CS-663 Assignment 3 Q3

Soham Naha (193079003)
Akshay Bajpai (193079002)
Mohit Agarwala (19307R004)

October 18, 2020

3 (50 points) Spatially Varying Blurring (to mimic the background-blur effect in video chats)

Input images:

- *2/data/bird.jpg*
- *2/data/flower.jpg*

If these images still lead to a computational cost that is beyond your computer's capabilities to allow for a sufficiently comprehensive experimentation, then you may downsample the images. Each input image has a visually distinct foreground and a background. In case of flower.jpg, the flower in focus at the center is the foreground and the rest is background. Similarly, in case of bird.jpg, the bird is the foreground and the rest is background. Your task is to keep the foreground intact and blur the background according to a spatially varying scheme as follows.

- (10 points) Generate a mask image M with values 1 for the foreground and 0 for the background. Design a partially/fully automatic scheme to generate this mask. Fully manual solutions will not be considered at all. Explain your algorithm in detail, and motivate and justify it. Display the (i) the original image, (ii) the mask image M , (iii) the original image with the pixels values in the foreground (as determined by M) set to black, (iv) the original image with the pixels values in the background (as determined by M) set to black.
- (20 points) Once the mask image M is generated, your task is to blur background using a spatially varying kernel. The kernel will be a circular disc function, with weights summing to 1. However, the disc radius in the kernel at every pixel location in the background will vary as a function of the pixel location as follows. Let P be any pixel location in the background, and let d P be the shortest distance from P to the foreground region in the mask image M . Set the radius r of the disc kernel to be spatially dependent as follows: r = d P if d P , and r = if d P ¿ , where := 20 for flower.jpg (at the original image size) and := 40 for bird.jpg (at the original image size). Write a function mySpatiallyVaryingKernel.m implementing this scheme. Note that you SHOULD NOT blur the foreground objects (i.e., the flower, the bird) at all.
- (10 points) For the image of spatially-varying r values, display a contour plot or a jet-colormapped image to clearly demonstrating the variation of r with respect to the distance from the foreground region in the mask M .
- (4 points) Display, as images, the blurring kernels (disc-shaped) computed at distances d P as 0.2 , 0.4 , 0.6 , 0.8 , and .
- (6 points) Display the output images, with a clearly-visible sharp foreground and blurred back- ground.

mySpatiallyVaryingKernel.py

```
import cv2
import numpy as np
import sys,os
from tqdm import tqdm
import matplotlib.pyplot as plt
from numpy import sqrt,e,power,linspace,outer,pi

```

```python

kern_size = lambda sigma: int(round(((sigma-0.8)/0.3+1)*2+1 )) if sigma >0.8 else 0
def im2double(im,d):

    im1 = im[:,:,0]
    im2 = im[:,:,1]
    im3 = im[:,:,2]
    out1 = (im1.astype(np.float64) - np.min(im1.ravel())) / (np.max(im1.ravel()) - np.min(im1.ravel()))
    out2 = (im2.astype(np.float64) - np.min(im2.ravel())) / (np.max(im2.ravel()) - np.min(im2.ravel()))
    out3 = (im3.astype(np.float64) - np.min(im3.ravel())) / (np.max(im3.ravel()) - np.min(im3.ravel()))
    out = cv2.merge((out1,out2,out3))
    return out


def meanShift(img):
    """ Perform mean shift segmentation on the input image
    inputs : image
    outputs: mean-shift segmented image and the original image resized """

    r,c,d = img.shape
    img = im2double(img,d)
    gaussian_blur = cv2.GaussianBlur(img, (5,5), 1.0)
    row,col = 128,128
    newimg = cv2.resize(img,(row,col))


    result1 = np.zeros((row,col))
    result2 = np.zeros((row,col))
    result3 = np.zeros((row,col))
    h=0.1
    sigma = 11.0
    count = 0
    window_size = 7
    padded = np.concatenate((np.concatenate((np.zeros((row,window_size,3)), newimg),axis=1),
                np.zeros((row,window_size,3))),axis=1)
    padded = np.concatenate((np.concatenate((np.zeros((window_size,col+2*(window_size),3)),padded),axis=0),
            np.zeros((window_size, col+2*(window_size),3))),axis=0)
    r,c,d = padded.shape
    #print(padded.shape)
    spatial = np.zeros((2*window_size+1,2*window_size+1))
    for is1 in range(2*window_size+1):
        for is2 in range(2*window_size+1):
            spatial[is1][is2] = ((is1-window_size)**2+(is2-window_size)**2)**0.5
    spatial = np.exp(-(spatial/sigma)**2)

    for idx1 in tqdm(range(window_size,r-window_size)):
        for idx2 in range(window_size,c-window_size):
            window = padded[idx1-window_size:idx1+window_size+1, idx2-window_size:idx2+window_size+1]
            #print(newimg[idx1][idx2])
            (x1,x2,x3) = newimg[idx1-window_size][idx2-window_size]
            N1 = 0.0 #numerator
            D1 = 1.0 #denominator
            N2 = 0.0 #numerator
            D2 = 1.0 #denominator
            N3 = 0.0 #numerator
            D3 = 1.0 #denominator


            for itern in range(30): # number of iterations
                for idx3 in range(2*window_size+1):
                    for idx4 in range(2*window_size+1):
                        (x_i1,x_i2,x_i3) = window[idx3][idx4]
                        diff1 = abs(x1-x_i1)
                        diff2 = abs(x2-x_i2)
                        diff3 = abs(x3-x_i3)
```

```
                        d1 = np.exp(-(diff1/h)**2)*spatial[idx3][idx4]
                        n1 = x_i1*d1
                        N1 += n1
                        D1 += d1

                        d2 = np.exp(-(diff2/h)**2)*spatial[idx3][idx4]
                        n2 = x_i2*d2
                        N2 += n2
                        D2 += d2

                        d3 = np.exp(-(diff3/h)**2)*spatial[idx3][idx4]
                        n3 = x_i3*d3
                        N3 += n3
                        D3 += d3

                x1 = float(N1)/D1 # in each iteration, x changes
                x2 = float(N2)/D2 # in each iteration, x changes
                x3 = float(N3)/D3 # in each iteration, x changes

            result1[idx1-window_size][idx2-window_size] = x1
            result2[idx1-window_size][idx2-window_size] = x2
            result3[idx1-window_size][idx2-window_size] = x3
    result = cv2.merge((result1,result2,result3))
    return result,newimg

def dnorm(x,mu,sigma):
    """Calculates the 1D Gaussian kernel
    inputs: x(kernel position),sigma(standard deviation of the kernel),mu(optional, default=0)
    outputs: 1D kernel
    """
    return 1 / (sqrt(2 * pi) * sigma) * e ** (-power((x - mu) / sigma, 2) / 2)

def gaussian_kernel(ksize,mu=0,sigma=1,verbose=False):
    # create the 1-D gaussian kernel
    """Calculates and returns 2D the Gaussian Kernel
    inputs: ksize(Gaussian Kernel Size),sigma(standard deviation)
    outputs: kernel_2D (2D gaussian kernel)
    """
    if ksize%2==0:
        ksize = ksize+1

    kernel_1D = linspace(-(ksize // 2), ksize // 2, ksize)
    for i in range(ksize):
        kernel_1D[i] = dnorm(kernel_1D[i], mu, sigma)
    kernel_2D = outer(kernel_1D.T, kernel_1D.T)
    kernel_2D *= 1.0 / np.sum(kernel_2D)
    return kernel_2D



def convolution(image, kernel_distances):

    """Performs Convolution of the image with a Gaussian kernel
    inputs: image(input image),kernel_size(Gaussian kernel size),sigma(Gaussian Kernel Std. deviation)
    outputs: output (convoluted image with the kernel)
    """

    image_row, image_col,dim = image.shape
    #print(image_row,image_col)

    output = np.zeros_like(image)

    plt.figure()
    plt.imshow(image/np.max(image))
```

```python
    max_kernel_size = 0
    max_kernel = 0
    padded_row,padded_col,_ = image.shape
    #print(image.shape)

    for row in tqdm(range(padded_row)):
        for col in range(padded_col):

            min_truncate = min(min(row,padded_row-row-1),min(col,padded_col-col-1))
            kernel_size = min(min_truncate,kern_size(kernel_distances[row,col,0]))

            if kernel_size!=0:
                kernel = gaussian_kernel(kernel_size+2,sigma=kern_size(kernel_distances[row,col,0]))

                if kernel_distances[row,col,0]>max_kernel_size:
                    max_kernel_size = kernel_distances[row,col,0]
                    max_kernel = kernel

                #print(row,col,kernel.shape,row+kernel.shape[0])
                for i in range(3):
                    output[row,col,i] = np.sum(kernel *image[row-kernel.shape[0]//2:row+kernel.shape[0]//2+1,
                        col-kernel.shape[1]//2:col + kernel.shape[1]//2+1,i])
            else:
                for i in range(3):
                    output[row,col,i] = image[row,col,i]


    return output,max_kernel

def mask(result2,resized_orig,threshold,distance_thresh):
    """Perform masking operation on the input image after relabling the pixel values close to each
    other to a same value (similar to K means clustering) .
    Inputs : Mean shift segmented image, original image,threshold for mask, threshold for K means
    Outputs : Mask,Masked image, original image after masking
              (All three for both foreground black and white cases)

    """
    masked = np.zeros_like(result2)
    unique_intensities ={}
    result = (result2).astype(np.float32)
    hsv_result = cv2.cvtColor(result,cv2.COLOR_RGB2HSV)
    h,s,v = cv2.split(hsv_result)
    r,c = v.shape
    diff = np.zeros_like(v)

    for i in tqdm(range(r)):
        for j in range(c):
            inten = v[i,j]
            flag=0
            if inten not in unique_intensities:
                diff = 999999
                intensity = 0
                for k in unique_intensities:
                    if abs(inten-k)<distance_thresh:
                        flag=1
                        if diff>abs(inten-k):
                            diff = abs(inten-k)
                            intensity=k
                        v[i,j] = intensity
                if flag==0:
                    unique_intensities[inten] = 0
                else:
                    unique_intensities[intensity] += 1
            else:
                unique_intensities[inten] += 1
```

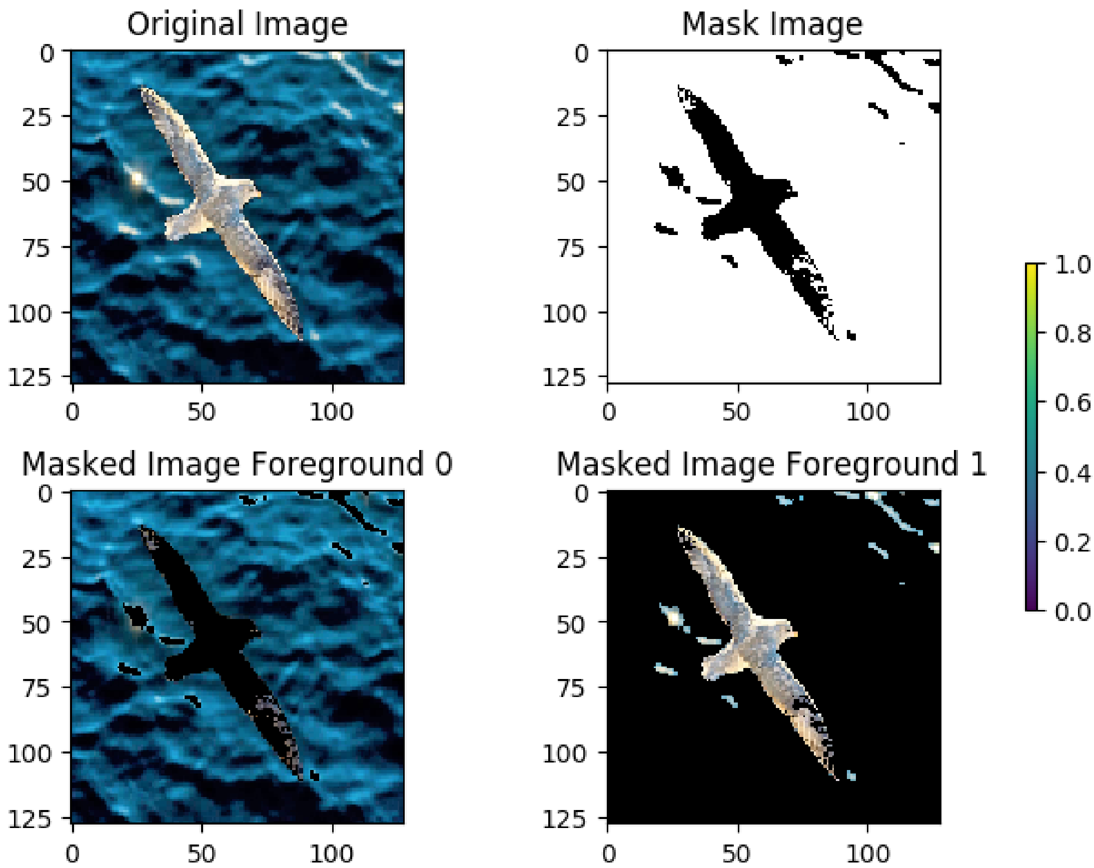
```python

    sort =sorted(unique_intensities.items(), key = lambda kv:(kv[1], kv[0]))
    #print(sort)
    h_copy = np.ones_like(h)
    s_copy = np.ones_like(s)
    v_copy = v.copy()
    for i in tqdm(range(r)):
        for j in range(c):
            if v_copy[i,j]==sort[-1][0]:
                v_copy[i,j] = 0
            else:
                v_copy[i,j] = 1
    masked = cv2.merge((h,s,v))
    masked_2 = cv2.merge((h,s,v_copy))
    masked = cv2.cvtColor(masked,cv2.COLOR_HSV2RGB)
    masked_2 = cv2.cvtColor(masked_2,cv2.COLOR_HSV2RGB)

    vectorized_mask1,img1,foreground_pixels1=masked_image(masked_2,resized_orig,0,threshold)
    vectorized_mask2,img2,foreground_pixels2=masked_image(masked_2,resized_orig,1,threshold)

    return vectorized_mask1,vectorized_mask2,img1,img2,foreground_pixels1,foreground_pixels2

def masked_image(masked_2,resized_orig,flag,threshold):

    """Called from mask function to perform threshold based mask generation on the
        K means clustered image
    Inputs : K means clustered iamge, original image, flag(for foreground=0 or 1) , threshold
    Outpus : Mask,Masked image, orignal image after masking
    """
    r,c,d = masked_2.shape
    vectorized_mask = np.zeros((r,c))
    foreground_pixels = []
    for i in range(r):
        for j in range(c):
                vectorized_mask[i,j] = masked_2[i,j,0] * masked_2[i,j,1] * masked_2[i,j,2]
                if vectorized_mask[i,j]>threshold:   #0.02 works for flower , 0.4 for bird
                    if flag==0:
                        vectorized_mask[i,j]=0
                    else:
                        vectorized_mask[i,j]=1
                    foreground_pixels.append((i,j))
                else:
                    if flag==0:
                        vectorized_mask[i,j]=1
                    else:
                        vectorized_mask[i,j]=0


    masked_img = np.zeros((resized_orig.shape[0],resized_orig.shape[1],3))
    masked_img[:,:,0] = resized_orig[:,:,0]*vectorized_mask
    masked_img[:,:,1] = resized_orig[:,:,1]*vectorized_mask
    masked_img[:,:,2] = resized_orig[:,:,2]*vectorized_mask
    masked_img = masked_img/np.max(masked_img)
    return vectorized_mask,masked_img,foreground_pixels

def plot_save(resized_image,vectorized_mask1,masked_img1,masked_img2,filename):
    fig,axes = plt.subplots(2,2, constrained_layout=True)
    axes[0][0].imshow(resized_image)
    axes[0][0].axis("on")
    axes[0][0].set_title("Original Image")
    axes[0][1].imshow(vectorized_mask1,cmap='gray')
    axes[0][1].axis("on")
    axes[0][1].set_title("Mask Image")
    #cbar = fig.colorbar(im,ax=axes.ravel().tolist(),shrink=0.45)
    axes[1][0].imshow(masked_img1)
```

```
    axes[1][0].axis("on")
    axes[1][0].set_title("Masked Image Foreground 0")
    im = axes[1][1].imshow(masked_img2)
    axes[1][1].axis("on")
    axes[1][1].set_title("Masked Image Foreground 1")
    cbar = fig.colorbar(im,ax=axes.ravel().tolist(),shrink=0.45)

    plt.savefig("../images/"+filename+"Mask_blur.png",bbox_inches="tight",pad=-1)


def all_functions(filename,threshold,alpha,distance_thresh):
    """
    Main function of the program. Called to run all other functions and finally save the outputs

    Inputs : Filepath,threshold(for mask) , alpha , distance_thresh(for k means)
    Outputs : all the outputs of the progrma saved here
    """
    #filename = sys.argv[1]
    #threshold = float(sys.argv[2])
    #alpha = float(sys.argv[3])
    #distance_thresh = float(sys.argv[4])

    img = cv2.imread(filename) # Read image here
    img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
    result2,resized_image = meanShift(img)
    filename = filename.split("/")[-1].split(".")[0]

    vectorized_mask1,vectorized_mask2,masked_img1,masked_img2,foreground_pixels1,foreground_pixels2 =
            mask(result2,resized_image,threshold,distance_thresh)  #0.4 for bird, 0.02 for flower

    plot_save(resized_image,vectorized_mask1,masked_img1,masked_img2,filename+"_"+str(alpha)+"_")
    #name = filename.rstrip("\n").split("/")[-1].split(".")[0]

    r,c,d = masked_img1.shape

    masked_image_distance = np.zeros_like(masked_img1)
    masked_image_kern_sizes = np.ones_like(masked_img1)*5

    kern_size = lambda sigma: int(round(((sigma-0.8)/0.3+1)*2+1 )) if sigma >0.8 else 0

    alpha_orig = alpha
    for factor in [0.2,0.4,0.6,0.8,1]:
        alpha = float(factor*alpha_orig)
        for i in tqdm(range(r)):
            for j in range(c):
                distance_min = 9999999
                for k in foreground_pixels1:
                    if ((i-k[0])**2+(j-k[1])**2)<distance_min**2:
                        distance_min = np.sqrt((i-k[0])**2+(j-k[1])**2)
                        masked_image_distance[i,j,:] = distance_min

                if distance_min>=alpha:
                    masked_image_distance[i,j,:] = alpha


        gaussian_blurred,max_kern = convolution(resized_image,masked_image_distance)

        plt.imsave("../images/"+filename+"_"+str(alpha)+"_"+str(factor)+"kernel.png",max_kern,
                    cmap='gray')

        fig,axes = plt.subplots(1,2, constrained_layout=True)
        axes[0].imshow(resized_image)
        axes[0].axis("on")
        axes[0].set_title("Original Image")
        im = axes[1].imshow(gaussian_blurred)
```

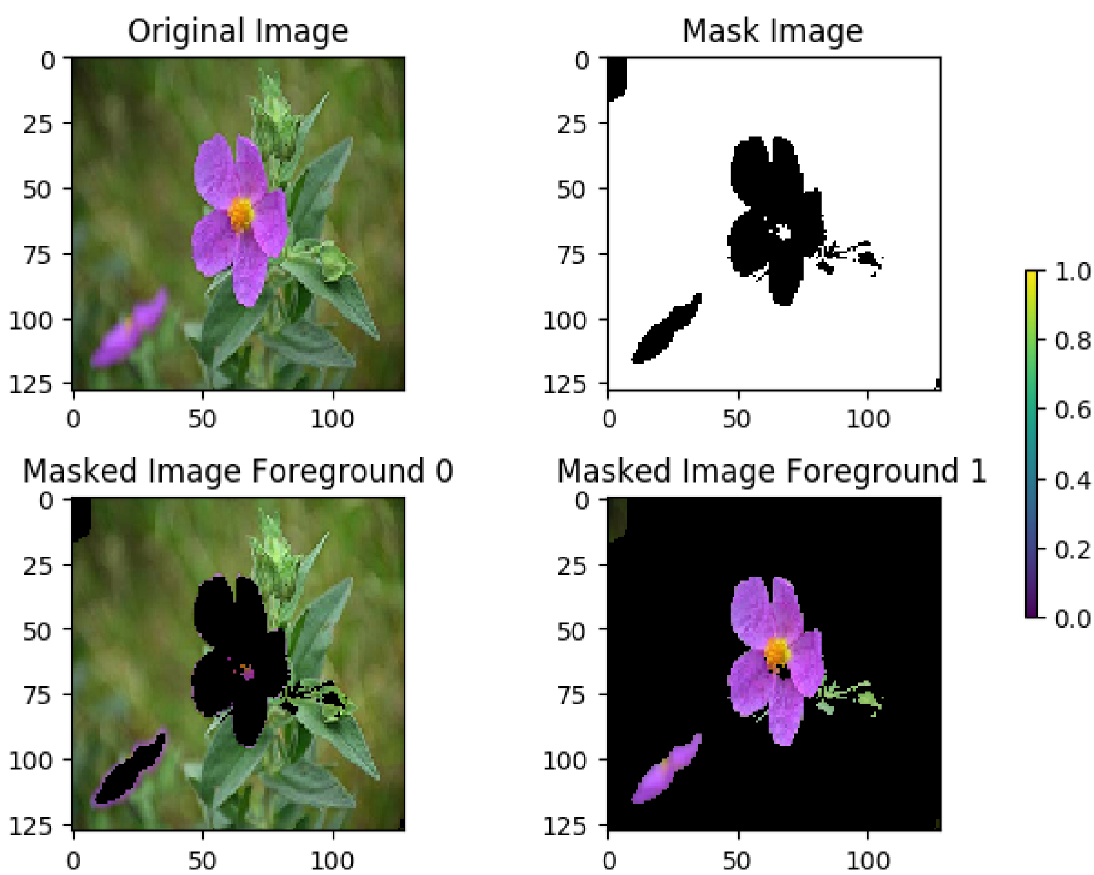
```
        axes[1].axis("on")
        axes[1].set_title("Blurred Image alpha="+str(alpha))
        cbar = fig.colorbar(im,ax=axes.ravel().tolist(),shrink=0.45)

        plt.savefig("../images/"+filename+"_"+str(alpha)+"_"+str(factor)+"_background_blur.png",
                    bbox_inches="tight",pad=-1)

        plt.figure()
        plt.imshow(masked_image_distance[:,:,0],cmap='jet')
        plt.axis("on")
        plt.title("Variation with distance alpha="+str(alpha))
        plt.colorbar()
        plt.savefig("../images/"+filename+"_"+str(alpha)+"_"+str(factor)+"_distance.png",
                    bbox_inches="tight",pad=-1)
```

myMainScript.py

```
from mySpatiallyVaryingKernel import all_functions

filenames = ["bird.jpg","flower.jpg"]
foldername = "../data/"


#for i in range(len(filenames)):
all_functions("../data/flower.jpg",0.02,20.0,0.40)
all_functions("../data/bird.jpg",0.4,5.0,0.35)
```



[a]



[b]

Figure 1: (a)Original,Mask,Foreground and Background for bird.png (b)Original,Mask,Foreground and Background for flower.png

Algorithm :
Motivation :

- To mask out the foreground from a given image, we first need to segment the image in such a way that, foreground pixels and background pixels belong to different segments.
- This will work good if, the intensities of foreground and background pixels are sufficiently different from each other.
- Thus , the first step logically should be performing mean shift segmentation on the given image.

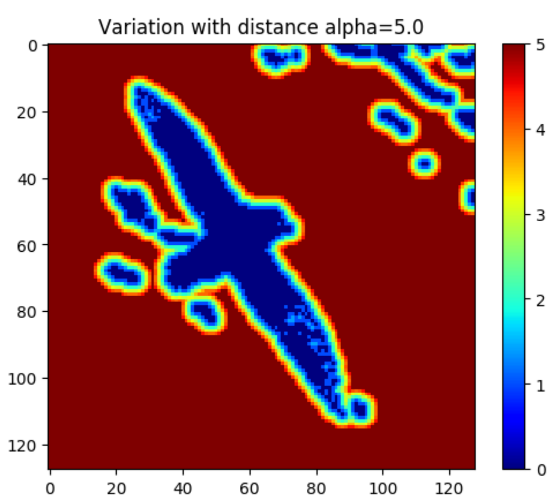
- Post mean shift, thresholding should work, to mask out the foreground region from the image.
- Since the image is in 3 channel space, it would work better if we can convert this to a single channel space.

[***Note :*** **For all the images, we resized them to 128x128 RGB images for computational conveniences. And accordingly the value of alpha was scaled down as it was given in accordance to the original image size**]

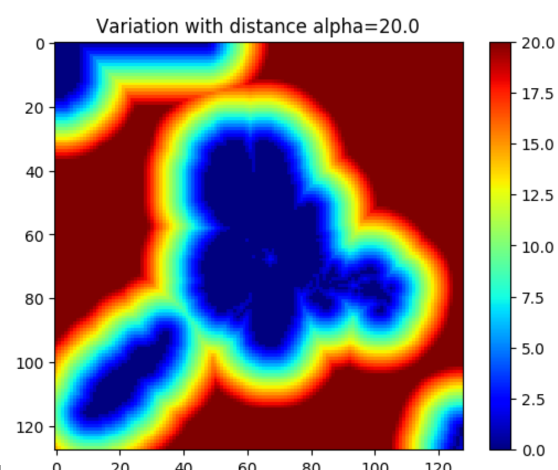
For all the images, we perform mean shift segmentation with the same code as in question 2 . We chose a common parameter of 0.1 as the standard deviation for the intensity Gaussian kernel, i.e. in case of the weighted sum.
For all the images we used the standard deviation of the spatial kernel to be 11.0.
Other parameters are :

- *2/data/baboonColor.png* :: num_iterations : 30
- *2/data/bird.jpg* :: num_iterations : 40
- *2/data/flower.jpg* :: num_iterations : 40

- Next we take the result of the mean shift and and relabel the pixel values which are close to each other upto a very small threshold , with the same values.(This is similar K means clustering) . *Note: This step is performed on V channel after converting the image into HSV*
- Now we have the image segmented perfectly into a fixed number of segments , which is converted back to RGB space.
-
  - To bring the image from 3 channels to a single channel, we perform pixel wise multiplication for each channel and get a single channel output image.
  - Then, we use a threshold to generate a masked foreground and background image
- Instead of using the conventional K means clustering we tried to implement a similar form of K means ourselves to get the nearly same output.



[a]



[b]

Figure 2: (a) Variation for bird.png (b)Variation for flower.png

Kernel for Bird.png varying alpha :

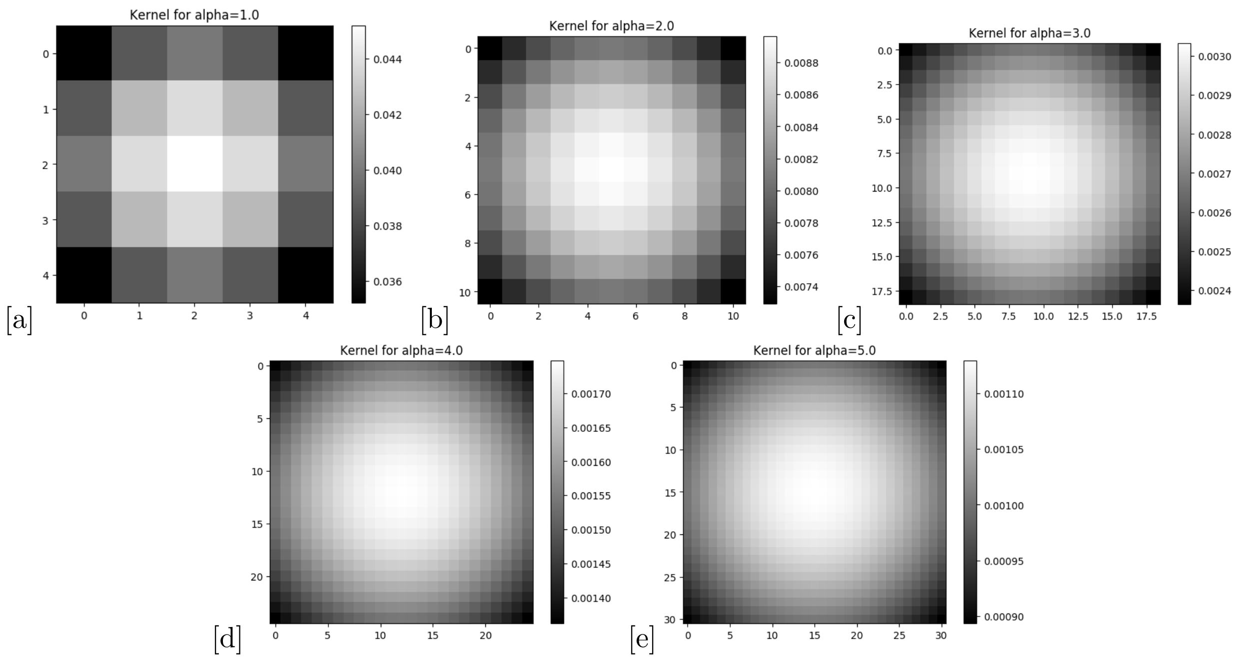


Figure 3: (a) 0.2 alpha bird.png (b) 0.4 alpha bird.png (c) 0.6 alpha bird.png(d) 0.8 alpha bird.png(e) 1 alpha bird.png

Kernel for flower.png varying alpha :

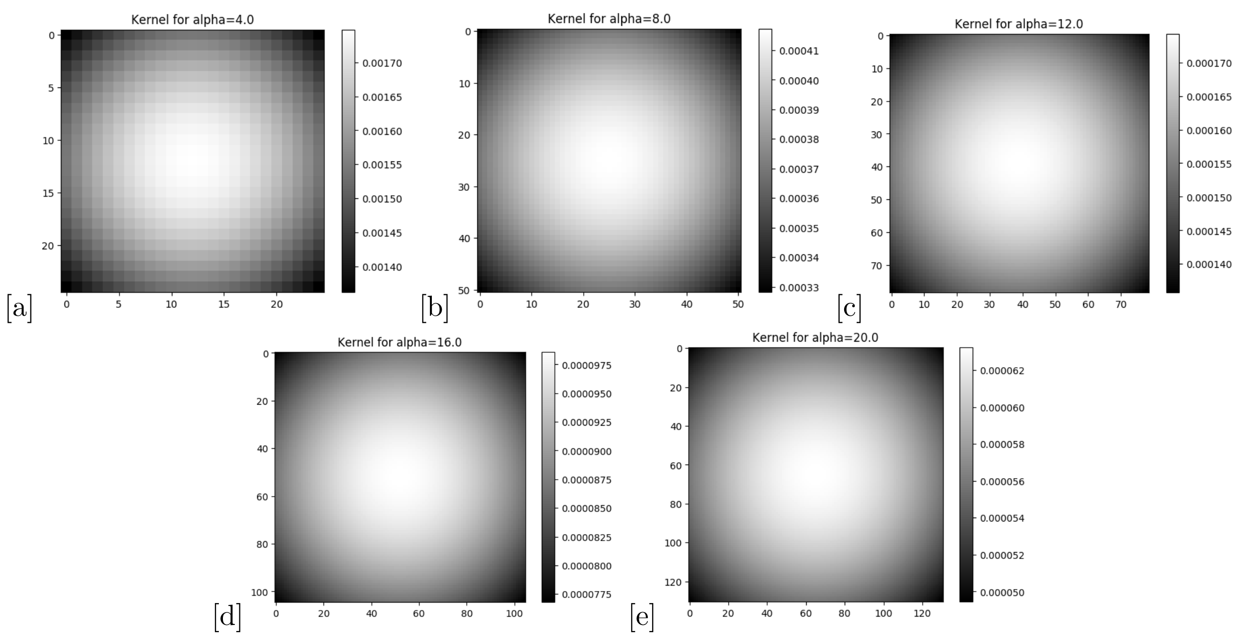


Figure 4: (a) Original and blurred bird.png (b)Original and blurred flower.png (c) Original and blurred bird.png (d)Original and blurred flower.png (e)Original and blurred flower.png

Variation of radius r with distance for different values of alpha for Bird.png :

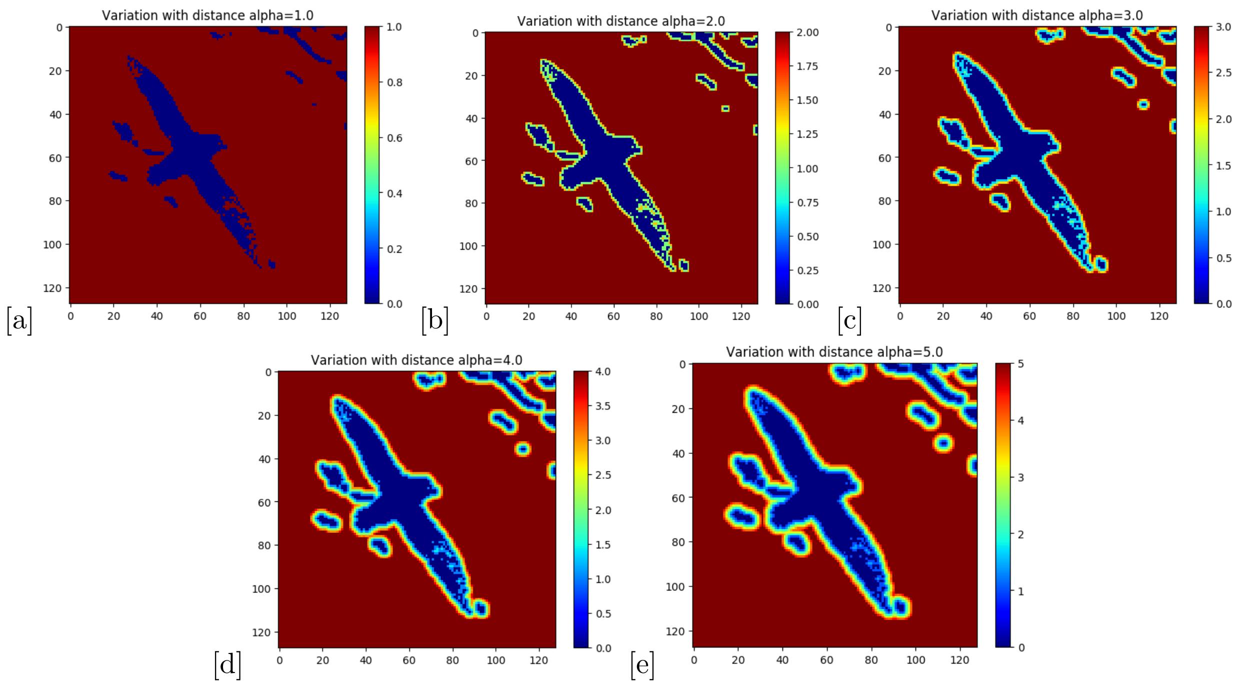


Figure 5: (a) 0.2 alpha (b) 0.4 alpha (c) 0.6 alpha (d) 0.8 alpha (e) alpha

Variation of radius r with distance for different values of alpha for Flower.png :

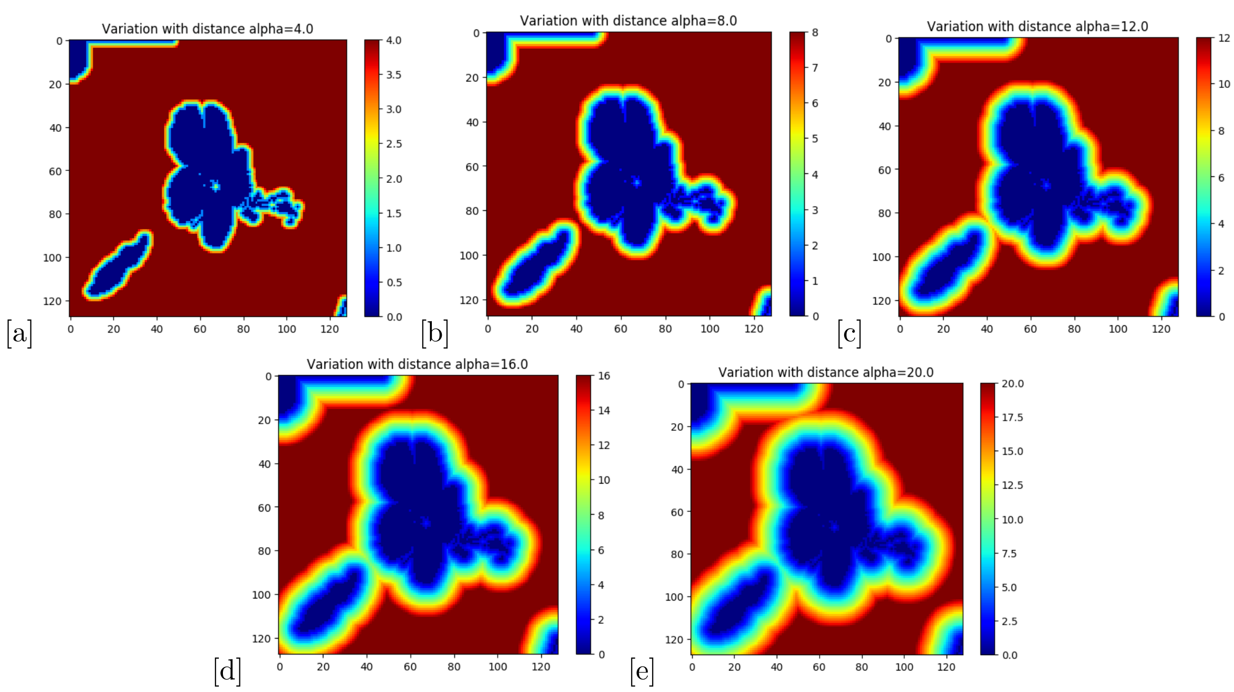
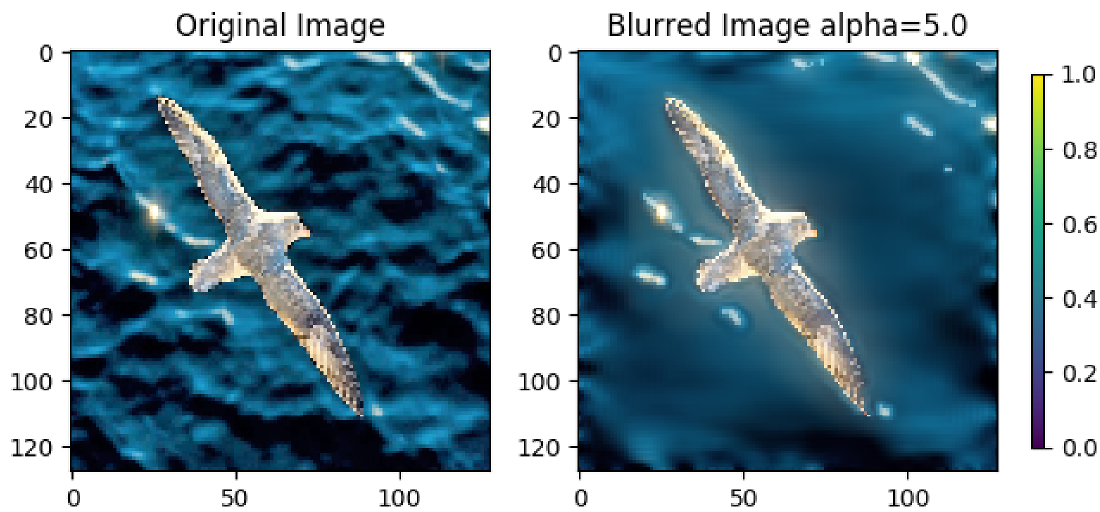
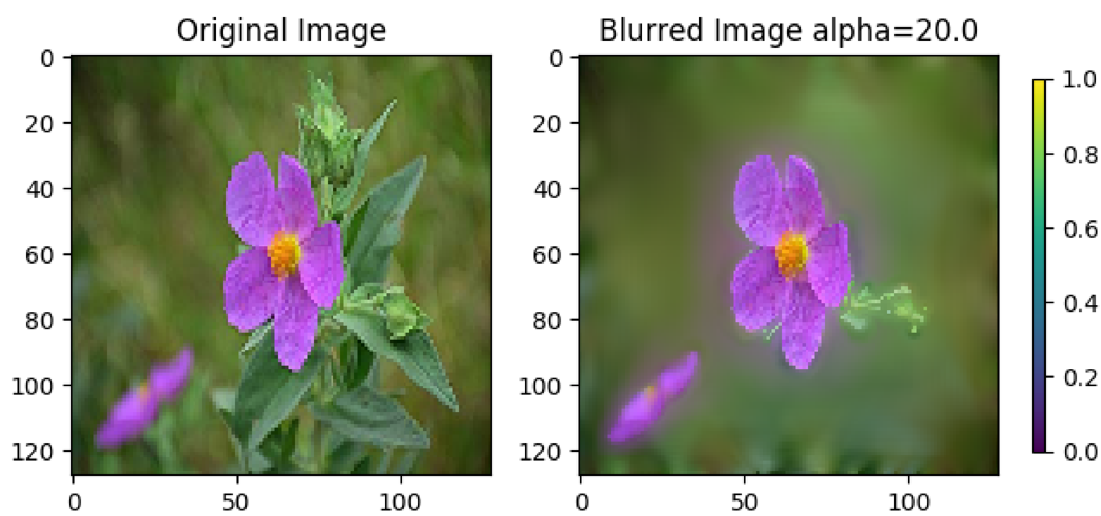


Figure 6: (a) 0.2 alpha (b) 0.4 alpha (c) 0.6 alpha (d) 0.8 alpha (e) alpha

Final output image with blurred background



[a]



[b]

Figure 7: (a) Original and blurred bird.png (b)Original and blurred flower.png

Note:

- Initially we were trying to implement the same algorithm for a smaller image and the results were not satisfactory. As we increased the size , the results improved and it is possible that for original size image the output would improve even further. However we could not test this as the mean shift code is slow and takes a lot of time to run if take the original size image itself.