**Assignment 1:**

**6A. The code for generating the Gaussian and Laplacian pyramid of the Elephant image respectively:**

import cv2

import numpy as np

from matplotlib import pyplot as plt

def gaussian\_step(image):

# function to perform the operation for the gaussian pyramid

w , h , l = image.shape

w\_d = int(w / 2)

h\_d = int(h / 2)

oput\_size = (h\_d , w\_d)

G\_output = np.zeros((w\_d, h\_d, l))

for color in range(3):

G\_output[:, :, color] = cv2.GaussianBlur( G\_output[:, :, color] , ( 3, 3 ) , 3 )

for color in range(3):

G\_output[:, :, color] = cv2.resize( image[:, :, color] , oput\_size, cv2.INTER\_AREA )

return G\_output

def laplacian\_step(g,G):

# function to perform the operation for the laplacian pyramid

w , h , l = G.shape

next\_size = (h,w)

G\_output = np.zeros((w, h, l))

for color in range(3):

ch = g[:, :, color]

G\_output[:, :, color] = cv2.resize( ch , next\_size, cv2.INTER\_AREA)

for color in range(3):

G\_output[:, :, color] = cv2.GaussianBlur( G\_output[:, :, color] , ( 3, 3 ) , 3 )

L\_output = G - G\_output

return L\_output

def pyramid\_gaussian(image , layers):

# a function to build the layers of the gaussian pyramid

w , h , l = image.shape

G = image.copy()

gp = [G]

for i in range(layers):

G = gaussian\_step(G)

gp.append(G)

return gp

def pyramid\_laplacian(gp):

# a function to build the layers of thbbkerx pyramid

w , h , l = gp[0].shape

layers = len(gp)

lp = []

for j in range(layers-2, -1, -1):

w , h , l = gp[j].shape

L = np.zeros((w, h, l))

L = laplacian\_step(gp[j+1],gp[j])

lp.append(L)

return lp

img = plt.imread('C:/Users/New User/Documents/MATLAB/elephant.jpeg',1)

img\_G\_list = pyramid\_gaussian(img,7)

for i in range(8):

# plt.subplot(4, 2, i + 1)

# using pyrDown() function

layer\_G = np.uint8(img\_G\_list[i])

# plt.imshow(layer\_G)

cv2.imshow("Gaussian Layers", layer\_G)

cv2.imwrite('C:/Users/New User/Documents/MATLAB/Gaussian\_Layer\_'+str(i)+'.jpg', layer\_G)

cv2.waitKey(0)

cv2.destroyAllWindows()

#

img\_L\_list = pyramid\_laplacian(img\_G\_list)

#

for j in range(7):

# plt.subplot(4, 2, j + 1)

# using pyrDown() function

layer\_L = np.uint8(img\_L\_list[j])

# plt.imshow(layer\_L)

cv2.imshow("Laplacian Layers", layer\_L)

cv2.imwrite('C:/Users/New User/Documents/MATLAB/Laplacian\_Layer\_'+str(j)+'.jpg', layer\_G)

cv2.waitKey(0)

cv2.destroyAllWindows()

######### Reconstruction verification ##############

img\_L\_list.reverse()

for k in range(6, 0, -1):

size = (img\_L\_list[k].shape[1], img\_L\_list[k].shape[0])

resized = np.zeros\_like(img\_L\_list[k])

G\_image = img\_G\_list[k+1]

for color in range(3):

resized[:, :, color] = cv2.resize(G\_image[:, :, color], size, cv2.INTER\_AREA)

for color in range(3):

resized[:, :, color] = cv2.GaussianBlur(resized[:, :, color] , ( 3, 3 ) , 3)

img\_G\_list[k] = img\_L\_list[k] + resized

cv2.imshow("reconstructed\_image", np.uint8(img\_G\_list[0]))

cv2.waitKey(0)

cv2.imwrite('C:/Users/New User/Documents/MATLAB/reconstructed\_elephant.jpg', np.uint8(img\_G\_list[0]))

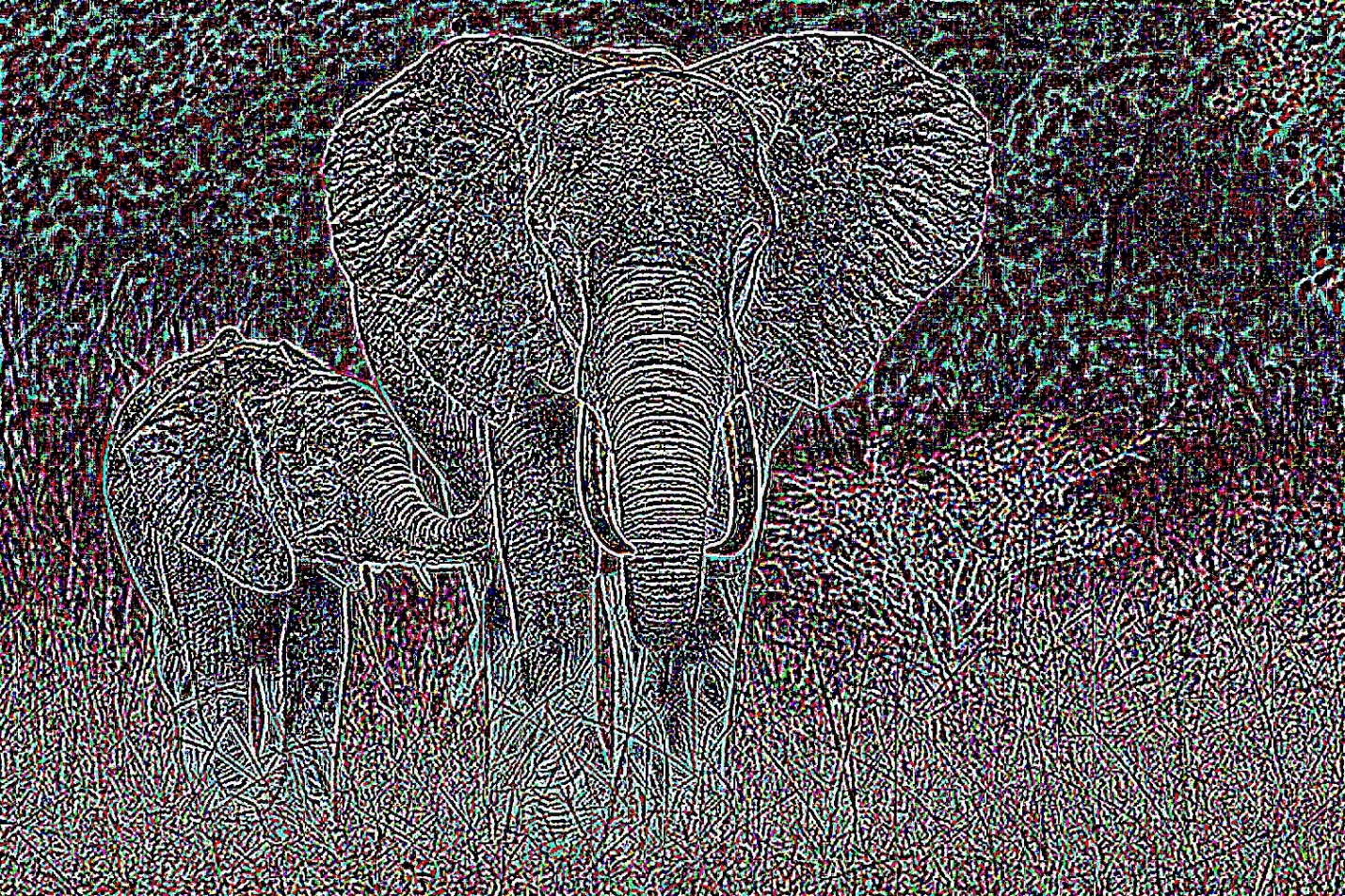
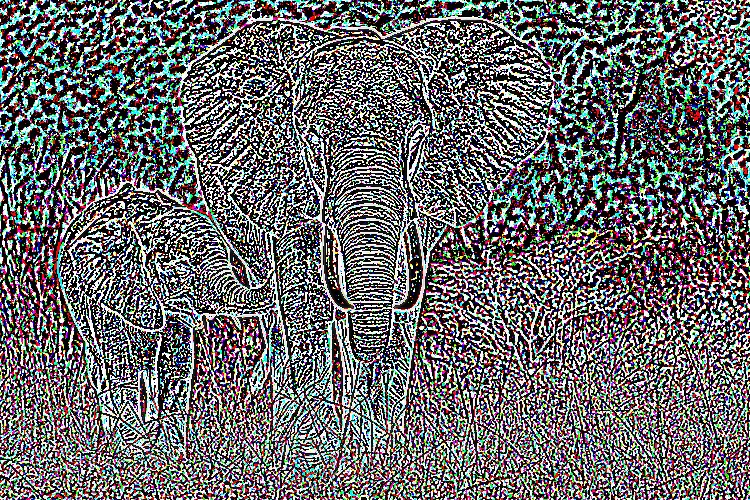
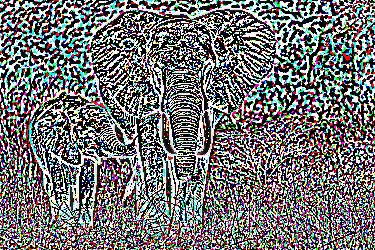
cv2.destroyAllWindows()

The output for the code is as follows:

The Gaussian layers**: figure 1**



The Laplacian layers**: figure 2**

* 

**6E. The orapple example for the multiresolution blending exercise and the intermediate Laplacian layers: figure 3**

The code is as follows:

import cv2

import numpy as np

from matplotlib import pyplot as plt

########## Section 6(e) - Performing apha blending #######################

def gaussian\_step(image):

# function to perform the operation for the gaussian pyramid

w , h , l = image.shape

w\_d = int(w / 2)

h\_d = int(h / 2)

oput\_size = (h\_d , w\_d)

G\_output = np.zeros((w\_d, h\_d, l))

for color in range(3):

G\_output[:, :, color] = cv2.GaussianBlur( G\_output[:, :, color] , (3, 3) , 3 )

for color in range(3):

G\_output[:, :, color] = cv2.resize( image[:, :, color] , oput\_size, cv2.INTER\_AREA)

return G\_output

def laplacian\_step(g,G):

# function to perform the operation for the laplacian pyramid

w , h , l = G.shape

next\_size = (h,w)

G\_output = np.zeros((w, h, l))

for color in range(3):

ch = g[:, :, color]

G\_output[:, :, color] = cv2.resize( ch , next\_size, cv2.INTER\_AREA)

for color in range(3):

G\_output[:, :, color] = cv2.GaussianBlur( G\_output[:, :, color] , ( 3, 3 ) , 3 )

L\_output = G - G\_output

return L\_output

def pyramid\_gaussian(image , layers):

# a function to build the layers of the gaussian pyramid

w , h , l = image.shape

G = image.copy()

gp = [G]

for i in range(layers):

G = gaussian\_step(G)

gp.append(G)

return gp

def pyramid\_laplacian(gp):

# a function to build the layers of thbbkerx pyramid

w , h , l = gp[0].shape

layers = len(gp)

lp = []

for j in range(layers-2, -1, -1):

w , h , l = gp[j].shape

L = np.zeros((w, h, l))

L = laplacian\_step(gp[j+1],gp[j])

lp.append(L)

return lp

def blender(right, left, mask):

blend = []

k = len(right)

for e in range(0,k):

left[e] = left[e].astype(float)

right[e] = right[e].astype(float)

mask[e] = mask[e].astype(float) / 255

# Multiply the foreground with the alpha matte

right[e] = np.multiply(mask[e], right[e])

# Multiply the background with ( 1 - alpha )

left[e] = np.multiply(1 - mask[e], left[e])

#print(e)

# Add the masked foreground and background.

blend.append(np.add(left[e],right[e]))

return blend

####################### Read the images #################

orange = cv2.imread("C:/Users/New User/Documents/MATLAB/orange.jpeg")

apple = cv2.imread("C:/Users/New User/Documents/MATLAB/apple.jpeg")

mask = cv2.imread("C:/Users/New User/Documents/MATLAB/mask.jpg")

# Generate the gaussian pyramids for all three images(i.e. apple, orange, mask)

orange\_1 = pyramid\_gaussian(orange, 4)

apple\_1 = pyramid\_gaussian(apple, 4)

mask\_1 = pyramid\_gaussian(mask, 4)

# Generate the laplacian pyramids for all three images(i.e. apple, orange, mask)

orange\_2 = pyramid\_laplacian(orange\_1)

apple\_2 = pyramid\_laplacian(apple\_1)

orange\_2.reverse()

apple\_2.reverse()

L\_blend = blender(orange\_2, apple\_2, mask\_1)

r = orange\_1[4].astype(float)

l = apple\_1[4].astype(float)

mask\_1[4] = mask\_1[4].astype(float) / 255

r = np.multiply(mask\_1[4], r)

l = np.multiply(1-mask\_1[4], l)

G\_image = np.add(l, r)

for k in range(3, -1, -1):

size = (L\_blend[k].shape[1], L\_blend[k].shape[0])

cv2.imwrite('C:/Users/New User/Documents/MATLAB/Blended\_Reconstructed\_Layer\_'+str(k)+'.jpg', L\_blend[k]+128)

resized = np.zeros\_like(L\_blend[k])

for color in range(3):

resized[:, :, color] = cv2.resize(G\_image[:, :, color], size, cv2.INTER\_AREA)

for color in range(3):

resized[:, :, color] = cv2.GaussianBlur(resized[:, :, color] , ( 3, 3 ) , 3)

L\_blend[k] = L\_blend[k] + resized

G\_image = L\_blend[k]

cv2.imshow("Blended\_image", np.uint8(L\_blend[0]))

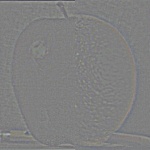
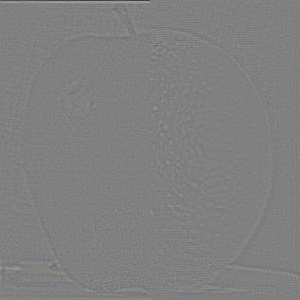
cv2.waitKey(0)

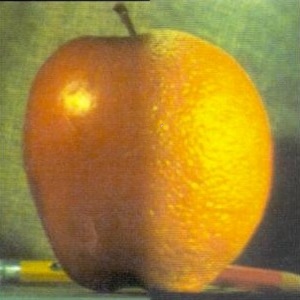
cv2.imwrite('C:/Users/New User/Documents/MATLAB/Multi\_resolution blending.jpg', np.uint8(L\_blend[0]))

cv2.destroyAllWindows()

The output is as follows:

Multi-resolution blending: **figure 3**





Assignment 2:

**4B & 4C. The code for the detector and descriptor combination along with the estimation of the homography matrix is as follows:**

Note: A key reference for the estimation of homography is #reference: http://nghiaho.com/?page\_id=671

import cv2

import math

import numpy as np

from math import sqrt

from math import pi

def findCorners(image, window\_size, k, threshold):

## Gradient Calculation Using Sobel

dx = cv2.Sobel(image,cv2.CV\_32F,1,0,ksize=5)

dy = cv2.Sobel(image,cv2.CV\_32F,0,1,ksize=5)

Ixx = dx\*dx

Ixy = dx\*dy

Iyy = dy\*dy

## Gaussian Blur

Ixx = cv2.GaussianBlur(Ixx,(5,5),2)

Iyy = cv2.GaussianBlur(Iyy,(5,5),2)

Ixy = cv2.GaussianBlur(Ixy,(5,5),2)

height, width = Ixx.shape

offset = np.uint8((window\_size-1)/2)

keypoint = []

win = np.zeros(Ixx.shape,np.float32)

for y in range(offset, height-offset):

for x in range(offset, width-offset):

#Calculate sum of squares

windowIxx = Ixx[y-offset:y+offset+1, x-offset:x+offset+1]

windowIxy = Ixy[y-offset:y+offset+1, x-offset:x+offset+1]

windowIyy = Iyy[y-offset:y+offset+1, x-offset:x+offset+1]

# Find the weighted sum of the window of gradient

Sxx = windowIxx.sum()

Sxy = windowIxy.sum()

Syy = windowIyy.sum()

# Find determinant and trace, use to get corner response

det = (Sxx \* Syy) - (Sxy\*\*2)

trace = Sxx + Syy

r = det - k\*(trace\*\*2)

# r=det/trace

win[y][x] = r

a,b,c,d = cv2.minMaxLoc(win)

threshold = b\*.6

for y in range(0, height):

for x in range(0, width):

b = win[y][x]

if b > threshold:

win[y][x] = b

else:

win[y][x] = 0

for y in range(offset, height-offset):

for x in range(offset, width-offset):

r = win[y-offset:y+offset+1, x-offset:x+1+offset]

a,b,c,d = cv2.minMaxLoc(r)

abc = np.zeros((window\_size,window\_size),np.float32)

abc[d] = b

win[y-offset:y+offset+1, x-offset:x+offset+1]=abc

for y in range(0, height):

for x in range(0, width):

b = win[y][x]

if b>threshold:

keypoint.append(cv2.KeyPoint(x,y,1,-1,0,0,-1))

return dx, dy, keypoint

def discriptors(gx,gy,keypoint):

mag = np.zeros(gx.shape)

angle = np.zeros(gx.shape)

angle = np.degrees(np.arctan2(gy,gx))%360

mag = ((gx\*gx)+(gy\*gy))\*\*.5

binarray=[]

for ikp in keypoint:

try:

x,y = ikp.pt

x = int(x)

y = int(y)

# 16 \* 16 window of angle

win = angle[y-8:y+8, x-8:x+8]

# 16 \* 16 window of magnitude

win2 = mag[y-8:y+8, x-8:x+8]

if len(win)<16:

continue

x1 = -2

y1 = -2

bindup = []

#Getting 16 4\*4 windows

for x in range(0,4):

x1 = -2

y1 = y1+4

for y in range(0,4):

x1 = x1+4

rsmall = win[y1-2:y1+2, x1-2:x1+2] #4\*4 window of magnitude and angle

rsmall2 = win2[y1-2:y1+2, x1-2:x1+2]

bi = dict()

for k in range(0,8):

bi[k] = 0

for i in range(0,4):

for j in range(0,4):

# getting the magnitude for each pixel and according to the angle distribute portions of magnitude between multiple bins

no=np.uint8(rsmall[i][j]/45)

bi[no] = (rsmall2[i][j]/45)\*((no\*45+45)-rsmall[i][j])

if no+1 in bi:

bi[no+1] = (rsmall2[i][j]/45)\*(rsmall[i][j]-no\*45)

else:

bi[0] = (rsmall2[i][j]/45)\*(rsmall[i][j]-no\*45)

# 128 dimensional 16\*8 bin

bindup.extend(list(bi.values()))

# Normalising the Histogram

barr = np.array(bindup,np.float32)

div = ((barr\*\*2).sum())\*\*.5

# Clipping the value to .2

binarray.append(np.clip((barr/div),0,.2))

except:

continue

return np.array(binarray,np.float32)

## Function For Matching Descriptors : - SSD

def match(binarray1,binarray2):

temp\_list = [0.0,0,0]

matched\_pts = []

matches = []

for i in range(len(binarray1)):

temp = 100

#SSD

for j in range(len(binarray2)):

k = binarray1[i] - binarray2[j]

k = (k\*k).sum()

if k < temp:

temp = k

temp\_list = [k,i,j]

t = temp\_list.copy()

matches.append(t)

sort = sorted(matches, key=lambda tup: tup[2])

last\_used = sort[0][2]

xyz = []

main\_match = []

#Custom function to remove bad matches

for num in sort:

if num[2] == last\_used:

xyz.append(num)

elif num[2] > last\_used:

ans = sorted(xyz, key=lambda tup: tup[0])

main\_match.append(ans[0])

xyz = []

last\_used = num[2]

xyz.append(num)

for t in main\_match:

matched\_pts.append(cv2.DMatch(t[1],t[2],t[0]))

return matched\_pts

def matchFeatures(kp1, kp2, desc1, desc2, img1, img2):

print("Matching Features...")

matcher = cv2.BFMatcher(cv2.NORM\_L2, True)

matches = matcher.match(desc1, desc2)

return matches

#def calculateHomography(points\_1, points\_2):

#

# return h

## Function For Corner Detection

nimg1 = "Quadrilateral.jpg"

nimg2 = "Tansformed\_Quad.jpg"

I = cv2.imread("C:/Users/New User/Documents/MATLAB/"+nimg1,0)

I2 =cv2.imread("C:/Users/New User/Documents/MATLAB/"+nimg2,0)

Iorigin=cv2.imread("C:/Users/New User/Documents/MATLAB/"+ nimg1,0)

I2origin=cv2.imread("C:/Users/New User/Documents/MATLAB/"+ nimg2,0)

window\_size = 3

k = 0.03

threshold = 0

dx1, dy1, keypoints\_1 = findCorners(Iorigin, window\_size, k, threshold)

print("Finding Corners in 1")

img2 = cv2.drawKeypoints(Iorigin,keypoints\_1,I,color=(0,0,255), flags=0)

print("Marking corners in 1")

descriptors\_1 = discriptors(dx1,dy1,keypoints\_1)

print("Extracting feature descriptors")

dx2, dy2, keypoints\_2 = findCorners(I2origin, window\_size, k, threshold)

print("Finding Corners in 2")

img3 = cv2.drawKeypoints(I2origin, keypoints\_2, I2, color=(255,0,255), flags=0)

print("Marking corners in 2")

descriptors\_2 = discriptors(dx2, dy2, keypoints\_2)

print("Extracting feature descriptors")

match\_points = matchFeatures(keypoints\_1, keypoints\_2, descriptors\_1, descriptors\_2, I, I2)

#match\_points = match(descriptors\_1, descriptors\_2)

#print("Matching the points")

imgx2 = cv2.drawMatches(Iorigin, keypoints\_1, I2origin, keypoints\_2, match\_points, I2, flags=2)

print("Writing outputs")

# cv2.imshow("Image1",img2)

# cv2.imshow("Image2",img3)

# cv2.imshow("Image4",imgx2)

cv2.imwrite("C:/Users/New User/Documents/MATLAB/Corners\_detected\_1.jpg",img2)

cv2.imwrite("C:/Users/New User/Documents/MATLAB/Corners\_detected\_2.jpg",img3)

cv2.imwrite("C:/Users/New User/Documents/MATLAB/Matched\_corners.jpg",imgx2)

# Initialize lists

list\_kp1 = []

list\_kp2 = []

# For each match...

for mat in match\_points:

# Get the matching keypoints for each of the images

img1\_idx = mat.queryIdx

img2\_idx = mat.trainIdx

# x - columns

# y - rows

# Get the coordinates

(x1,y1) = keypoints\_1[img1\_idx].pt

(x2,y2) = keypoints\_2[img2\_idx].pt

# Append to each list

list\_kp1.append((x1, y1, 1))

list\_kp2.append((x2, y2, 1))

# Only three points were matched using the detector/descriptor combination

#

list\_kp1.append((165 , 210, 1))

list\_kp2.append((294 , 230, 1))

print("Estimating the rotation and translation parameters")

A = np.asarray(list\_kp1)

B = np.asarray(list\_kp2)

rows, cols = A.shape

Am = np.mean(A, axis=0) # centroid of source points

Bm = np.mean(B, axis=0) # centroid of destination points

M = np.dot((B - Bm).T, (A - Am)) # considering only rotation after repositioning with respect to origin

# singular value decomposition

U, S, Vt = np.linalg.svd(M)

# rotation matrix

R = np.dot(U, np.transpose(Vt))

rangle = (180\*math.acos(R[0,0]))/pi

print("The rotation angle is \n",rangle)

print(" and the corresponding matrix: %d\n",R)

# translation vector

L = B.mean(0) - np.dot(R, A.mean(0))

print("The translation matrix is %d", L)

# RMSE determined by using rigid body model

err = 0

for i in range(A.shape[0]):

Bp = np.dot(R, A[i, :]) + L

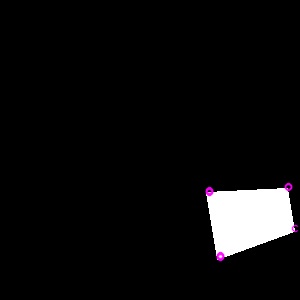
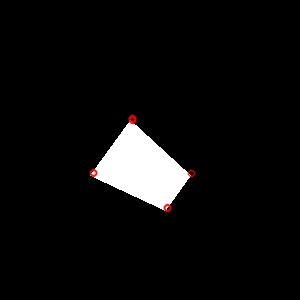
err += np.sum((Bp - B[i, :])\*\*2)

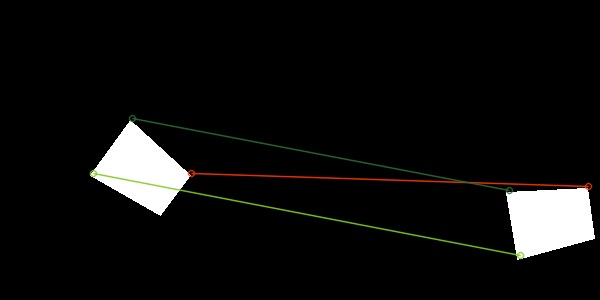
RMSE = np.sqrt(err/rows/3)

print("The margin of error:",RMSE)

For the exercise the output is as follows: **figure 4**

The detector and descriptor results**: figure 4**





The homography matrix estimation was done using three points of correspondence. All four point are detected in the detector but the there is no corresponding matches from the descriptor matching

The rotation parameter was calculated to be 24 degrees where the actual rotation is 45 degrees.

The translation parameters were x = 27, y = 124 as against the actual 30 and 100 respectively

**5A & 5B. The image stitching is performed by using SIFT and BF matcher**

The code is as follows:

import cv2

import numpy as np

from matplotlib import pyplot as plt

MAX\_FEATURES = 100

GOOD\_MATCH\_PERCENT = 0.15

def alignImages(im1, im2):

# Convert images to grayscale

im1Gray = cv2.cvtColor(im1, cv2.COLOR\_BGR2GRAY)

im2Gray = cv2.cvtColor(im2, cv2.COLOR\_BGR2GRAY)

# Detect ORB features and compute descriptors.

orb = cv2.ORB\_create(MAX\_FEATURES)

keypoints1, descriptors1 = orb.detectAndCompute(im1Gray, None)

keypoints2, descriptors2 = orb.detectAndCompute(im2Gray, None)

# Match features.

matcher = cv2.DescriptorMatcher\_create(cv2.DESCRIPTOR\_MATCHER\_BRUTEFORCE\_HAMMING)

matches = matcher.match(descriptors1, descriptors2, None)

# Sort matches by score

matches.sort(key=lambda x: x.distance, reverse=False)

# Remove not so good matches

numGoodMatches = int(len(matches) \* GOOD\_MATCH\_PERCENT)

matches = matches[:numGoodMatches]

# Draw top matches

imMatches = cv2.drawMatches(im1, keypoints1, im2, keypoints2, matches, None)

#cv2.imwrite("C:/Users/New User/Documents/MATLAB/matches.jpg", imMatches)

# Extract location of good matches

points1 = np.zeros((len(matches), 2), dtype=np.float32)

points2 = np.zeros((len(matches), 2), dtype=np.float32)

for i, match in enumerate(matches):

points1[i, :] = keypoints1[match.queryIdx].pt

points2[i, :] = keypoints2[match.trainIdx].pt

# Find homography

H, mask = cv2.findHomography(points1, points2, cv2.RANSAC, 3.0)

# print(H)

# Use homography

return H

def warp\_image(image, homography):

p1 = np.ones(3, np.float32)

p2 = np.ones(3, np.float32)

p3 = np.ones(3, np.float32)

p4 = np.ones(3, np.float32)

(y, x) = image.shape[:2]

p1[:2] = [0, 0]

p2[:2] = [x, 0]

p3[:2] = [0, y]

p4[:2] = [x, y]

min\_x = None

min\_y = None

max\_x = None

max\_y = None

for pt in [p1, p2, p3, p4]:

hp = np.dot(np.matrix(homography, np.float32),

np.matrix(pt, np.float32).T)

# Warping the image homogenous point with the homography matrix

hp\_arr = np.array(hp, np.float32)

normal\_pt = np.array([[hp\_arr[0] / hp\_arr[2]],

hp\_arr[1] / hp\_arr[2]], np.float32)

# converting the homogeous coordinates to carteasian cordinates

if(max\_x is None or normal\_pt[0, 0] > max\_x):

max\_x = normal\_pt[0, 0]

if(max\_y is None or normal\_pt[1, 0] > max\_y):

max\_y = normal\_pt[1, 0]

if(min\_x is None or normal\_pt[0, 0] < min\_x):

min\_x = normal\_pt[0, 0]

if(min\_y is None or normal\_pt[1, 0] < min\_y):

min\_y = normal\_pt[1, 0]

# define condition to check if warped images boundaries are in negative cartesian space

# if true the offset is calculated and the homography is added to the translation matrix to correct it

translationMatrix = np.zeros(shape=(3, 3))

translationMatrix[0] = [0, 0, -min\_x]

translationMatrix[1] = [0, 0, -min\_y]

newHomography = np.add(homography, translationMatrix)

# The new warped image

warp = cv2.warpPerspective(image,

newHomography,

(int(max\_x - min\_x), int(max\_y - min\_y)))

return warp, (int(min\_x), int(min\_y))

def create\_mosaic(images, origins):

# This will make the first image, in the images list,

# have an origin of (0,0)

# so that we can stitch them sequentially.

new\_origins = []

o\_x, o\_y = origins[0]

for origin in origins:

x, y = origin

new\_origins.append([x + abs(o\_x), y + abs(o\_y)])

# Dimensions for the mosaic

max\_height = 0

max\_width = -370

for image, origin in zip(images, new\_origins):

x, y = image.shape[:2]

max\_width += origin[0]

max\_height += origin[1]

max\_width += images[1].shape[1]

max\_height += images[1].shape[0]

final = np.ones((max\_height, max\_width, images[0].shape[2]), np.uint8)

final = cv2.cvtColor(final, cv2.COLOR\_BGR2BGRA)

for i in range(len(images)):

y, x, \_ = images[i].shape

o\_x, o\_y = new\_origins[i]

final[abs(o\_y):abs(y) + abs(o\_y),

abs(o\_x):abs(x) + abs(o\_x), :\_] = images[i]

# cv2.imshow('final', final)

# cv2.waitKey(0)

# cv2.descriptorstroyAllWindows()

return final

left = cv2.imread("C:/Users/New User/Documents/MATLAB/keble\_a.jpg")

middle = cv2.imread("C:/Users/New User/Documents/MATLAB/keble\_b.jpg")

right = cv2.imread("C:/Users/New User/Documents/MATLAB/keble\_c.jpg")

# cv2.namedWindow('image', cv2.WINDOW\_NORMAL)

# cv2.imshow("image", left\_image)

# cv2.waitKey(0)

# compute homography for left image

homography1 = alignImages(left, middle)

warped\_left, origin1 = warp\_image(left, homography1)

# compute homography for right image

homography2 = alignImages(right, middle)

warped\_right, origin2 = warp\_image(right, homography2)

# print("warped\_left: ", warped\_left.shape)

# print("middle: ", middle.shape)

# print("warped\_right: ", warped\_right.shape)

images = (warped\_left, warped\_right, middle)

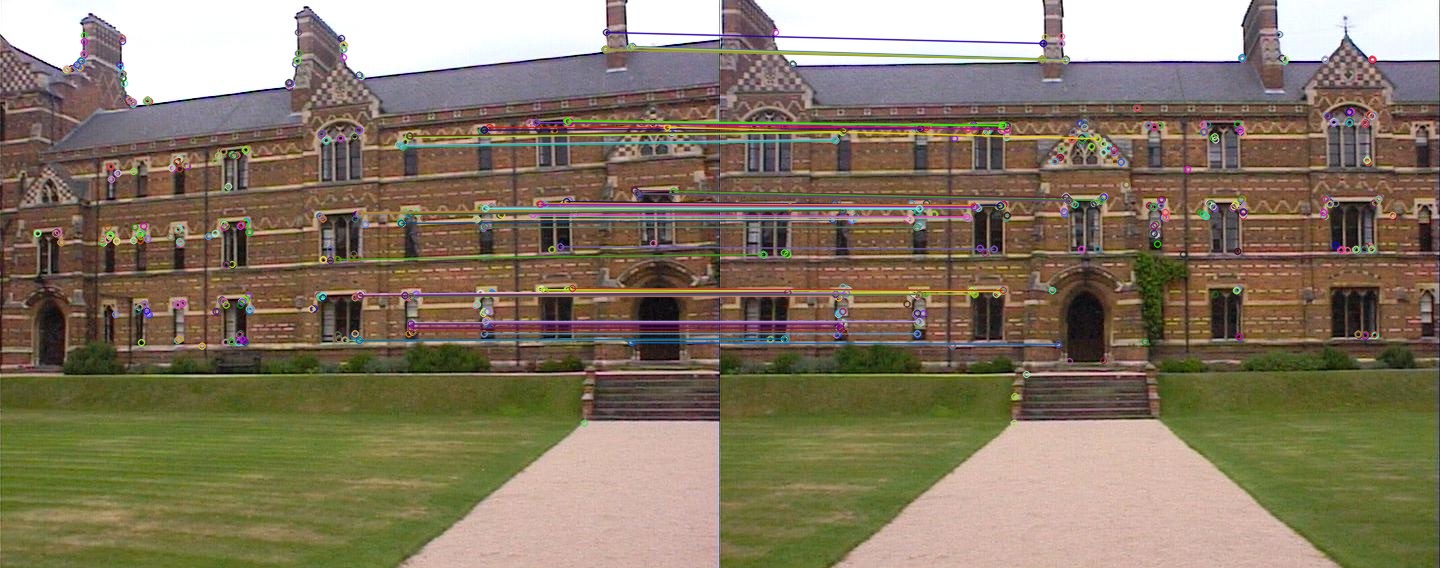
origins = (origin1, origin2, (0, 0))

mosaic1 = create\_mosaic(images, origins)

cv2.imwrite("C:/Users/New User/Documents/MATLAB/keble\_abc.jpg", mosaic1)

The results for the stitching exercise is as follows:

The image stitching: **figure 5**



A matching of features between the

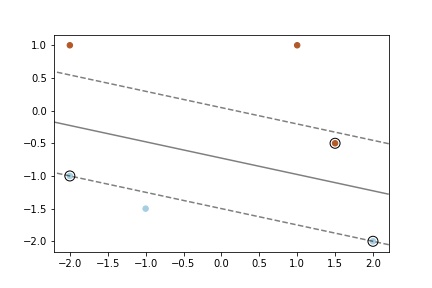
1. right and center profile
2. left and center profile
3. The final result

The left and right profile alignment is still not accurate and the degree of misalignment is clearly visible **figure 5**

**Assignmnet 3:**

**2.3 Plot of the SVW along with margin**: **figure 6**

**Figure 6**

****

* 1. **Resnet with the dataset augmentation was maintaining an accuracy of 71%**

PDF attached

4.1 Celebrities GAN output: **figure 7**

**Figure 7**

****

4.2 Squares dataset generator: **figure 8**

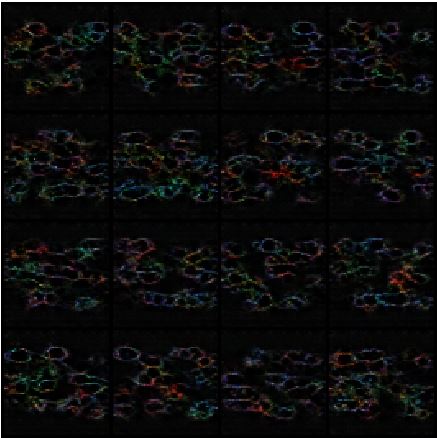
**Figure 8**

****

4.3 Squares generated using GAN: **figure 9**

**Figure 9**

GAN generated images

****

4.5 GANs code for generating dogs: **figure 10**

**Figure 10**

****