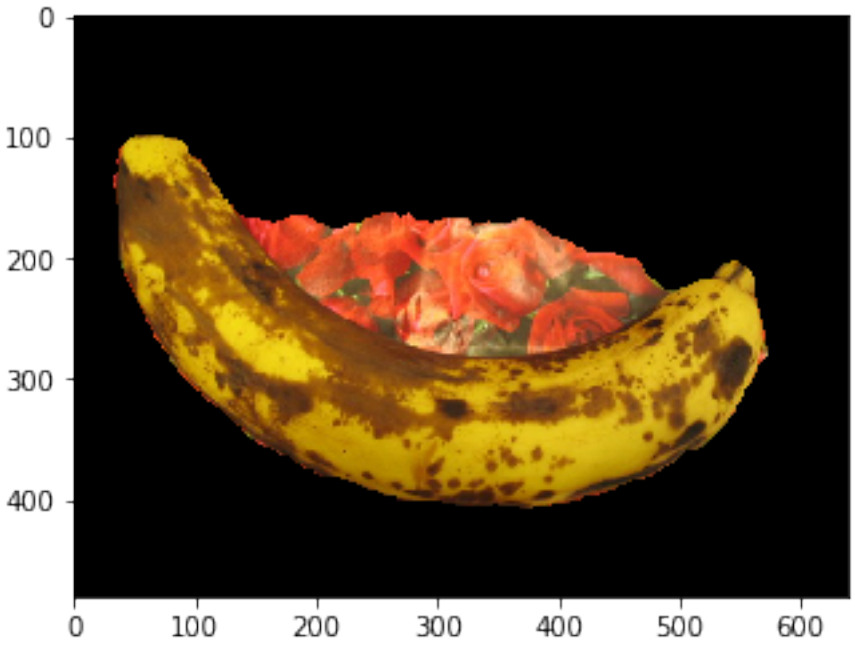
[118]: <matplotlib.image.AxesImage at 0x7fa1921a5cf8>



[119]:
```
print ("Gamma = 60")
plt.imshow(out4)
```

```
Gamma = 60
```

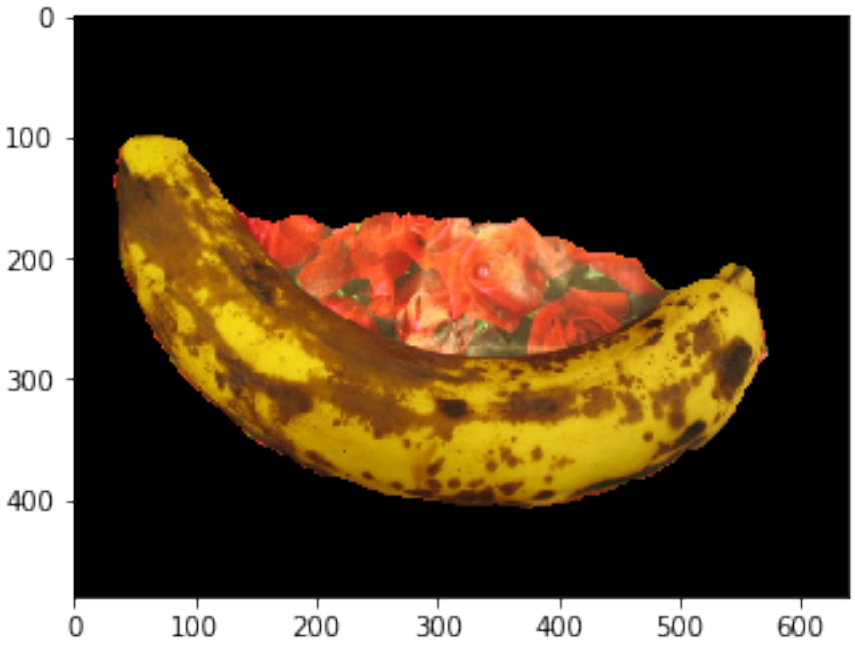
[119]: <matplotlib.image.AxesImage at 0x7fa192104e10>



[120]:
```
print ("Gamma = 70")
plt.imshow(out5)
```

```
Gamma = 70
```

[120]: <matplotlib.image.AxesImage at 0x7fa1920e4f28>



1.8 7. Varying color space

- In this particular case (banana3.jpg), better results seen for YCrCb than the rest
- In another case tested, person2.jpg, there is not much difference in the foreground segmentation.

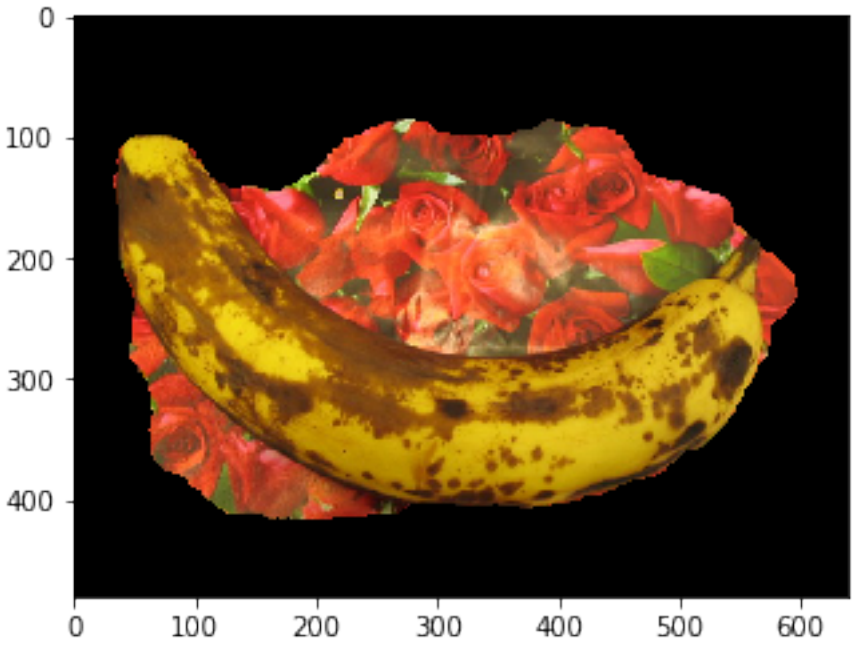
[135]:
```
name = 'banana3'
print ("Original Image")
img = image_list(name)
plt.imshow(img)
img_YCrCb = cv2.cvtColor(img, cv2.COLOR_RGB2YCrCb)
img_LAB = cv2.cvtColor(img, cv2.COLOR_RGB2LAB)
img_HSV = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)
out1 = grabcut(img, bbox_list(name),1)
out2 = grabcut(img_YCrCb, bbox_list(name),1)
out3 = grabcut(img_LAB, bbox_list(name),1)
out4 = grabcut(img_HSV, bbox_list(name),1)
```

```
Original Image
```

[136]:
```
print ("RGB Color space")
plt.imshow(out1)
```

```
RGB Color space
```

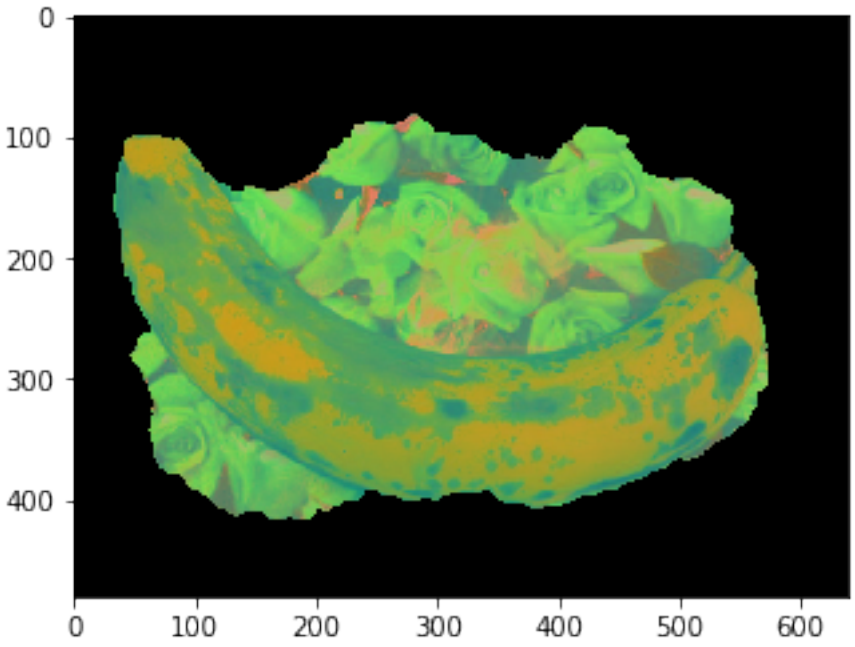
[136]: <matplotlib.image.AxesImage at 0x7fa188177eb8>



[137]:
```
print ("YCrCb colorspace")
plt.imshow(out2)
```

```
YCrCb colorspace
```

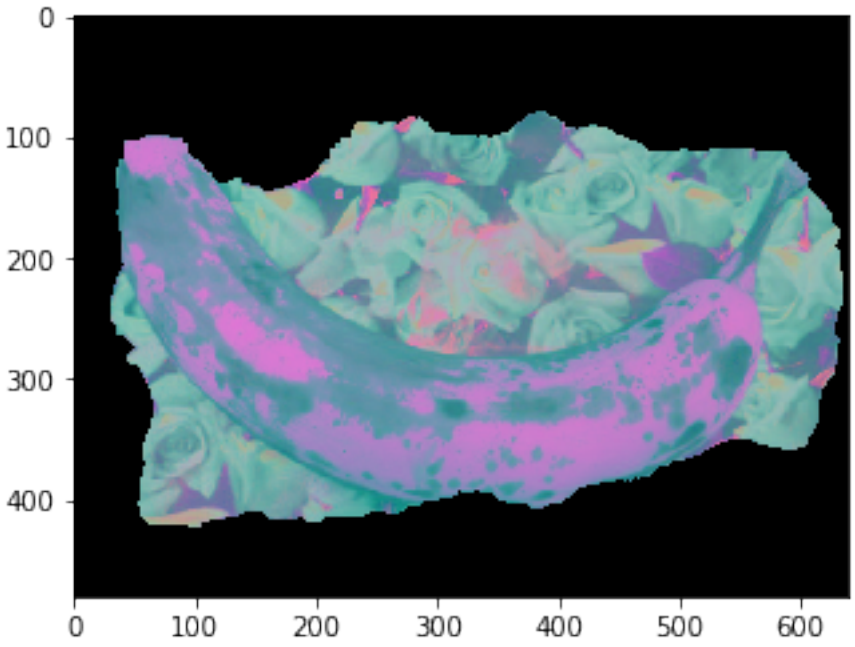
[137]: <matplotlib.image.AxesImage at 0x7fa190231fd0>



[138]:
```
print ("LAB colorspace")
plt.imshow(out3)
```

```
LAB colorspace
```

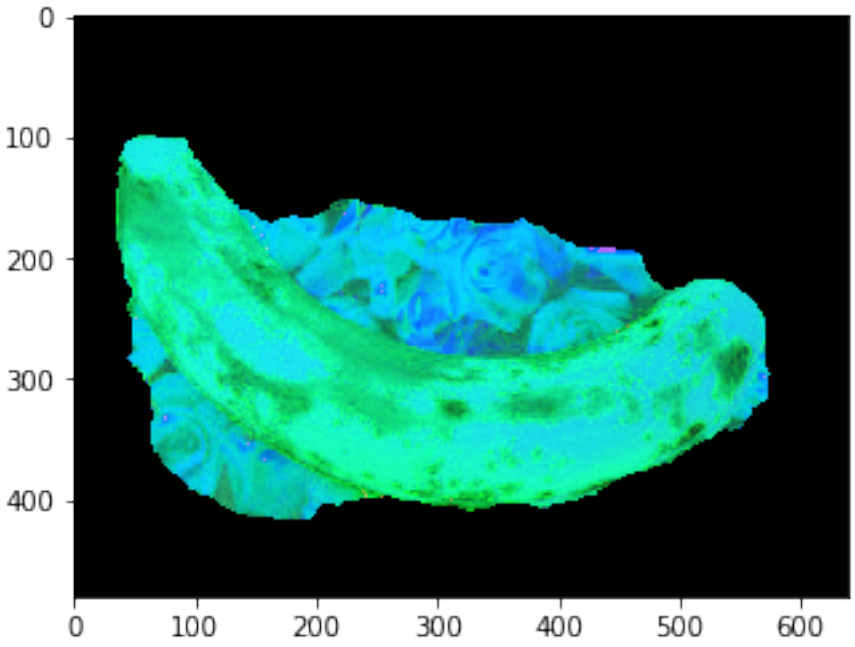
[138]: <matplotlib.image.AxesImage at 0x7fa190219128>



[139]:
```
print ("HSV colorspace")
plt.imshow(out4)
```

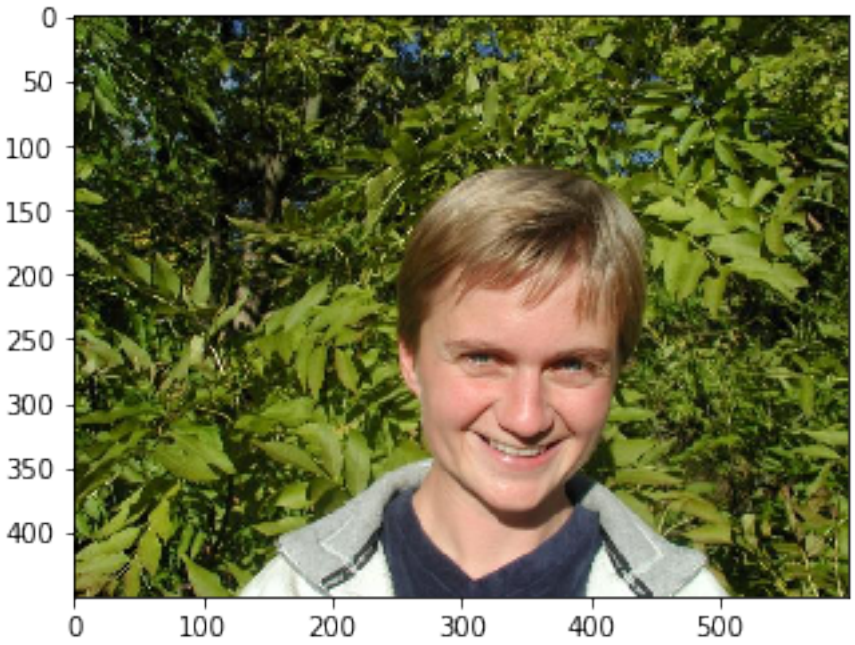
```
HSV colorspace
```

[139]: <matplotlib.image.AxesImage at 0x7fa190174240>



[140]:
```
name = 'person2'
print ("Original Image")
img = image_list(name)
plt.imshow(img)
img_YCrCb = cv2.cvtColor(img, cv2.COLOR_RGB2YCrCb)
img_LAB = cv2.cvtColor(img, cv2.COLOR_RGB2LAB)
img_HSV = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)
out1 = grabcut(img, bbox_list(name),1)
out2 = grabcut(img_YCrCb, bbox_list(name),1)
out3 = grabcut(img_LAB, bbox_list(name),1)
out4 = grabcut(img_HSV, bbox_list(name),1)
```

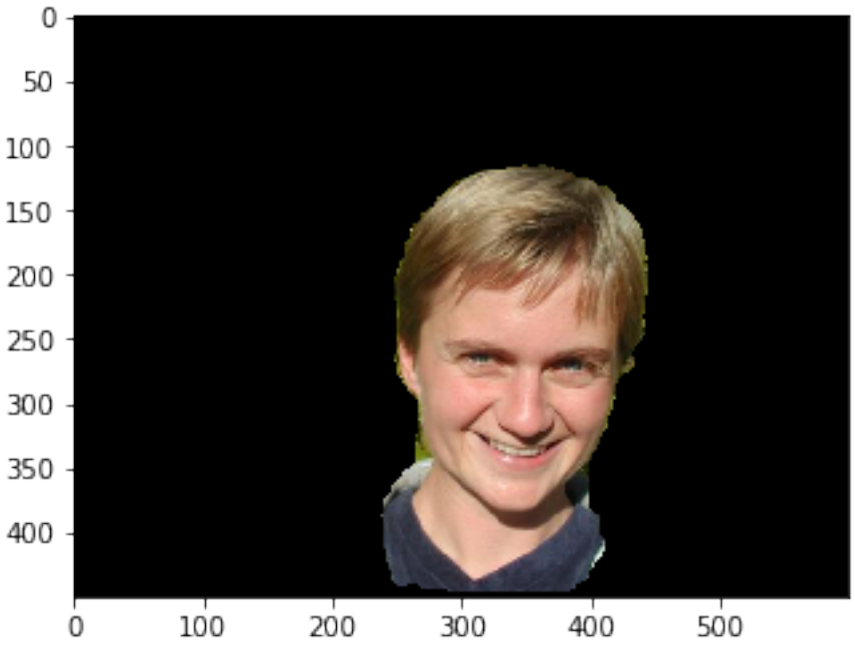
```
Original Image
```



[141]:
```
print ("RGB Color space")
plt.imshow(out1)
```

```
RGB Color space
```

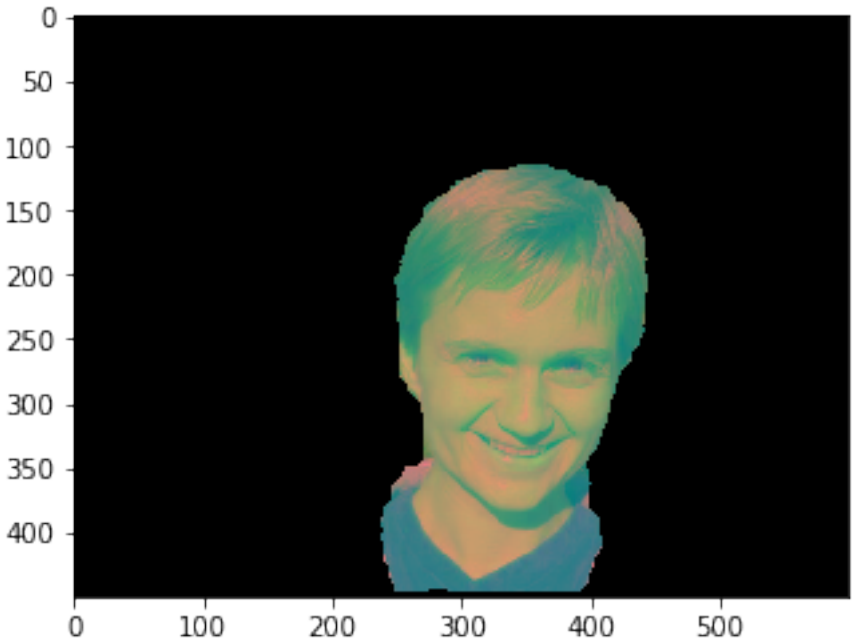
[141]:
```
<matplotlib.image.AxesImage at 0x7fa1882de438>
```



[142]:
```
print ("YCrCb colorspace")
plt.imshow(out2)
```

```
YCrCb colorspace
```

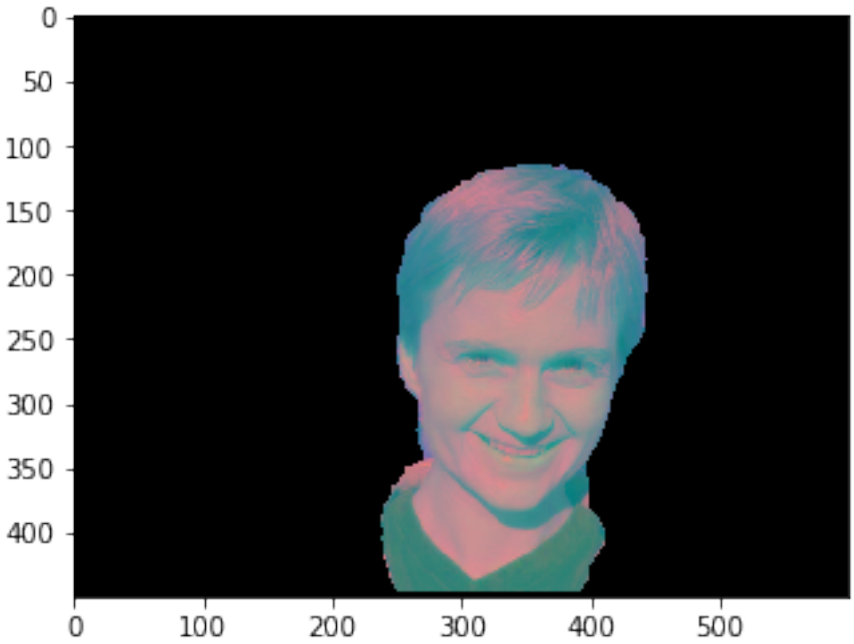
[142]: <matplotlib.image.AxesImage at 0x7fa19011e550>



[143]:
```
print ("LAB colorspace")
plt.imshow(out3)
```

```
LAB colorspace
```

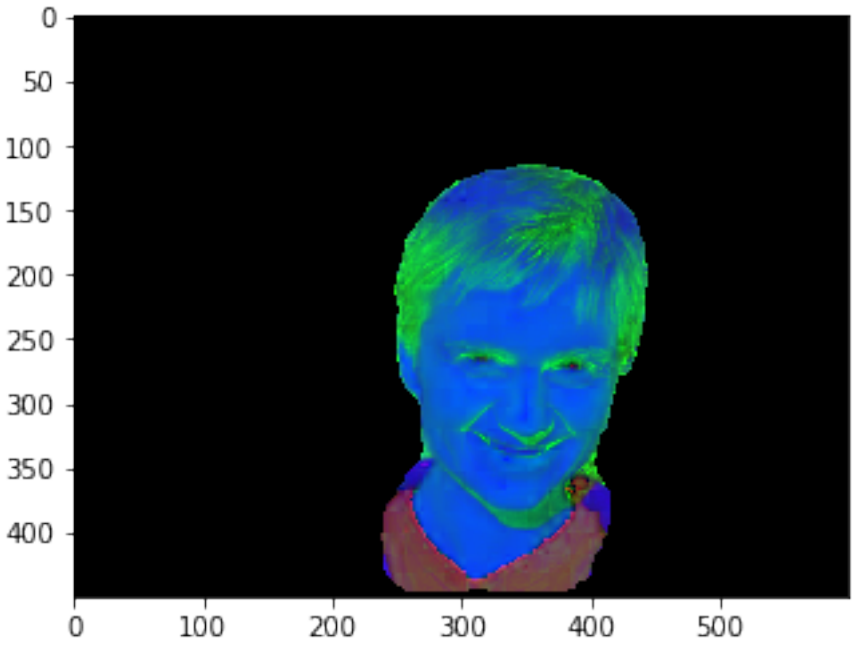
[143]: <matplotlib.image.AxesImage at 0x7fa190082668>



[144]:
```
print ("HSV colorspace")
plt.imshow(out4)
```

```
HSV colorspace
```

[144]: <matplotlib.image.AxesImage at 0x7fa18b7c1780>

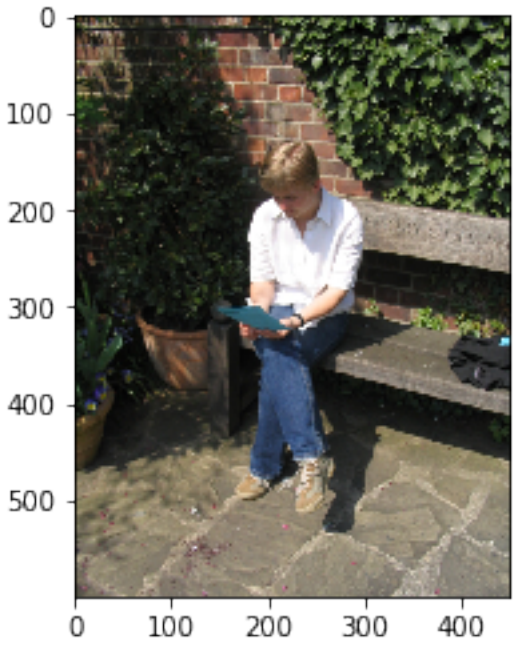


1.9 8. Varying bounding box

- Variation of tightly versus loosly bounded
- We increase the size of the bounding box rectangle on the existing and test
- We see that the tightly bounded bounding box works better
- This is due to the fact that a loose bounding box has more pixels, which gives more number of pixels to consider in the probable foreground, even though they are not in the foreground.

[171]:
```
name = 'person6'
img = image_list(name)
out = grabcut(image_list(name), bbox_list(name),3)
print ("Original")
plt.imshow(img)
out1 = grabcut(img, (110, 105, 320,545),3)
```

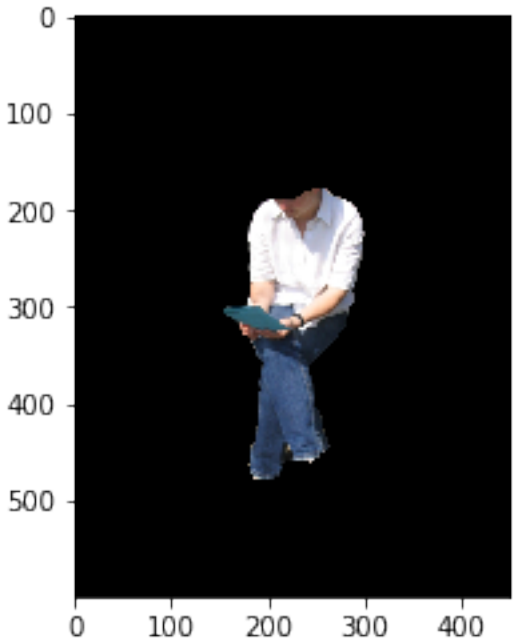
```
Original
```



[172]:
```
print ("Tight bounding box")
plt.imshow(out)
```

```
Tight bounding box
```

[172]: <matplotlib.image.AxesImage at 0x7fa17bc4f4e0>



[173]:
```
print ("Loose bounding box")
plt.imshow(out1)
```

```
Loose bounding box
```

[173]: <matplotlib.image.AxesImage at 0x7fa17bcb5080>