**Problem Description**

For Lab 3, all the problems were concerning image warping, which was performing some modifications to an image based on user input in this case. Problems included the following:

* Inserting an image into a specific space after performing an affine transformation using homographies. The image then was inserted into the destination image using forward and reverse warping.
* Modifying the given sample code given in class to perform multiple single point warps, using then the new image as the source.
* Implementing a multi-point warping given multiple input points. Very similar to the past problem.
* Implementing a program to morph a face into another. Similar to the example seen in class where a woman’s face and a tiger I believe are morphed together and can be seen overlapping each other.

**Algorithms Implemented**

For this lab, there were no algorithms implemented other than the ones Dr. Fuentes has provided in his sample code in the class web page.

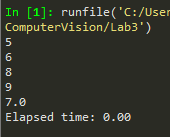
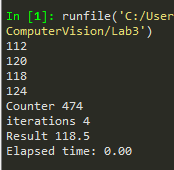
**Experimental Results**

1) For the first problem were a method needed to perform real valued index on an image, the instructions were a bit ambiguous, so I hope I implemented the expected problem. I used quiz 3 and the pixel averaging power point presentation in the class webpage. What I did was implement a method that would take two X and Y set of points ((0,0), (1,1)) and given those coordinates, I would iterate through the array, keep a counter variable that would add all of the values, and return the average of those given pixel values. As a demonstration I will use the following simple array.

[1,2,3], If I were to give my method the input of (1,1) and (2,2). The values between those points are:

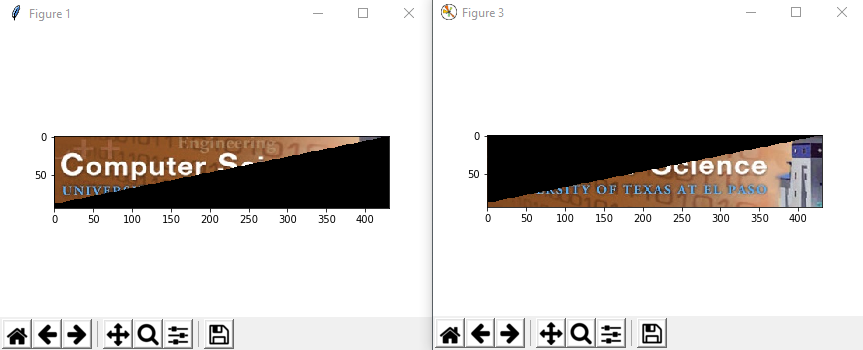
[4,5,6], 5,6,8, and 9. So 5+6+8+9 = 28/4 = 7. To make sure, I print the values the method does traverse

[7,8,9]]) as well as the result the method returns.

Simple Array result Given an actual picture calculating the avg of a region with   
 coords (1,1),(2,2)

2) For the second problem, I was able to successfully break the banner into two different triangles as shown below.



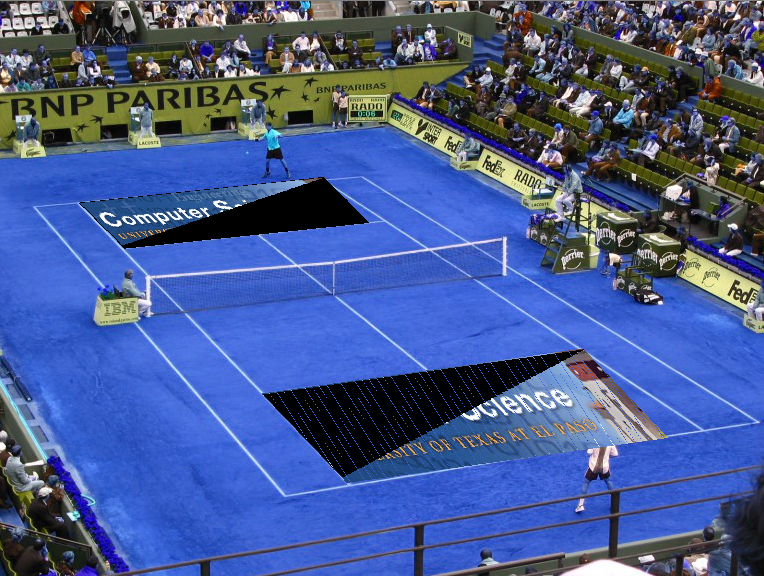
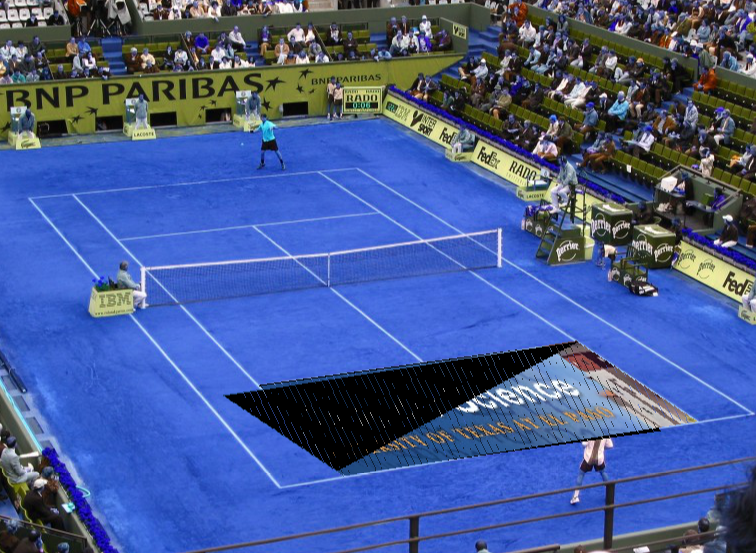
The colors are different since PIL and OpenCV have a different order on RGB/BGR, but since the functionality is the same, I did not change the colorspaces. I broke the triangles by first calculating the row and col size of the original image, and basically what I did was the following. I would take the source img and iterate only to the row and col that I got before, but each time I finished a loop, I would substract the “slope” to the columns, which I got by simply diving the img.shape[1]/img.shape[0]. Everytime I would be setting my second image to the value of the source, and I would delete that pixel from the source by setting it to zero. So by the time the loop ended, I would end with the source img which had the first half set to 0, and the second img would be a copy of the first half before changing those pixels to 0. The code to achieve this is the following.

for y in range (1,row):  
 for x in range (1,col):  
 if(x == col):  
 triangle2[y,x] = img[y,x]  
 continue

else:  
 triangle2[y,x] = img[y,x]  
 img[y,x] = 0

col -= slope

Afterwards, I had to modify Dr. Fuentes code a little bit in order for the ginput to accept 6 different points in the tennis image. To achieve this, I basically duplicated the code twice, and for the second triangle, I applied an np.flip function in order to correct the order of the pixels, since the code only accepted “top left, top right, and bottom left.” If I had left the array without modifying it first, the second triangle would not have aligned correctly as shown below.

The problem I encountered can be clearly seen from the pictures above. I was able to correctly break the triangles and align them properly, but once they overlapped, the region I had set to pixel intensity 0 covers the first triangle. I could not make it to properly eliminate the black triangle but the triangles are properly aligned. You can also somewhat see how the first triangle in the upper left is displayed correctly but the second’s black triangle hides some of it.

Approaches I tried to eliminate the black triangles. I tried to modify the code given to us by making the second triangle image check its region. If the pixels where it would be drawn had intensity 0, meaning it was black, to replace those pixels with the new ones, therefore replacing the black ones with the “good triangle”. And if the pixels were not 0, meaning there was the first good triangle there, to ignore those pixels. This in my head makes sense, although it may not be the best, most efficient way, I was not able to properly implement it. The way the sample code was made it somewhat difficult for me to modify it, as there were a lot of times I knew what the code was generally doing, but I did not have a solid enough understanding to modify it. For example the following line:

dest\_im = source\_im[source\_coords[0],source\_coords[1],:].reshape(dest\_rows,dest\_cols,3)

I know this is where the dest image is being set equal to the source img, but the way the pixels are benign copied all at once, made it hard for me to do something like

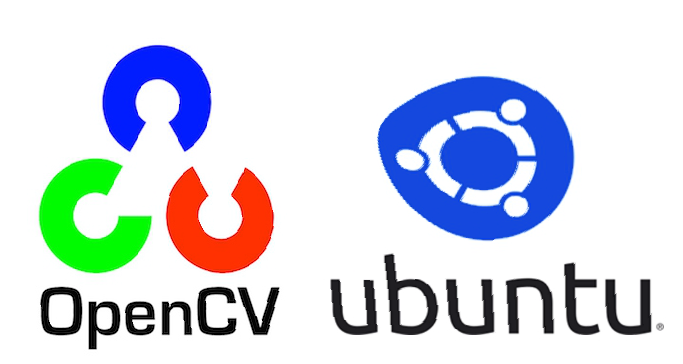
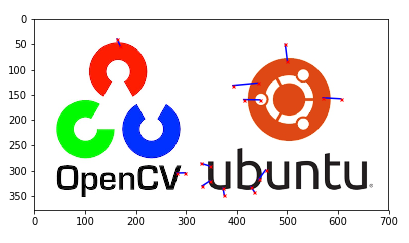
If(dest\_im[y,x] == 0)  
 dest\_im[y,x] == source\_im[y,x]  
else  
 ignore

3) Again, I was unable to actually modify the sample code in order to implement backward mapping. I know the theory that instead of checking the source img, transforming those cords into the new\_coords using the transformation method, and then setting the destination image equal to the source img as

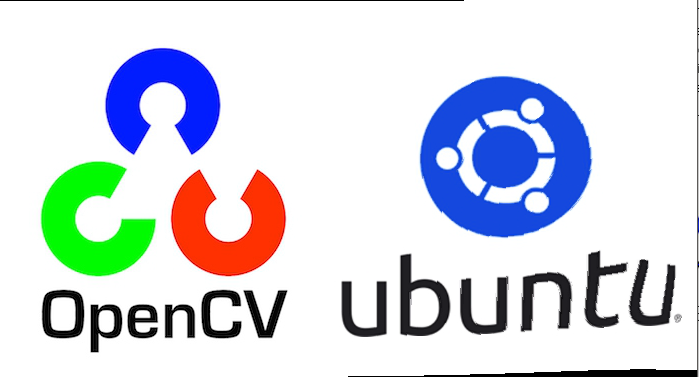
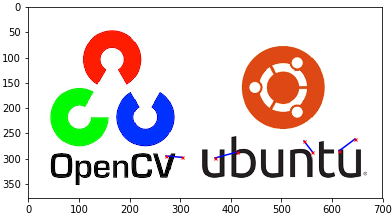
dest\_im = source\_im[source\_coords[0],source\_coords[1],:].reshape(dest\_rows,dest\_cols,3)

It would simply be backwards. First check the dest image desired coordinates, and map those coordinates to the source img. Similar to how in class we saw that we multiply the homography x the desired coordinates, in order to get the source img coordinates and then start correctly mapping the the new\_coords to coords.

4) For the fourth problem, it was very straightforward. The sample code was basically doing everything for us, except that it was only taking 2 points as input, and then it would finish. What I did was only add a while True loop that would infinitely run the program, meaning it would take two points, perform the warp, and then that image would then become the new source. The images shown below to the left display the many point warps I did, specially around the ubuntu logo, and the result to the right. How it works it applies the point warp, and then that image becomes the new source. The result is displayed each time you select two points. The time it takes to run the single point warp was 0.03 seconds, which is calculated after the user ends clicking two points.



5) The fifth one was also straightforward. I used the same code as the single point warp, but I modified a few things to it. The most important is that I wait for all the set of points until the user then notifies my program that s/he wants to exit. I made the scrool wheel on the mouse be my exit indicator. This was done using ginput and passing “mouse\_stop = 2 as a parameter. After each set of points, I would save the img to an array, until the user would exit. When the exit was detected, I would iterate through my array of dest\_images and I will simply just start displaying them all. I used this approach because a video is basically multiple images shown right after another, so I tried to mimic that same behavior. Also, I remembered one time we were asked to do something similar in one in-class quiz/exercise, and I used the same approach here. I used the in-built cv function “cv2.waitKey()” with a value of 100 to make the results display in the same window after 0.1 seconds, therefore making the transition effect as if it was a video. The elapsed time came out to be 0.93 seconds, I start the timer after the user click the scroll wheel key.

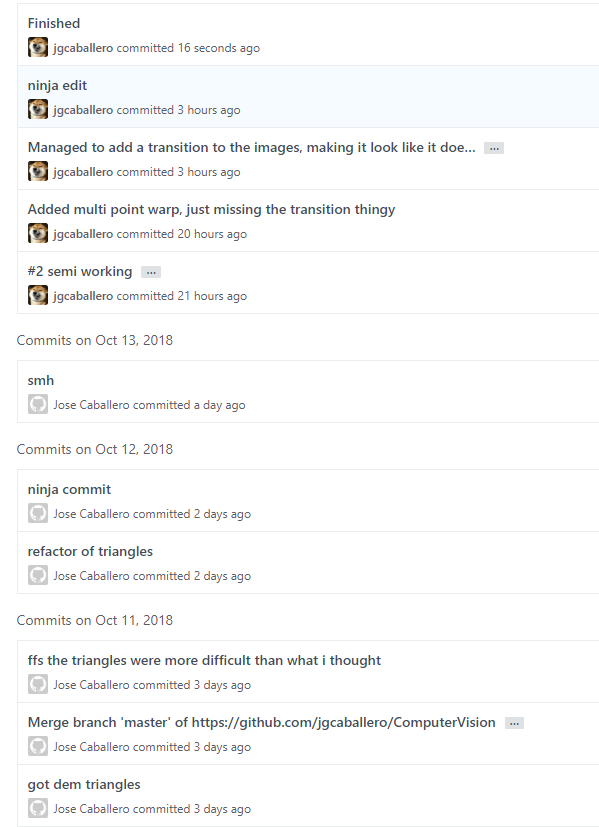


**Discussion of Results**

Everything worked as expected. The only thing I couldn’t get to properly work was the black triangles as I discussed in the experimental results in part 2. Also, the sample code that was given was somewhat difficult to even start using it. I had to do investigate how to get some things to work such as the plt.switch\_backend. I think it was discussed a little bit in class, but I still had some trouble even starting the program. As I also said in the experimental results section. Most of the sample code was done using the “pythonic ways” that you hear about so much in the web. I understood the general idea of what the code was doing, and I could understand the results, what I did not understand was what was going on behind the scenes. This made the task even more difficult. Most of the lab I felt that I knew what needed to be done, but I could not properly implement it due to the current structure of the code.

**Conclusion**

In conclusion, I think this may have been the hardest lab we have had in the semester.Unfortunately, I was unable to do some of what was asked, and this time I even started ahead of time because I figured this was a difficult topic. I wish we could review the material in class again as some of the knowledge I needed was missing. I feel conflicted on what I actually learned, because most of the stuff was already given to us, we just had to modify it. I guess I would have liked more if we had to implement the first part (the sample code) on our own, and then implement these methods as a second lab. I see the flaws in that as if some people could not implement the first part, then they would fall behind on the second lab. Parts 4 and 5 were the easiest for me because I just needed to add some logic to the program behavior, whereas part 2 and 3 actually needed to think about approaches and debugging why stuff did not work. Question 1 was very ambigious in my opinion, and had to rely on what we had done in one of the quizzes to kind of get it started. I would have made question 6 extra credit as well, as the slides did not cover the topic in detail, and I was not able to find useful information, but that might just be me.



Obligatory commit history picture.

**Appendix**

import numpy as np

import cv2

from PIL import Image

import time

import pylab as plt

plt.switch\_backend('TKAgg')

def real\_value\_indexing(img):

r0 = 1

r1 = 1

c0 = 20

c1 = 20

counter = 0

iterations = 0;

img2 = np.array([[1,2,3],

[4,5,6],

[7,8,9]])

start = time.time()

for y in range (r0, c0+1):

for x in range (r1, c1+1):

counter += img[y,x]

iterations += 1

print(img[y,x])

print('Counter', counter)

print('iterations', iterations)

print('Result', counter/iterations)

elapsed\_time = time.time()-start

print('Elapsed time: {0:.2f} '.format(elapsed\_time))

def triangle(img):

triangle2 = np.full\_like(img, 0)

row = img.shape[0]

col = img.shape[1]

slope = round(img.shape[1]/img.shape[0])

for y in range (1,row):

for x in range (1,col):

if(x == col):

triangle2[y,x] = img[y,x]

continue

else:

triangle2[y,x] = img[y,x]

img[y,x] = 0

col -= slope

return img, triangle2

def transform(H,fp):

# Transforming point fp according to H

# Convert to homogeneous coordinates if necessary

if fp.shape[0]==2:

t = np.dot(H,np.vstack((fp,np.ones(fp.shape[1]))))

else:

t = np.dot(H,fp)

return t[:2]

def forward\_mapping():

img = cv2.imread('images/banner\_small.jpg',1)

triangle1, triangle2 = triangle(img)

im1 = np.array(triangle1, dtype=np.uint8)

plt.figure(3)

plt.imshow(triangle1)

plt.show()

x2 = [0,0,im1.shape[0]-1]

y2 = [0,im1.shape[1]-1,0]

fp2 = np.vstack((x2,y2))

# im1 = np.fliplr(im1)

# im1 = np.flipud(im1)

im2 = np.array(triangle2, dtype=np.uint8)

plt.figure(1)

plt.imshow(triangle2)

plt.show()

source\_im = np.array(Image.open('images/tennis.jpg'), dtype=np.uint8)

plt.figure(2)

plt.imshow(source\_im)

plt.show()

max\_row = source\_im.shape[0]-1

max\_col = source\_im.shape[1]-1

x = [0,0,im2.shape[0]-1]

y = [0,im2.shape[1]-1,0]

fp = np.vstack((x,y))

#print("Click destination points, top-left, top-tight, and bottom-left corners")

tp = np.asarray(plt.ginput(n=3), dtype=np.float).T

tp = tp[[1,0],:]

print('fp', fp)

print('tp', tp)

#print("Click destination points, top-right, bottorm-left, and bottom-right corners")

tp2 = np.asarray(plt.ginput(n=3), dtype=np.float).T

tp2 = tp2[[1,0],:]

print('fp', fp2)

print('tp', tp2)

start = time.time()

#Using pseudoinverse

# Generating homogeneous coordinates

fph = np.vstack((fp,np.ones(fp.shape[1])))

tph = np.vstack((tp,np.ones(tp.shape[1])))

H = np.dot(tph,np.linalg.pinv(fph))

fph2 = np.vstack((fp2,np.ones(fp2.shape[1])))

tph2 = np.vstack((tp2,np.ones(tp2.shape[1])))

H2 = np.dot(tph2,np.linalg.pinv(fph2))

print('wat', (transform(H,fp)+.5).astype(np.int))

Cs = (transform(H,fp)+.5).astype(np.int)

Cs[Cs<0] = 0

Cs[0,Cs[0]>max\_row] = max\_row

Cs[1,Cs[1]>max\_col] = max\_col

Cs2 = (transform(H2,fp2)+.5).astype(np.int)

Cs2[Cs2<0] = 0

Cs2[0,Cs2[0]>max\_row] = max\_row

Cs2[1,Cs2[1]>max\_col] = max\_col

#Generating pixel coordinate locations

ind = np.arange(im2.shape[0]\*im2.shape[1])

row\_vect = ind//im2.shape[1]

col\_vect = ind%im2.shape[1]

coords = np.vstack((row\_vect,col\_vect))

new\_coords = transform(H,coords).astype(np.int)

new\_coords[new\_coords<0] = 255

new\_coords[0,new\_coords[0]>max\_row] = max\_row

new\_coords[1,new\_coords[1]>max\_col] = max\_col

target\_im = source\_im

# print('---------------------------')

# for y in range(im2.shape[0]):

# for x in range(im2.shape[1]):

# if(np.dot(im2[y,x],im2[y,x]) == 0 ):

# np.delete(im2, im2[y,x])

# print('---------------------------')

#

# print('---------------------------')

# for y in range(im1.shape[0]):

# for x in range(im1.shape[1]):

# if(np.dot(im1[y,x],im1[y,x]) == 0 ):

# np.delete(im1, im1[y,x])

# print('---------------------------')

target\_im[new\_coords[0],new\_coords[1],:] = im2[coords[0],coords[1],:]

#Generating pixel coordinate locations

ind2 = np.arange(im1.shape[0]\*im1.shape[1])

row\_vect2 = ind2//im2.shape[1]

col\_vect2 = ind2%im2.shape[1]

coords2 = np.vstack((row\_vect2,col\_vect2))

new\_coords2 = transform(H2,coords2).astype(np.int)

target\_im = source\_im

target\_im[new\_coords2[0],new\_coords2[1],:] = im1[coords2[0],coords2[1],:]

elapsed\_time = time.time()-start

print('Elapsed time: {0:.2f} '.format(elapsed\_time))

cv2.imshow('image',target\_im)

plt.figure(3)

plt.imshow(target\_im)

plt.show()

def single\_point\_warp():

source\_im = np.array(Image.open('images/opencv.jpg'), dtype=np.uint8)

plt.figure(1)

plt.imshow(source\_im)

plt.show()

print("Click source and destination of warp point")

p = np.asarray(plt.ginput(n=2, mouse\_stop=2), dtype=np.float32)

print(p)

if(p.size == 0):

#cv2.imshow('image',dest\_im)

return

print(p[0]-p[1])

plt.plot(p[:,0], p[:,1], color="blue")

plt.plot(p[0][0], p[0][1],marker='x', markersize=3, color="red")

plt.plot(p[1][0], p[1][1],marker='x', markersize=3, color="red")

#plt.show()

start = time.time()

#Generate pixels coordinates in the destination image

dest\_im = np.zeros(source\_im.shape, dtype=np.uint8)

max\_row = source\_im.shape[0]-1

max\_col = source\_im.shape[1]-1

dest\_rows = dest\_im.shape[0]

dest\_cols = dest\_im.shape[1]

#Painting outline of source image black, so out of bounds pixels can be painted black

source\_im[0]=0

source\_im[max\_row]=0

source\_im[:,0]=0

source\_im[:,max\_col]=0

#Generate pixel coordinates in the destination image

ind = np.arange(dest\_rows\*dest\_cols )

row\_vect = ind//dest\_cols

col\_vect = ind%dest\_cols

coords = np.vstack((row\_vect,col\_vect))

#Computing pixel weights, pixels close to p[1] will have higher weights

dist = np.sqrt(np.square(p[1][1] - row\_vect) + np.square(p[1][0] - col\_vect))

weight = np.exp(-dist/100) #Constant needs to be tweaked depending on image size

#Computing pixel weights, pixels close to p[1] will have higher weights

source\_coords = np.zeros(coords.shape, dtype=np.int)

disp\_r = (weight\*(p[0][1]-p[1][1])).astype(int)

disp\_c = (weight\*(p[0][0]-p[1][0])).astype(int)

source\_coords[0] = coords[0] + disp\_r

source\_coords[1] = coords[1] + disp\_c

#Fixing out-of-bounds coordinates

source\_coords[source\_coords<0] = 0

source\_coords[0,source\_coords[0]>max\_row] = max\_row

source\_coords[1,source\_coords[1]>max\_col] = max\_col

dest\_im = source\_im[source\_coords[0],source\_coords[1],:].reshape(dest\_rows,dest\_cols,3)

cv2.imshow('image',dest\_im)

elapsed\_time = time.time()-start

print('Elapsed time: {0:.2f} '.format(elapsed\_time))

def multi\_point\_warp(dest\_im):

source\_im = np.array(Image.open('images/opencv.jpg'), dtype=np.uint8)

plt.figure(1)

plt.imshow(source\_im)

plt.show()

print("Click source and destination of warp point")

p = np.asarray(plt.ginput(n=2, mouse\_stop=2), dtype=np.float32)

print(p)

if(p.size == 0):

#cv2.imshow('image',dest\_im)

return dest\_im, False

print(p[0]-p[1])

plt.plot(p[:,0], p[:,1], color="blue")

plt.plot(p[0][0], p[0][1],marker='x', markersize=3, color="red")

plt.plot(p[1][0], p[1][1],marker='x', markersize=3, color="red")

#plt.show()

start = time.time()

#Generate pixels coordinates in the destination image

dest\_im = np.zeros(source\_im.shape, dtype=np.uint8)

max\_row = source\_im.shape[0]-1

max\_col = source\_im.shape[1]-1

dest\_rows = dest\_im.shape[0]

dest\_cols = dest\_im.shape[1]

#Painting outline of source image black, so out of bounds pixels can be painted black

source\_im[0]=0

source\_im[max\_row]=0

source\_im[:,0]=0

source\_im[:,max\_col]=0

#Generate pixel coordinates in the destination image

ind = np.arange(dest\_rows\*dest\_cols )

row\_vect = ind//dest\_cols

col\_vect = ind%dest\_cols

coords = np.vstack((row\_vect,col\_vect))

#Computing pixel weights, pixels close to p[1] will have higher weights

dist = np.sqrt(np.square(p[1][1] - row\_vect) + np.square(p[1][0] - col\_vect))

weight = np.exp(-dist/100) #Constant needs to be tweaked depending on image size

#Computing pixel weights, pixels close to p[1] will have higher weights

source\_coords = np.zeros(coords.shape, dtype=np.int)

disp\_r = (weight\*(p[0][1]-p[1][1])).astype(int)

disp\_c = (weight\*(p[0][0]-p[1][0])).astype(int)

source\_coords[0] = coords[0] + disp\_r

source\_coords[1] = coords[1] + disp\_c

#Fixing out-of-bounds coordinates

source\_coords[source\_coords<0] = 0

source\_coords[0,source\_coords[0]>max\_row] = max\_row

source\_coords[1,source\_coords[1]>max\_col] = max\_col

dest\_im = source\_im[source\_coords[0],source\_coords[1],:].reshape(dest\_rows,dest\_cols,3)

# cv2.imshow('image',dest\_im)

elapsed\_time = time.time()-start

print('Elapsed time: {0:.2f} '.format(elapsed\_time))

return dest\_im, True

'''

1) real value index

'''

#img = cv2.imread('images/cat.jpg',0)

#source\_im = np.array(Image.open('images/opencv.jpg'), dtype=np.uint8)

#real\_value\_indexing(img)

'''

2) Forward Mapping

'''

#forward\_mapping()

'''

4) Single Point Warp

'''

#while True:

# single\_point\_warp()

'''

5) Multi Point Warp

'''

#dest\_im = np.array(Image.open('images/opencv.jpg'), dtype=np.uint8)

#isTrue = True

#arr = []

#while isTrue:

## for x in range(arr.shape[0]):

# dest\_im, isTrue = multi\_point\_warp(dest\_im)

# arr.append(dest\_im)

# if(not isTrue):

# for y in range(len(arr)):

# cv2.imshow('image',arr[y])

# cv2.waitKey(100)

k = cv2.waitKey(0)

if k == 27: # wait for ESC key to exit

for y in range(arr.shape[0]):

print('iteration #', y)

cv2.imshow('image',arr[y])

cv2.destroyAllWindows()

elif k == ord('s'): # wait for 's' key to save and exit0

cv2.imshow('image',dest\_im)

cv2.destroyAllWindows()