ASSIGNMENT 1

COMPUTER VISION - SPRING 2019

B S N V Chaitanya(S20160020115) D Sumanth(S20160020125)
K Anjali Poornima(S20160020132)

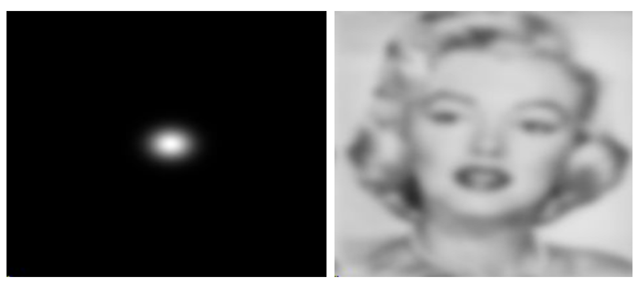
QUESTION 1 - Creating a hybrid image from two images

Analysing the steps for a pair of images:

1. Get the high frequency component (by applying high pass gaussian filter in frequency domain with sigma = α) from image A

a) High pass gaussian filter(α = 25) b) High pass filter applied on image A

2. Get the low frequency component(by applying low pass gaussian filter in frequency domain with sigma = β) from image B

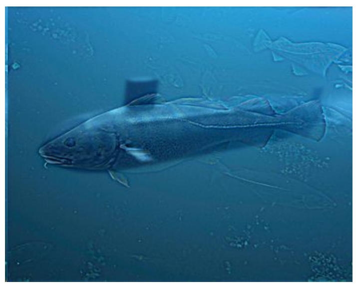
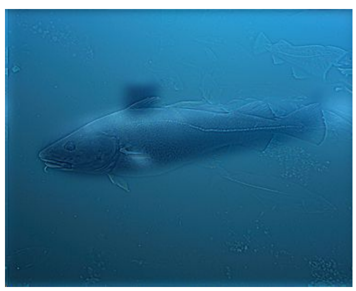


a)Low pass gaussian filter(β = 10) b) Low pass filter applied on image B

3. Sum up the low frequency information and high frequency information to get hybrid image.

Image A Image B Hybrid Image

Analysis of the hybrid image with varying sigma for low pass and high pass :

High Pass (α) Fish(A)	LowPass (β) Submarine(B)	Hybrid Image	Remarks
10	50		Since α is low, the high pass filter allows most of the high frequency components and also since β is high, the image B is more blurred. Hence edges from Image A will be more prominent.
30	30		Both α and β are equal, hence the high frequencies from Image A and low frequencies from Image B are equally preserved.
50	10		Since α is high, the high pass filter rejects most of the frequencies , only strong edges are preserved and also since β is low, the image B is not much blurred. Hence Image B will be more prominent.

Code:

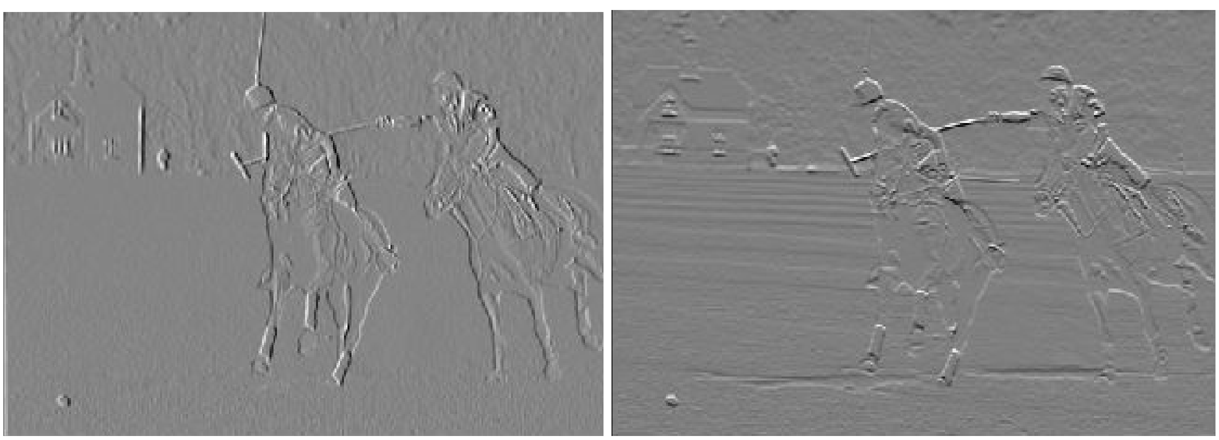
```
import cv2, math
import numpy as np
from numpy.fft import fft2, fftshift, ifft2, ifftshift
def gaussianFilter(img, sigma, highPass, name):
   numRows, numCols, _ = img.shape
   centerX = int(numRows/2)
   centerY = int(numCols/2)
   filter = np.array([[math.exp(-((i - centerX)**2 + (j - centerY)**2) / (2 * sigma**2)) for j in
range(numCols)] for i in range(numRows)])
   filter = 1 - filter if highPass else filter
   cv2.imwrite("result/"+name+".bmp", filter*255.0)
   return filter

def convolveImages(filter, image):
   product = np.zeros(image.shape) + 0.j
```

```
    for i in range(image.shape[2]):
            freqImage = fftshift(fft2(image[:,:,i]))
            product[:,:,i] = (ifft2(ifftshift(freqImage * filter)))
    return np.real(product)

img1 = cv2.imread("bird.bmp")
img2 = cv2.imread("plane.bmp")
#print(img1.shape, img2.shape)
highPassGaussian = gaussianFilter(img1, 20, True, "highPassGaussian")
lowPassGaussian = gaussianFilter(img2, 30, False, "lowPassGaussian")

highPassImg = convolveImages(highPassGaussian, img1)
lowPassImg = convolveImages(lowPassGaussian, img2)

cv2.imwrite("result/high.bmp", highPassImg)
cv2.imwrite("result/low.bmp", lowPassImg)

hybridImg = highPassImg + lowPassImg
cv2.imwrite("results/hybridImg_bird_plane_20_30.bmp", hybridImg)
```

QUESTION 2 - Corner Detection(Shi-Tomasi and Harris Corner Detection)

Analysing the steps for Shi-Tomasi Corner Detection Algorithm:

1. Get the gradient at each point in image using sobel filter in x and y directions.



a) Vertical edges(X gradient) **b) Horizontal edges(Y gradient)**

2. Create the H matrix from the entries in the gradient by calculating I_{xx}, I_{yy} and I_{xy} by considering a window of 3x3.
3. Compute the eigenvalues using inbuilt numpy command(np.linalg..eig(H)).
4. Find those points with large response where λ_{min} > threshold(in our case, threshold = 3 for a normalized image).

Corners using Shi-Tomasi before non-max suppression

5. Choose those points where λ_{min} is a local maximum as features i.e., do non-max suppression.

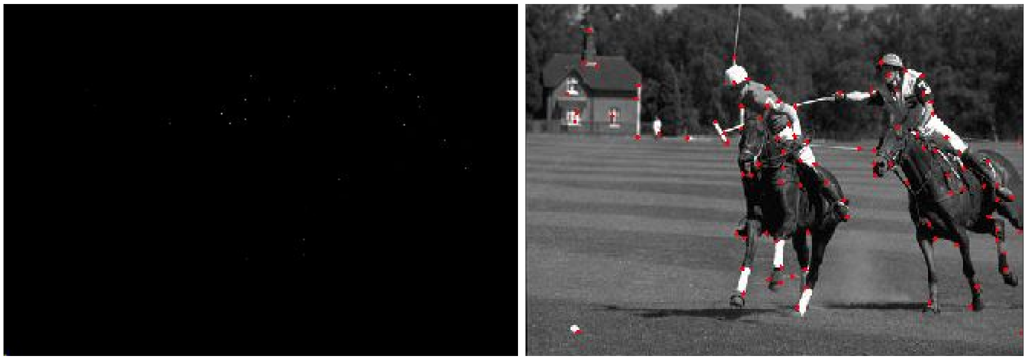
Corners using Shi-Tomasi after non-max suppression

Analysing the steps for Harris Corner Detection algorithm:

1. Get the gradient at each point in image using sobel filter in x and y directions similar to that of doing Shi-Tomasi Corner detection.
2. Create the H matrix from the entries in the gradient by calculating I_{xx} ,I_{yy} and I_{xy} by considering a window of 3x3.
3. Compute the eigenvalues using inbuilt numpy command(np.linalg..eig(H)).
4. Threshold the Harris Operator for feature detection , $f = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2$, f = determinant(H) - k(Trace(H))^2 . If f>Threshold(here 5), consider those points.

Corners using Harris Corner detection before non-max suppression

5. Choose those points where f is a local maximum as features i.e., do non-max suppression.



Corners using Harris Corner detection after non-max suppression

Conclusions made:

Harris corner detection is the Shi-Tomasi modification. While the **Harris algorithm** was **more computationally efficient** , the **Shi-Tomasi algorithm** was found to be more **accurate**. The runtime for Shi-Tomasi was **`15.19997262954712 seconds`** while that of Harris was **`5.642521381378174 seconds.`**

Code:

```
import cv2
from matplotlib import pyplot as plt
import numpy as np
from numpy import linalg as LA
import time

imgName = "image3.jpg"
imgRgb = cv2.imread(imgName)
img = cv2.cvtColor(imgRgb, cv2.COLOR_BGR2GRAY)/255

sobelX = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
sobelY = np.transpose(sobelX)

def convolveSobel(conv_img, filterType, w = 3):
        n, m = conv_img.shape
        w = filterType.shape[0]
        conv_img = np.pad(conv_img,(int(w/2),int(w/2)),mode='constant')
        temp1=[]
        for i in range(n):
        temp=[]
        for j in range(m):
        k = conv_img[i:i+w, j:j+w]
        val = np.multiply(k, filterType)
        temp.append(np.sum(val))
        temp1.append(temp)
        return np.array(temp1)
Gx = convolveSobel(img, sobelX)
Gy = convolveSobel(img, sobelY)
```

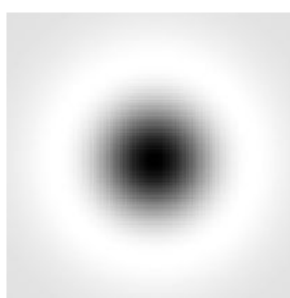
```
Ixx=Gx**2
Iyy=Gy**2
Ixy=Gx*Gy

def ShiTomasi(img, Ixx, Iyy, Ixy, thr = 5, w = 3):
    n, m, _ = img.shape
    corners = np.zeros((n,m))
    wlen = int(w/2)
    for y in range(wlen, n-wlen):
    for x in range(wlen, m-wlen):
    H = np.zeros((2,2))
    Sxx=np.sum(Ixx[y-wlen:y+1+wlen, x-wlen:x+1+wlen])
    Syy=np.sum(Iyy[y-wlen:y+1+wlen, x-wlen:x+1+wlen])
    Sxy=np.sum(Ixy[y-wlen:y+1+wlen, x-wlen:x+1+wlen])
    H[0,0]=Sxx
    H[0,1]=Sxy
    H[1,0]=H[0,1]
    H[1,1]=Syy
    eigenValue, v= LA.eig(H)
    min_eigen_value = min(eigenValue[0],eigenValue[1])
    #print(min_eigen_value)
    if min_eigen_value>thr:
    corners[y, x] = min_eigen_value
    return corners

def Harris(img, Ixx, Iyy, Ixy, thr = 5, w = 3, lambd = 0.04):
    n, m, _ = img.shape
    corners = np.zeros((n,m))
    wlen = int(w/2)
    for y in range(wlen, n-wlen):
    for x in range(wlen, m-wlen):
    Sxx=np.sum(Ixx[y-wlen:y+1+wlen, x-wlen:x+1+wlen])
    Syy=np.sum(Iyy[y-wlen:y+1+wlen, x-wlen:x+1+wlen])
    Sxy=np.sum(Ixy[y-wlen:y+1+wlen, x-wlen:x+1+wlen])

    det = (Sxx * Syy) - (Sxy**2)
    trace = Sxx + Syy
    r = det - lambd*(trace**2)
    if r > thr:
    corners[y, x] = r
    return corners

def NonMaxSuppression(img, corners, stride = 10):
    corner_img = img.copy()
    n, m, _ = corner_img.shape
    cornersNew = np.zeros(corners.shape)
    wlen = stride//2
```

```
        for y in range(wlen, n-wlen):
        for x in range(wlen, m-wlen):
        arr = np.array(corners[y:y+stride, x:x+stride])
        if(np.amax(arr) != 0):
        m_at = np.where(arr == np.amax(arr))
        if(m_at[0] == int(stride/2) and m_at[1] == int(stride/2)):
                cornersNew[y, x] = corners[y+m_at[0], x+m_at[1]]
                cv2.circle(corner_img,(x+m_at[1],y+m_at[0]),2,255,-1)
        return corner_img

start_time = time.time()
corners = ShiTomasi(imgRgb, Ixx, Iyy, Ixy, thr = 3)
corner_img = NonMaxSuppression(imgRgb, corners, stride = 10)
plt.imshow(corner_img, cmap="gray")
plt.figure()
cv2.imwrite("out_ShiTomasi.jpg", corner_img)
print("---ShiTomasi: %s seconds ---" % (time.time() - start_time))

start_time = time.time()
corners = Harris(imgRgb, Ixx, Iyy, Ixy)
corner_img = NonMaxSuppression(imgRgb, corners, stride = 15)
plt.imshow(corner_img, cmap="gray")
plt.figure()
cv2.imwrite("out_Harris.jpg", corner_img)
print("---Harris: %s seconds ---" % (time.time() - start_time))
```

QUESTION 3 - Implement a Laplacian blob detector

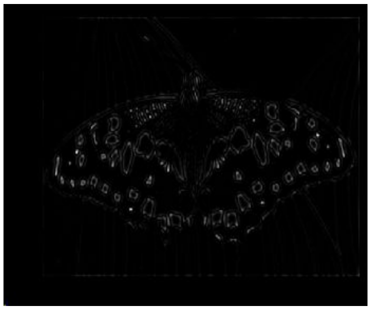
Analysing the steps to implement blob detection for a sample image:

1. A normalized Laplacian of Gaussian is generated using the formula
   $(x^2 + y^2 - 2\sigma^2)e^{-(x^2 + y^2)/2\sigma^2}$.



Laplacian of Gaussian with sigma = 15

2. The image is filtered with scale-normalized Laplacian at a scale, say sigma = σ.
3. The square of Laplacian response for current level of scale space is saved.
4. Generate a scale spaced responses by increasing the sigma by a factor k.

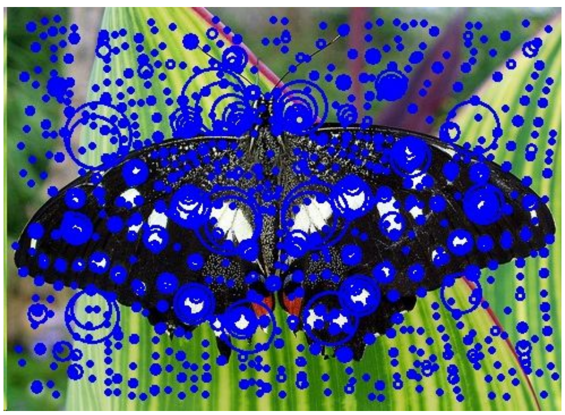


$\sigma = 2$

$\sigma = 5$

$\sigma = 8$

5. Non-maximum suppression in scale space is performed. This is done by considering the 26 neighbours(in a cube of 3 x 3 x 3) around a pixel(x,y). Check whether this pixel(x,y) is maximum in the neighbourhood . If maximum then that center and corresponding scale is stored otherwise rejected.
6. After performing the Non-Maximum Suppression, the resulting circles are displayed with their characteristic radius($2 * \sigma^{1/2}$).



Before Non-Max suppression

After Non-Max suppression

Code:

```
import cv2, numpy as np
import matplotlib.pyplot as plt
from skimage.feature import peak_local_max

imgName = "suntest"
imgRgb = cv2.imread(imgName+".jpg")
img = cv2.cvtColor(imgRgb, cv2.COLOR_BGR2GRAY)/255
n,m=img.shape

def LOG(sigma):
	size = int(sigma*6)
	size = size + 1 if(size%2==0) else size
	log = np.zeros((size, size))
	centerX = int(size/2)
	centerY = int(size/2)
	def getVal(x, y):
	return ((x-centerX)**2 + (y-centerY)**2 - 2*(sigma**2))*np.exp(-((x-centerX)**2 +
```

```
(y-centerY)**2)/(2*(sigma**2)))
    log = [[getVal(x, y)*(sigma**2) for y in range(size)] for x in range(size)]
    return np.array(log)

def convolveImg(img, log):
    M, N = img.shape
    ans = np.zeros((M, N))
    #res = np.zeros((M+X, N+Y))
    w = log.shape[0]
    w_pad = int(w/2)
    padImg = np.pad(img, (w_pad, w_pad), mode = 'constant')

    for i, j in np.ndindex(ans.shape):
    ans[i, j] = np.sum(np.multiply(padImg[i:i+w, j:j+w], log))
    #res[i, j] = ans[i, j]/np.sum(log)
    return np.square(ans)

def getResponse(img, sigma, k, numbers = 10):
    responses = []
    M, N = img.shape

    for _ in range(numbers):
    print(sigma)
    log = LOG(sigma)
    conv = convolveImg(img, log)
    #np.concatenate(conv, np.zeros(M, padY))
    responses.append(conv)
    plt.imshow(conv, cmap="gray")
    plt.figure()
    sigma *= k
    return np.array(responses)

K = 1.2509
sigma = 4
num_scales = 10
responses = getResponse(img, sigma, K, num_scales)

def getMaxes(img, st_dev, K, conv):
    copy = img.copy()
    radius = np.zeros([img.shape[0], img.shape[1]])
    for k in range(num_scales):
    s = int(st_dev*6)
    s = s + 1 if(s%2==0) else s
    coordinates = peak_local_max(conv[k], min_distance=(k+1)*5)
    for i in coordinates:
    cv2.circle(copy,(i[1],i[0]), int(st_dev*np.sqrt(2)), (255,0,0), 2)
    radius[i[0],i[1]] = s
```

```
        st_dev*=K
        cv2.imwrite("out_"+imgName+"_before_NMS.jpg",copy)
        plt.imshow(copy)
        return radius

rad_matrix = getMaxes(imgRgb, sigma, K, responses)
def nonMax(radius):
        n, m = radius.shape
        for i, j in np.ndindex(n, m):
        if radius[i,j] != 0:
        wsize=int(radius[i,j]/2)-int(radius[i,j]/3)
        for i1 in range(i-wsize,i+wsize):
        for j1 in range(j-wsize,j+wsize):
                #if (i1>=0 and i1<n) and (j1>=0 and j1<m):
                if radius[i1,j1] < radius[i,j]:
                radius[i1,j1]=0
        return radius

rad_matix = nonMax(rad_matrix)
copy=imgRgb.copy()
for i in range(n):
        for j in range(m):
        if rad_matrix[i,j]!=0:
        cv2.circle(copy, (j,i), int((rad_matrix[i,j]/7)*np.sqrt(2)), (0,0,255), 2)
plt.imshow(copy)
cv2.imwrite("out_"+imgName+".jpg",copy)
```