Assignment 1

EE P 596 A

Submitted by Mathew Varghese

10th October 2021

In [413…

```
#imports
import sys
import torch
import torchvision
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import cv2
import math

#checking if everything was imported correctly
print(sys.version)
print(torch.__version__)
print(torchvision.__version__)
print(np.__version__)
print(matplotlib.__version__)
print(cv2.__version__)
```

```
3.8.11 (default, Aug  6 2021, 09:57:55) [MSC v.1916 64 bit (AMD64)]
1.9.1+cpu
0.10.1+cpu
1.21.2
3.4.3
4.5.3
```

Q1

Q1. Image load / save. Load the `picket_fence.jpg` image provided. Print the data type of the image, and the data type of the pixels in the image. Print the image dimensions. Is the image accessed by (row,column) or (column,row)? What are the red/green/blue pixel values at (0,0)? In what order are the colors stored? What data type is the image? What data type are the pixels?

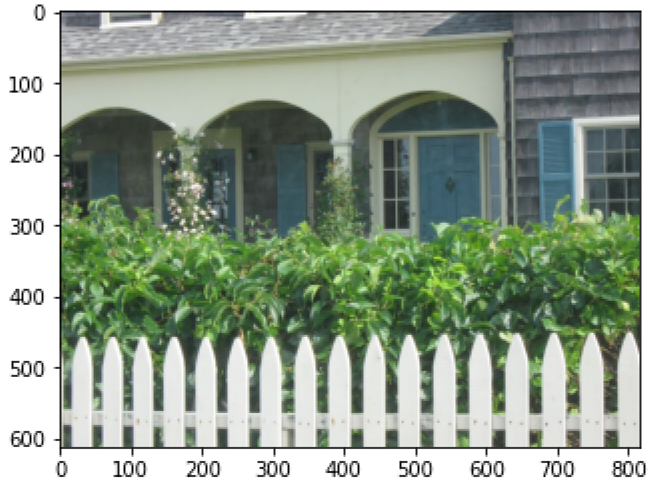
Answers

In [414…

```
image1=cv2.imread("picket_fence.jpg")  #reading the image
image1=image1[:,:,::-1] #converting RGB to BGR
plt.imshow(image1)
```

Out[414…

```
<matplotlib.image.AxesImage at 0x19857045b20>
```



In [415…

```
print("Data type of Image is",image1.dtype)
print("Data type of pixels in the image is",image1[0][0].dtype)
print("Dimensions of image are width:{},height:{},depth:{}".format(*image1.shape))
```

```
Data type of Image is uint8
Data type of pixels in the image is uint8
Dimensions of image are width:612,height:816,depth:3
```

In [416…

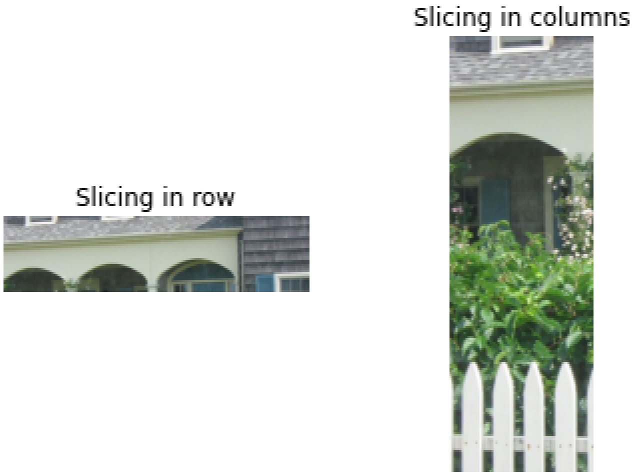
```
fig,ax=plt.subplots(1,2)

ax[0].set_title("Slicing in row")
ax[0].axis('off')
ax[0].imshow(image1[0:200,:]) #slicing rows


ax[1].set_title("Slicing in columns")
ax[1].axis('off')
ax[1].imshow(image1[:,:200]) #slicing columns
```

Out[416…

```
<matplotlib.image.AxesImage at 0x198570be4f0>
```



Slicing in row

Slicing in columns

In [417…

```
print("Image dimensions (length,width,depth) are", image1.shape)
```

```
Image dimensions (length,width,depth) are (612, 816, 3)
```

The above slicing shows that the images are accessed by row, column

In [418…

```
rgbval=image1[0][0]
print('The values at 0,0 are red:{},green:{},blue:{}'.format(*rgbval))
```

```
The values at 0,0 are red:129,green:128,blue:126
```

The colors are sorted in the order Red, Green, Blue (we swapped the image from BGR to RGB in the 1st step for the image to be used with matplotlib)

In [419…

```
print("Image data type is",image1.dtype)
print("Image pixels type is",image1[0][0].dtype)
```

```
Image data type is uint8
Image pixels type is uint8
```

Q2

Q2. Color channels. Create a new RGB image by setting the green and blue pixel values to zero everywhere. The resulting image should look red.

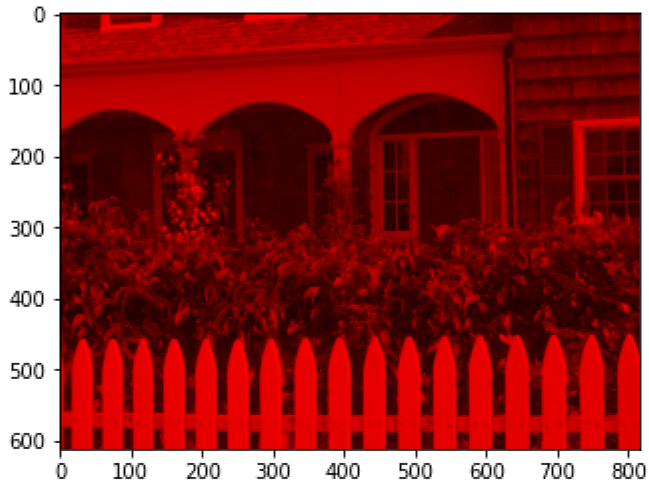
Answers

In [420…

```
red_image=image1.copy() #deep copying image for not changing the original image
red_image[:,:,1:]=red_image[:,:,1:]*0 #setting all channels except red as 0
plt.imshow(red_image)
```

Out[420…

```
<matplotlib.image.AxesImage at 0x19857134d90>
```



Q3

Q3. Photographic negative. Create a new RGB image (from the original image) by subtracting each pixel value from 255. The resulting image should look like a photographic negative.

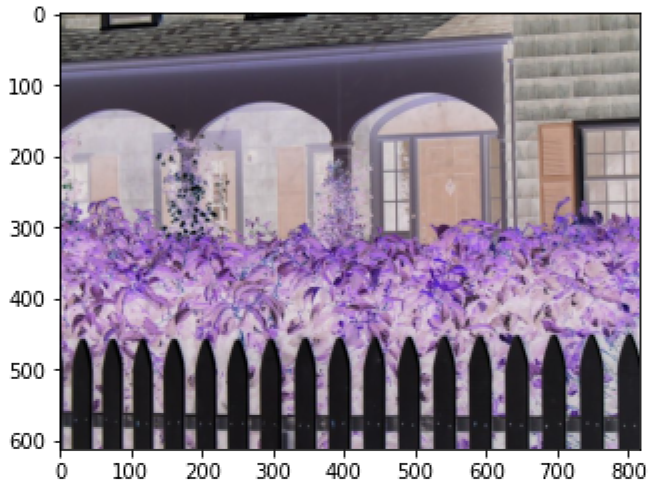
Answers

In [421…

```
negative_image=255-image1[:,:,:] #subtracting every pixel from 255 to create the negative
plt.imshow(negative_image)
```

Out[421…

```
<matplotlib.image.AxesImage at 0x1985778c7f0>
```



Q4

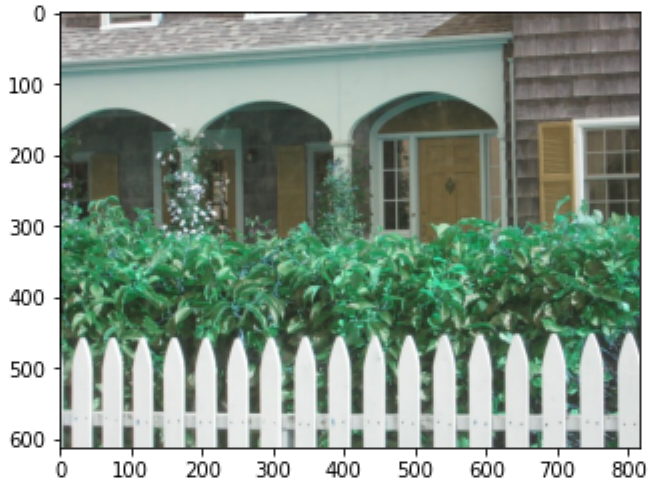
Q4. Swap color channels. Create a new RGB image (from the original image) by swapping the red and blue channels.

In [422…

```
swap_image= image1.copy()
swap_image[:,:,[0,2]]=swap_image[:,:,[2,0]]  #swapping the 1st and 3rd channel, ie the rec
plt.imshow(swap_image)
```

Out[422…

```
<matplotlib.image.AxesImage at 0x1985780e9a0>
```



Q5

Q4. Foliage detection. Create a new binary image that contains an `ON` pixel wherever the original input pixel is approximately green, and `OFF` everywhere else. (You will have to think about this a bit, but the solution is not hard.) Below is my output for comparison. What data type is the output image, and what data type are the pixels? Save the image to a `.png` file on disk; load the file in a separate image viewing program to verify that the output looks correct.

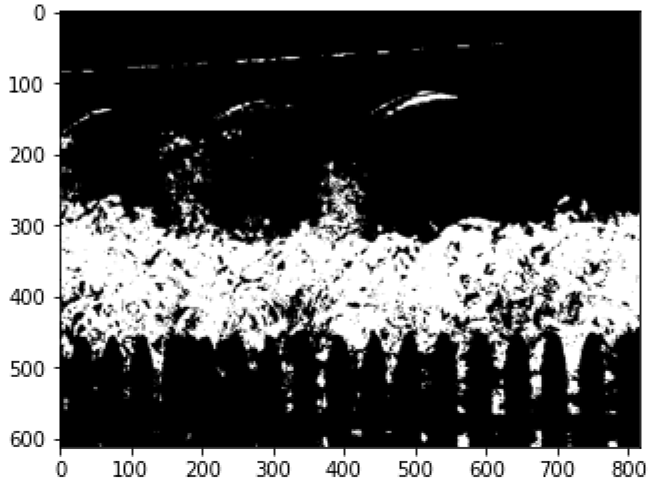
I used https://colorpicker.me/#0cf059 to pick green color

In [423…

```
hsv = cv2.cvtColor(image1, cv2.COLOR_BGR2HSV)  #converting the colorspace from RGB to HSV
mask=cv2.inRange(hsv, (50, 80, 0), (100, 255,255))  #creating a mask in the HSV colorspace
plt.imshow(mask,cmap='gray')
```

Out[423…

```
<matplotlib.image.AxesImage at 0x19859c2f760>
```



In [424…

```
cv2.imwrite('Mask.jpg',mask) #writing the detected green to a file.
```

Out[424…

```
True
```

Q6

Q6. Shift. Translate the original image to the right by 200 pixels, and down by 100 pixels. Fill in the missing values with zero.

In [425…

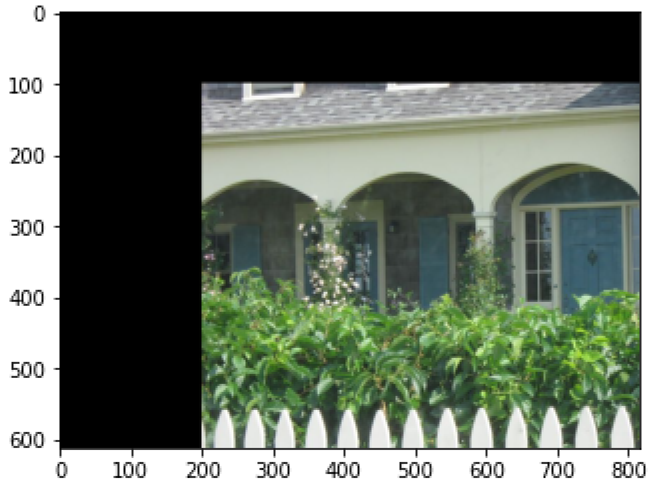
```
shift_x=200 #pixels to be shifted in x direction
shift_y=100 #pixels to be shifted in y direction
affine_translation_matrix=np.float32([
     [1,0,shift_x],
     [0,1,shift_y]
]) #creating an affine transform matrix for the shifting
```

In [426…

```
shifted_image=cv2.warpAffine(image1,affine_translation_matrix,(image1.shape[1],image1.shap
plt.imshow(shifted_image)
```

Out[426…

```
<matplotlib.image.AxesImage at 0x19859c9fa00>
```



Q7

Q7. Rotate. Rotate the original image clockwise by 90 degrees.

In [427…

```
img_center=(int(image1.shape[1]/2),int(image1.shape[0]/2)) #finding the center of the imag
rot_mat = cv2.getRotationMatrix2D( img_center,270, 1 ) #getting rotation matrix from inbui
rot_mat
```

Out[427…

```
array([[-1.8369702e-16, -1.0000000e+00,  7.1400000e+02],
       [ 1.0000000e+00, -1.8369702e-16, -1.0200000e+02]])
```

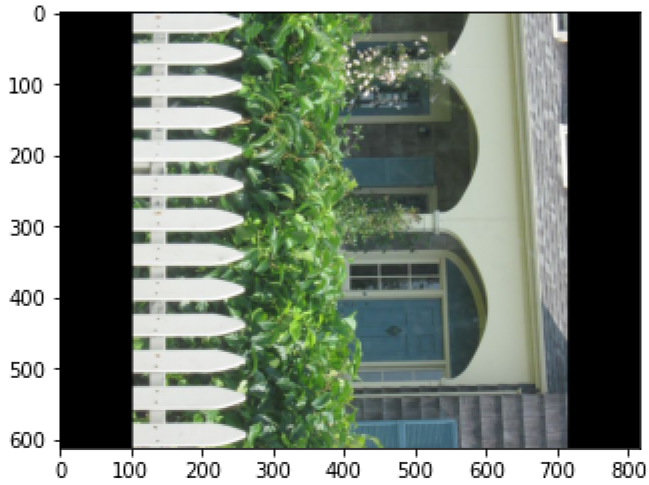
In [428…

```
rotated_image=image1.copy()
rotated_image=cv2.warpAffine(rotated_image,rot_mat,(image1.shape[1],image1.shape[0])) #usi

plt.imshow(rotated_image)
```

Out[428…

```
<matplotlib.image.AxesImage at 0x19859d0dca0>
```

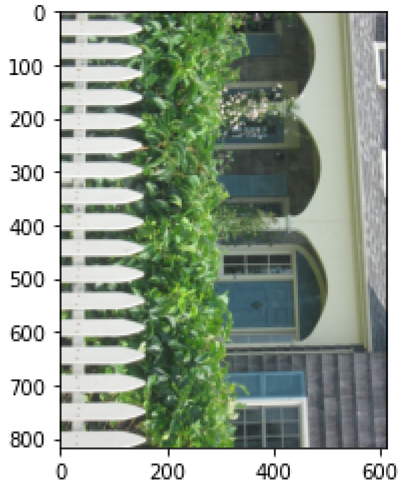


In [429…

```
rot_image=cv2.rotate(image1,cv2.ROTATE_90_CLOCKWISE)  #another method of rotating the imag
plt.imshow(rot_image)
```

Out[429…

```
<matplotlib.image.AxesImage at 0x19859d77490>
```



Q8

Q8. Similarity transform. Write a function to apply a similarity transform to an input image, yielding an output image of the same size. The function should take scale, rotation angle, and translation as input (all as floating point values). Use nearest neighbor interpolation for simplicity, but be sure to use “inverse mapping” to ensure that every pixel in the output gets painted. Test this on the original image above by passing the values scale=2.0, theta=45.0 (degrees), shift=[100,100]. (Yes, the order matters here, but the order is up to you, as is the specific interface of the function.)

In [430…

```
def similarity(image, scale=1, rot_angle=0,shift_x=0,shift_y=0):
    ''' A function for applying similarity transformations on an image.
        Takes an image, scale, rotation angle, shift in x direction and shift in y directi
        Returns the transformed image'''
    affine_translation_matrix=np.float32([
    [1,0,shift_x],
    [0,1,shift_y]
    ])    #Defining the affine matrix for simple translation
    rot_mat = cv2.getRotationMatrix2D( img_center,rot_angle, scale )  #defining affine mat
    image=cv2.warpAffine(image,affine_translation_matrix,(image.shape[1],image.shape[0]))
    image=cv2.warpAffine(image,rot_mat,(image.shape[1],image.shape[0]))  #applying rotatic
    return image
```

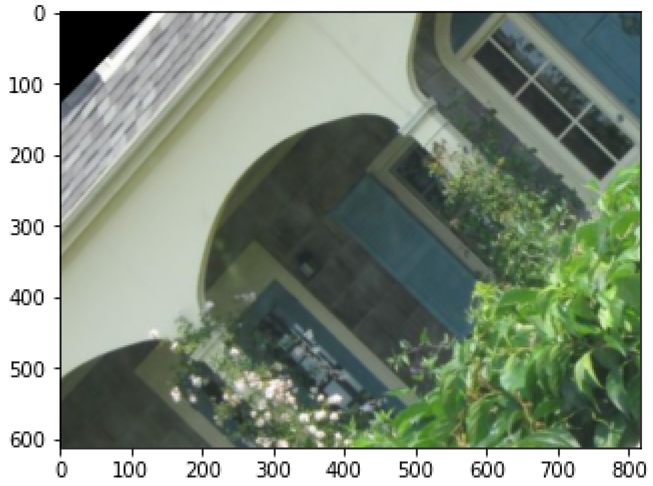
In [431…

```
    rot=float(input("Enter rotation angle: "))
scal=float(input("Enter scaling factor: "))
shift_x=float(input("Enter x shift: "))
shift_y=float(input("Enter y shift: "))
similarity_image=similarity(image=image1,scale=scal,rot_angle=rot,shift_x=shift_x,shift_y=
plt.imshow(similarity_image)
```

```
Enter rotation angle: 45
Enter scaling factor: 2
Enter x shift: 100
Enter y shift: 100
```

Out[431…

```
<matplotlib.image.AxesImage at 0x1985a9d3700>
```



Q9

Q8. Grayscale conversion. Look up the most common formula to transform RGB to grayscale. (For example, look at OpenCV's cv::COLOR_RGB2GRAY .) Apply this formula to the original image. Although the result looks reasonable, it is technically not correct. List two reasons why this common formula is wrong. (Hint: This is explained in the textbook.) To convince yourself that the common formula is wrong, apply the formula to an image of all pure blue pixels, that is, (0,0,255). (Extra credit: Implement the correct formula, assuming Rec. 709, and apply it to both the pure blue image as well as the image above.)

In [432…

```
grey_image=cv2.cvtColor(image1,cv2.COLOR_RGB2GRAY) #converting rgb to grey using the inbui
plt.imshow(grey_image,cmap="gray")
```

```
<matplotlib.image.AxesImage at 0x1985abbbdc0>
```

Out[432…

In [433…

```
r,g,b=cv2.split(image1)  #splitting the rgb channels in the image
grey_image=0.299*r+0.587*g+0.114*b  #applying the formula used in the rgb2grey function(mo
plt.imshow(grey_image,cmap="gray")
```

Out[433…

```
<matplotlib.image.AxesImage at 0x1985ac2ac10>
```

The function COLOR_RGB2GREY converts the red, green and blue colors in a specific weightage as below RGB[A] to Gray:Y←0.299⋅R+0.587⋅G+0.114⋅B

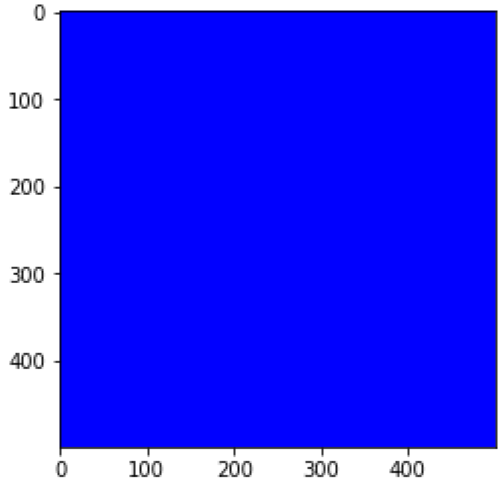
1. This is not correct as this value have been arrived as a result of an experiment involving a number of people and by taking the average of the vision sensitivities of the collection. This might be different for different people.
2. This weightage was created long before displays and is probably invalid now .Also, every display have it's own characteristics. We are not taking the display parameters into consideration when using the given weights.

In [434…

```
import numpy as np
blue_image= np.zeros([500,500,3],dtype=np.uint8)  #creating a 500x500x3 image with all zer
blue_image[:,:,2]=255   #setting the blue channel as high
plt.imshow(blue_image)
```

Out[434…

```
<matplotlib.image.AxesImage at 0x1985bc69df0>
```

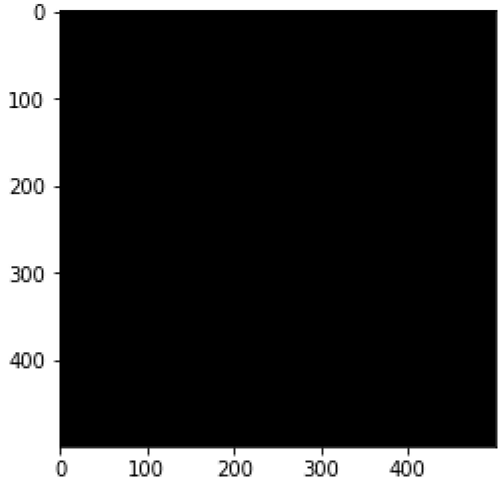


In [435…

```
r,g,b=cv2.split(blue_image) #splitting the color channels
blue_grey=0.299*r+0.587*g+0.114*b  #applying the formula used in the rgb2grey function(mos
plt.imshow(blue_grey,cmap="gray")
```

Out[435…

```
<matplotlib.image.AxesImage at 0x1985bcd03d0>
```



The greyscale image produced is all black. It proves that the color conversion formula is not accurate

Q10

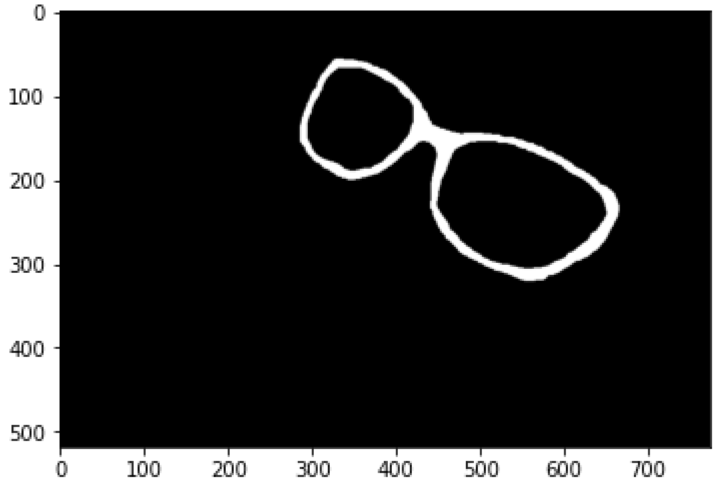
Q10. Moments. Write a function to compute the first- and second-order moments (both standard and centralized) of a binary image. Apply the function to the image `glasses_outline.png` . Print the results.

In [436…

```
image1=cv2.imread("glasses_outline.png")  #reading the image
image1=image1[:,:,::-1] #converting RGB to BGR
plt.imshow(image1)
```

Out[436…

```
<matplotlib.image.AxesImage at 0x1985bf28ca0>
```



In [437…

```
def moment(image,p=0,q=0):
    '''function to find the moments.
        Takes an image, the p and q for finding the moment
        Returns the moment'''
#     threshold=180
    moment=0
    moment_image=cv2.cvtColor(image,cv2.COLOR_RGB2GRAY) #converting to grey image for finc
    xdim=moment_image.shape[0]  #height
    ydim=moment_image.shape[1]  #width

    for x in range(0,xdim):
        for y in range(0,ydim):
                                                #commented is an optional code for thresho
#             if moment_image[x][y]>threshold:
#                 moment_image[x][y]=255
#             else:
#                 moment_image[x][y]=0
            moment=moment+x**p*y**q*moment_image[x][y]/255  #formula for calculating momen
    return moment
```

In [438…

```
#calculating all the moments using the function
#zeroth moment
m00=moment(image1,p=0,q=0)
print("0th order moment m00={}".format(m00))

#first order moments
m10=moment(image1,p=1,q=0)
m01=moment(image1,p=0,q=1)
print("first order moment m01={} and m10={}".format(m01,m10))

#second order moments
m11=moment(image1,p=1,q=1)
m20=moment(image1,p=2,q=0)
m02=moment(image1,p=0,q=2)
print("second order moments m11={} and m20={} and m02={}".format(m11,m20,m02))
```

```
0th order moment m00=11792.0
first order moment m01=5617082.0 and m10=2254903.0
second order moments m11=1131815800.0 and m20=490541389.0 and m02=2813565690.0
```

In [439…

```
# calculating 0th order centralized moments
u00=m00
print("Zeroth order centralized moment u00={}".format(u00))

# calculating 1st order centralized moments
u01=0
```

```
u10=0
print("First order centralized moment u01={} and u10={}".format(u01,u10))

# calculating 2nd order centralized moments
u11=m11-yc*m10
u20=m20-xc*m10
u02=m02-yc*m01
print("Second order centralized moments u11={} and u20={} and u02={}".format(u11,u20,u02))
```

```
Zeroth order centralized moment u00=11792.0
First order centralized moment u01=0 and u10=0
Second order centralized moments u11=57699869.44996619 and u20=59351807.97820556 and u02=1
37886399.40264606
```

Q11

Q11. Binary image processing. Using the function from the previous problem, write a function to compute the orientation and eccentricity of a binary image. Apply the function to the image `glasses_outline.png` , and draw an ellipse overlaid on the image showing the "best fitting ellipse". Explain two reasons why it is better to store `glasses_outline` as a PNG file rather than a JPEG. (Hint: This is explained in the textbook, but you can also discover the answer on your own if you save the image as a JPEG and compare the resulting files.)

In [440…

```
def orientation_eccentricity(u11,u02,u20):
    ''' Inputs 2nd order moments
        Returns orientation and eccentricity
        '''
    theta=math.atan(2*u11/(u20-u02))/2 #finding orientation angle using formula
    print("Orientation theta is :{} ".format(theta))

    eccentricity=math.sqrt(2*math.sqrt((u20-u02)**2+4*u11**2)/(u20+u02+math.sqrt((u20-u02)
    print("Eccentricity is :{}".format(eccentricity))

    return theta, eccentricity
```

In [441…

```
theta, eccentricity=orientation_eccentricity(u11,u02,u20)
```

```
Orientation theta is :-0.48662391090035356
Eccentricity is :0.910407886930469
```

In [442…

```
#finding orientation using the formula
P=np.float32([[math.cos(theta) , -math.sin(theta)],[math.sin(theta),math.cos(theta)]]) #us
P
```

Out[442…

```
array([[ 0.8839167 ,  0.46764436],
       [-0.46764436,  0.8839167 ]], dtype=float32)
```

In [443…

```
#finding the centroid using formula
xc=m10/m00
yc=m01/m00
```

In [444…

```
#finding the covarience matrix using formula

covar_matrix=1/u00*np.float32([[u20,u11],[u11,u02]])
covar_matrix
```

Out[444…

```
array([[ 5033.2266,  4893.1367],
       [ 4893.1367, 11693.216 ]], dtype=float32)
```

In [445…

```
#finding eigen vector using the formula
lamda=np.matmul(np.transpose(P),covar_matrix)
lamda=np.matmul(lamda,P)

lamda
```

Out[445…

```
array([[ 2.4444670e+03, -1.6641067e-04],
       [ 5.7038676e-05,  1.4281975e+04]], dtype=float32)
```

We can see that the off diagonal elements are almost zero

In [446…

```
max_lambda=np.amax(lamda)
min_lamda=lamda[0,0]

# finding a and b from formula
a=4*math.sqrt(max_lambda)
b=4*math.sqrt(min_lamda)
print("a:{}, b:{}".format(a,b))
```

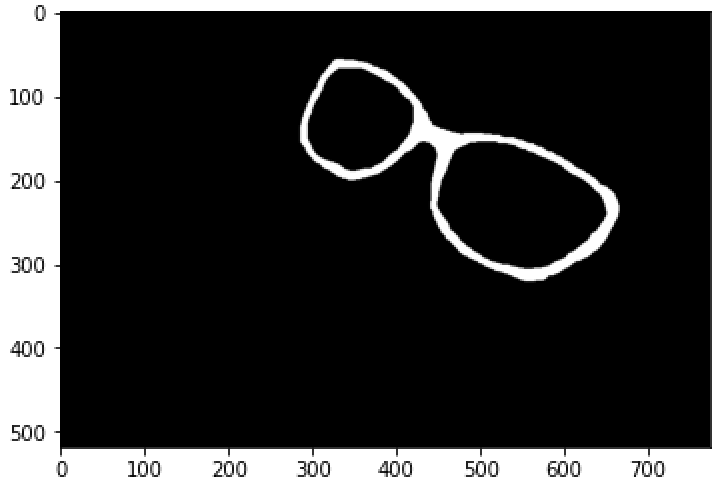
```
a:478.0288628838221, b:197.76620706341618
```

In [447…

```
image1=cv2.imread("glasses_outline.png")  #reading the image
image1=image1[:,:,::-1] #converting RGB to BGR
image1=cv2.cvtColor(image1,cv2.COLOR_RGB2GRAY)
plt.imshow(image1,cmap="gray")
```

Out[447…

```
<matplotlib.image.AxesImage at 0x1985bfa0340>
```



In [448…

```
#We are trying to fit an ellipse on the glasses. We have a,b,axis , but we need the center
# Initial approach is to find the farthest points in the width direction and height direct
# The assumption is that the farthest points of the object of interest in the image would
```

In [449…

```
smallx=255  #initializing least x as max value
largex=0    #initializing max y as max value
smally=255  #initializing least y as max value
largey=0    #initializing max y as max value
xdim=image1.shape[0]
ydim=image1.shape[1]
for x in range(0,xdim):
    for y in range(0,ydim):
        #finding the farthest points in the width and height directions
        if image1[x,y]>250:
            if x<smallx:
                smallx=x
            if x>largex:
                largex=x
            if y<smally:
                smally=y
            if y>largey:
                largey=y
center=(smally+(largey-smally)/2,smallx+(largex-smallx)/2)

center
```

Out[449…

```
(460.0, 188.5)
```

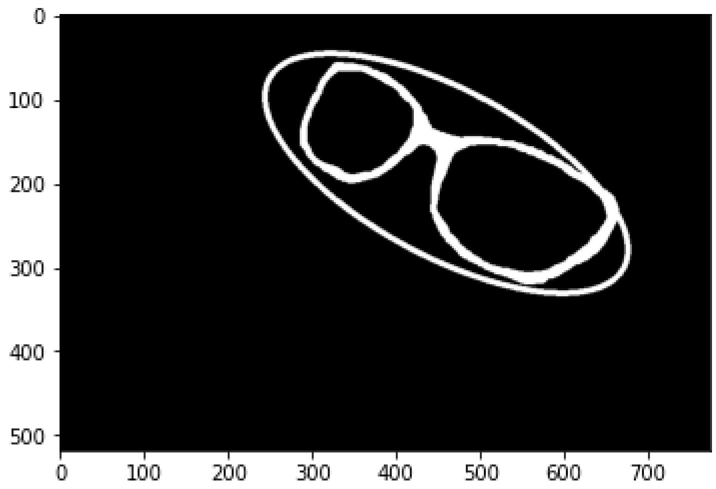
In [450…

```
color = (0, 0, 255) #setting color as blue
thickness = 5  #setting thickness
angle = theta
axesLength = (a, b)
center_coordinates = center
window_name = 'Image'

# cv2.ellipse(image1, center_coordinates, axesLength ,angle, startAngle, endAngle, color,
img=cv2.ellipse(image1, (center_coordinates, axesLength, math.degrees(-theta)), (255, 0, 2
plt.imshow(img,cmap="gray")
```

Out[450…

```
<matplotlib.image.AxesImage at 0x1985c91c430>
```



My 2nd approach is to find the contour usign the inbuilt function and getting the ellipse center using the function

In [451…

```
contours, hierarchy  = cv2.findContours(image1, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE
ellipse=cv2.fitEllipse(contours[0])
center=ellipse[0]
center
```

Out[451…

```
(460.31317138671875, 188.60821533203125)
```

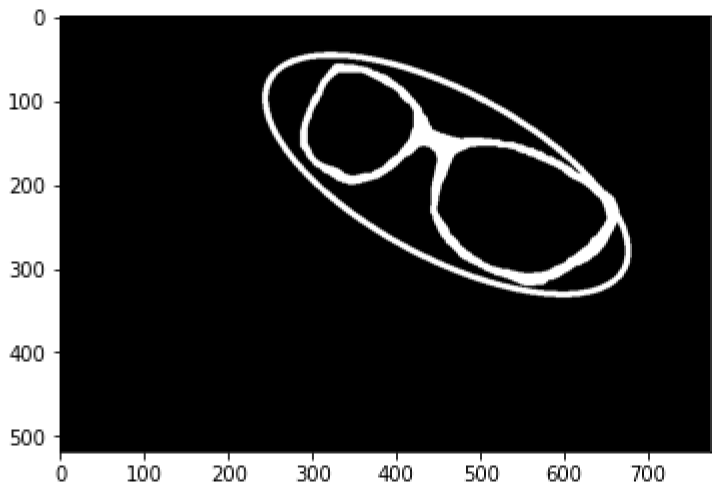
In [452…

```
color = (0, 0, 255)
thickness = 5
startAngle = 0
endAngle = 360
angle = theta
axesLength = (a, b)
center_coordinates = center
window_name = 'Image'

# cv2.ellipse(image1, center_coordinates, axesLength ,angle, startAngle, endAngle, color,
img=cv2.ellipse(image1, (center_coordinates, axesLength, math.degrees(-theta)), (255, 0, 2
plt.imshow(img,cmap="gray")
```

Out[452…

```
<matplotlib.image.AxesImage at 0x1985c97f7c0>
```



We can see that both the approaches created similar ellipses.

In [453…

```
cv2.imwrite('ellipse.png',img) #writing the ellipse overlapped image to a file.
cv2.imwrite('ellipse.jpg',img) #writing the ellipse overlapped image to a file.
```

Out[453…

```
True
```

PNG is better than JPEG because

1. PNG uses loseless compression whereas jpg uses lossy compression. That means information is lost when jpg images are stored and can create artifacts upon scaling.
2. PNG supports transparency but JPG doesnot support transparency.