ECE 4554 / 5554: Computer Vision: Homework 3

Fall 2020

Instructions

- The assignment is due at Canvas before midnight on Oct. 25. Late submissions are allowed at the cost of 1 token per 24-hour period. A submission received only a minute after midnight will cost an additional token.
- Problems 1 through 4 are worth 10 points each. Notice that one of the problems is required for 5554 students, but is optional (extra-credit) for 4554 students. Problem 5 is worth 20 points.
- Prepare an answer sheet that contains all of your written answers in a single file named *LastName_FirstName*_HW3.pdf, using your own name. Handwritten solutions are permitted, but they must be easily legible to the grader. Do not place this pdf file inside the zip file that is described in the next bullet.
- Place all of your Python-related files (*.py, *.npy, *.png, etc.) into a single zip file named *LastName_FirstName*_HW3.zip. Upload 2 files to Canvas: this zip file and the pdf file that is described above.
- For the Python implementation problems, try to ensure that the grader can run your code "out of the box". For example, do not encode absolute path names for input files; provide relative path names, assuming that the input files reside in the same directory as the source files.
- If any plots are required, include them in your answer sheet (the pdf file). Also, your Python code must display the plots when executed.
- After you have submitted to Canvas, it is your responsibility to download the files that you submitted and verify that they are correct and complete. *The files that you submit to Canvas are the files that will be graded.*

Problem 1. We have discussed the 2-dimensional *affine transformation*, which maps a point (*x, y*) to a new location (x', y') in the plane. The transformation can be represented using the following equation, where 6 constant parameters (shown here as a_1 through a_6) completely specify the transformation:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (1)$$

a) Consider the problem that you are given 3 pairs of points, and you want to use those points to determine the parameters a_1 through a_6. For example, assume that the following correspondences are known:

$$(x_1', y_1') \leftrightarrow (x_1, y_1)$$
$$(x_2', y_2') \leftrightarrow (x_2, y_2)$$
$$(x_3', y_3') \leftrightarrow (x_3, y_3)$$

Show that it is possible to write one matrix equation that represents the (linear) relationship between all of these scalar values. (*Hint*: create a 6x1 vector that contains the individual affine parameters only. Create another 6x1 vector that contains the 6 terms x_1', y_1', …, x_3', y_3'. Then create a single square matrix that represents the mapping for all 3 pairs of points.) For this part, you do not need solve for a_1 through a_6, although a matrix solver such as NumPy could do this easily.

b) Intuitively, explain why at least 3 point correspondences are needed in order to solve for the affine transformation parameters.

c) For this part, it is acceptable to use a matrix solver such as NumPy. Starting with your answer to part (a), find the affine transformation parameters a_1 through a_6 for the following set of corresponding points:

$$(x_i', y_i') \leftrightarrow (x_i, y_i)$$
$$(5, 4) \leftrightarrow (0, 0)$$
$$(7, 4) \leftrightarrow (1, 0)$$
$$(6, 5) \leftrightarrow (0, 1)$$

If you use NumPy or some other solver, provide your code as part of your solution.

a.)

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a_1x + a_2y + a_3 \\ a_4x + a_5y + a_6 \\ 0 + 0 + 1 \end{bmatrix} \Rightarrow \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_1x + a_2y + a_3 \\ a_4x + a_5y + a_6 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} & & \\ & & \\ & & \\ & & \\ & & \\ & & \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \end{bmatrix} = \begin{bmatrix} x_1' \\ y_1' \\ x_2' \\ y_2' \\ x_3' \\ y_3' \end{bmatrix}$$

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_2 & y_2 & 1 \\ x_3 & y_3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_3 & y_3 & 1 \end{bmatrix}$$

b) affine transformation is a function that maps from an affine space to another and preserves structure. They can change the orientation, size, or position of an object.

An affine transformation is composed of various translations that modify the object's position in an image

There's also scaling operations, rotation, and shear mapping.

The use of Homogeneous coordinates for affine transformation allows us to use the mathematical properties of matrics to perform these transformations in projective space.

c.)

```
import numpy as np
from numpy.linalg import inv
#Ax = N
A = np.empty
N = np.array([  [5],
                [4],
                [7],
                [4],
                [6],
                [5] ])

x = np.array([  [0, 0, 1, 0, 0, 0],
                [0, 0, 0, 0, 0, 1],
                [1, 0, 1, 0, 0, 0],
                [0, 0, 0, 1, 0, 1],
                [0, 1, 1, 0, 0, 0],
                [0, 0, 0, 0, 1, 1] ])

A = np.dot( inv(x), N)

print(A)
print("Matrix A is of shape:" + str(A.shape) )
```

```
PS C:\Users\jrsav\Documents\Fall Semester 2020\Computer Vision\HW3> python .\HW3_P1.py
[[2.]
 [1.]
 [5.]
 [0.]
 [1.]
 [4.]]
Matrix A is of shape:(6, 1)
```

Problem 2. The 2D *planar perspective transformation*, also known as 2D *homography*, also maps a point (x, y) to a new location (x', y') in the plane. This transformation can be represented using the following equation, where 8 constant parameters a_1 through a_8 completely specify the transformation. Recall that the homogeneous scale factor s is eliminated when solving for (x', y').

$$\begin{bmatrix} s\,x' \\ s\,y' \\ s \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{2}$$

(For some situations, it is better to replace 1 in the homography matrix by a 9th parameter, a_9. Let's ignore those cases for this problem.)

a) Consider the problem that you are given 4 pairs of points, and you want to use those points to determine the parameters a_1 through a_8. For example, assume that the following correspondences are known:

$$\begin{aligned} (x_1', y_1') &\leftrightarrow (x_1, y_1) \\ (x_2', y_2') &\leftrightarrow (x_2, y_2) \\ (x_3', y_3') &\leftrightarrow (x_3, y_3) \\ (x_4', y_4') &\leftrightarrow (x_4, y_4) \end{aligned}$$

Show that it is possible to write one matrix equation that represents the relationship between these all of these scalar values. (*Hint*: create an 8x1 vector that contains the individual homography parameters only. Create another 8x1 vector that contains the 8 terms x_1', y_1', …, x_4', y_4'. Then create a single square matrix that represents the mapping for all 4 pairs of points.) For this part, you do not need solve for a_1 through a_8, although a matrix solver such as NumPy could do this.

b) Intuitively, explain why at least 4 point correspondences are needed in order to solve for the homography parameters.

c) For this part, it is acceptable to use a matrix solver such as NumPy. Find the homography parameters a_1 through a_8 for the following set of corresponding points:

$$\begin{aligned} (x_i', y_i') &\leftrightarrow (x_i, y_i) \\ (5, 4) &\leftrightarrow (0, 0) \\ (7, 4) &\leftrightarrow (1, 0) \\ (7, 5) &\leftrightarrow (1, 1) \\ (6, 6) &\leftrightarrow (0, 1) \end{aligned}$$

If you use NumPy or some other solver, provide your code as part of your solution.

a.)

$$\begin{bmatrix} sx' \\ sy' \\ s \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} sx' \\ sy' \\ s \end{bmatrix} = \begin{bmatrix} a_1x + a_2y + a_3 \\ a_4x + a_5y + a_6 \\ a_7x + a_8y + 1 \end{bmatrix}$$

$$x' = \frac{a_1x + a_2y + a_3}{s} \qquad y' = \frac{a_4x + a_5y + a_6}{s} \qquad s = a_7x + a_8y + 1$$

$$x' = \frac{a_1x + a_2y + a_3}{a_7x + a_8y + 1} \qquad y' = \frac{a_4x + a_5y + a_6}{a_7x + a_8y + 1}$$

$$x'a_7x + x'a_8y + x' = a_1x + a_2y + a_3 \Rightarrow x' = a_1x + a_2y + a_3 - x'a_7x - x'a_8y$$

$$y'a_7x + y'a_8y + y' = a_4x + a_5y + a_6 \Rightarrow y' = a_4x + a_5y + a_6 - y'a_7x + y'a_8y$$

$$\begin{bmatrix} \\ \\ \\ \\ \\ \\ \\ \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \\ a_8 \end{bmatrix} = \begin{bmatrix} x_1' \\ y_1' \\ x_2' \\ y_2' \\ x_3' \\ y_3' \\ x_4' \\ y_4' \end{bmatrix}$$

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1'x_1 & -x_1'y_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y_1'x_1 & -y_1'y_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2'x_2 & -x_2'y_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -y_2'x_2 & -y_2'y_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3'x_3 & -x_3'y_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -y_3'x_3 & -y_3'y_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4'x_4 & -x_4'y_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -y_4'x_4 & -y_4'y_4 \end{bmatrix}$$

1a.)

Homography is used to map to the other and if homography exists, then it needs a minimum 4 point to specify it precisely. If less than 4 point correspondence is used, then there's no linear map that relates the parameters. Each point correspondence generates two linear equations for a and are used to multiply out in order to solve for a.

c.)

```
import numpy as np
from numpy.linalg import inv

x1_p = 5
y1_p = 4
x2_p = 7
y2_p = 4
x3_p = 7
y3_p = 5
x4_p = 6
y4_p = 6

x1 = 0
y1 = 0
x2 = 1
y2 = 0
x3 = 1
y3 = 1
x4 = 0
y4 = 1
#Ax = N
A = np.empty
N = np.array([  [x1_p],
                [y1_p],
                [x2_p],
                [y2_p],
                [x3_p],
                [y3_p],
                [x4_p],
                [y4_p],])

x = np.array([  [x1, y1, 1, 0, 0, 0, -x1_p*x1, -x1_p*y1],
                [0, 0, 0, x1, y1, 1, -y1_p*x1, -y1_p*y1],
                [x2, y2, 1, 0, 0, 0, -x2_p*x2, -x2_p*y2],
                [0, 0, 0, x2, y2, 1, -y2_p*x2, -y2_p*y2],
                [x3, y3, 1, 0, 0, 0, -x3_p*x3, -x3_p*y3],
                [0, 0, 0, x3, y3, 1, -y3_p*x3, -y3_p*y3],
                [x4, y4, 1, 0, 0, 0, -x4_p*x4, -x4_p*y4],
                [0, 0, 0, x4, y4, 1, -y4_p*x4, -y4_p*y4], ])

A = np.dot( inv(x), N)

print(A)
print("Matrix A is of shape:" + str(A.shape) )
```

```
PS C:\Users\jrsav\Documents\Fall Semester 2020\Computer Vision\HW3> python .\HW3_P2.py
[[16.]
 [ 7.]
 [ 5.]
 [ 8.]
 [ 8.]
 [ 4.]
 [ 2.]
 [ 1.]]
Matrix A is of shape:(8, 1)
```

Problem 3. Let $\|\boldsymbol{x}\|$ represent the L_2 norm for any n-dimensional vector $\boldsymbol{x}$. This norm is also called the Euclidian norm, and is defined as

$$\|\boldsymbol{x}\| = \sqrt{\sum_{i=1}^{n} x_i^2}$$

In this equation, each scalar value x_i is a component of $\boldsymbol{x}$.

Prove that $\|\boldsymbol{x} + \boldsymbol{y}\| \leq \|\boldsymbol{x}\| + \|\boldsymbol{y}\|$ for all $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$. (This relation is called the triangle inequality.)

For $x, y \in \mathbb{R}^n$

$$\|x\| = \sqrt{\sum_{i=1}^{n} x_i^2} \qquad \|y\| = \sqrt{\sum_{i=1}^{n} y_i^2}$$

$$\|x+y\| = \sqrt{\sum_{i=1}^{n} (x_i + y_i)^2} \Rightarrow \|x+y\|^2 = \left(\sqrt{\sum_{i=1}^{n} (x_i+y_i)^2}\right)^2 = \sum_{i=1}^{n} (x_i+y_i)^2$$

$$\Rightarrow \|x+y\|^2 = \sum_{i=1}^{n} x_i^2 + \sum_{i=1}^{n} y_i^2 + 2\sum_{i=1}^{n} x_i y_i$$

Now

$$\sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + \dots x_n y_n \leq x_1 y_1 + x_1 y_2 + \dots x_1 y_n +$$
$$x_2 y_1 + x_2 y_2 + \dots x_2 y_n +$$
$$x_n y_1 + x_n y_2 + \dots x_n y_n$$
$$= x_1 \sum_{i=1}^{n} y_i + x_2 \sum_{i=1}^{n} y_i + \dots x_n \sum_{i=1}^{n} y_i$$
$$= (x_1 + x_2 + \dots x_n) \sum_{i=1}^{n} y_i$$
$$= \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i$$

Hence

$$\sum_{i=1}^{n} x_i y_i \leq \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i$$

Using: $\|x+y\|^2 = \sum_{i=1}^{n} x_i^2 + \sum_{i=1}^{n} y_i^2 + 2\sum_{i=1}^{n} x_i y_i$

$$\leq \sum_{i=1}^{n} x_i^2 + \sum_{i=1}^{n} y_i^2 + 2\sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i$$

$$= \|x\|^2 + \|y\|^2 + 2\sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i$$

$$\Rightarrow \|x+y\|^2 \leq \|x\|^2 + \|y\|^2 + 2\sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i$$

From Hölder's Inequality

$$\sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i \leq \left(\sum_{i=1}^{n} x_i\right)^{1/2} \left(\sum_{i=1}^{n} y_i^2\right)^{1/2} = \|x\| \, \|y\|$$

Thus,

$$\|x+y\|^2 \leq \|x\|^2 + \|y\|^2 + 2\sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i$$

$$\leq \|x\|^2 + \|y\|^2 + 2\|x\| \, \|y\|$$

$$= (\|x\| + \|y\|)^2$$

$$\Rightarrow \|x+y\|^2 \leq (\|x\| + \|y\|)^2$$

$$\Rightarrow \|x+y\| \leq \|x\| + \|y\| \qquad \forall \; x, y \in \mathbb{R}^n$$

Problem 4. (For 5554 students, this problem is required. For 4554 students, this problem will count as extra credit.) Using the definition of a derivative, show analytically that the following kernel is a good discrete approximation of the second derivative, $\frac{\partial^2 I}{\partial x^2}$.

1	-2	1

Problem 5. (20 points.) Write a Python/OpenCV program that will automatically create an image mosaic I_{out} from two input images, I_1 and I_2. Name your script `P5.py`. Paste this script into your answer sheet, and also submit the source file.

Your program should perform the following steps:

- Load two images. Two examples are provided to you in files `wall1.png` and `wall2.png`, and you should hard-code these file names into your program.
- Detect keypoints and create feature descriptors for both of the input images. You are welcome to use the ORB descriptor[1,2] that was discussed in class recently, or you may use a different descriptor such as SIFT. You are allowed to use "canned" library functions to detect keypoints and create descriptors.
- Using the keypoints that were detected in the previous step, determine a good set of matches (corresponding pairs of points) between the two images. You are allowed to use library functions for the matching step. For guidance, please refer to OpenCV tutorials.[3]
- Using results from the previous step, compute a homography matrix `H` that will map points from one image onto corresponding points in the other image. Use `numpy.save()` to store this matrix in file `outputP5H.npy`. Submit this file with your solutions. For this step your must write your own function `findHomography`. For inspiration, you may refer to OpenCV's version of this function.[4] There are two major issues to consider: 1) Should your function use only 4 points, as you did for Problem 2 of this assignment? A common alternative is to use the method of least squares, which allows more than the minimum of 4 points. 2) Consider the effect of incorrect matches, which we can call "outliers." A common means to identify and ignore outliers is known as RANSAC. We will discuss both techniques, least-squares estimation and RANSAC, in an upcoming lecture.
- Using the homography from the previous step, transform (warp) one of the input images onto the frame of reference of the other input image. You must write your own function `homography_warp` that uses your matrix `H`. As you learned in Homework 2, it is best to iterate over the *output* image array during the warping procedure.
- Using results from the previous step, create a composite output image by combining (blending) pixel values from the two images. To do this, create a new image that is large enough to hold both (registered) views, with all pixels initialized to black. Then copy both input images (after one of them has been warped) into the new image, averaging the pixel values whenever 2 pixels overlap. Don't worry about artifacts that result at the boundaries of the original images. Store the result in file `outputP5wall.png`. Paste this output image into your answer sheet, and also submit the image file.

In addition to the Great Wall images that were provided with this assignment, also demonstrate your program with at least one more pair of images of your own. Paste these images into your solution (pdf) file. You are also welcome to submit your images separately (png, jpg, etc.), but this is not required.

Notes:

- For this problem, do not use any OpenCV/NumPy/etc. functions except for the following: for loading and saving image files, for basic math and matrix operations, for keypoint detection, and for keypoint matching.

---

[1] https://scikit-image.org/docs/dev/auto_examples/features_detection/plot_orb.html#sphx-glr-download-auto-examples-features-detection-plot-orb-py
[2] https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_orb/py_orb.html
[3] https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_matcher/py_matcher.html
[4] https://docs.opencv.org/master/d9/d0c/group__calib3d.html#ga4b3841447530523e5272ec05c5d1e411

- If you find a library function that computes a homography from pairs of points, you are not allowed to use it. Instead, you must write your own code to compute the homography, possibly using matrix operations from NumPy.
- For computing the homography, it is usually a good idea to select corresponding points that are widely distributed within the images. Don't select points that are nearly collinear. You may use this sort of heuristic in your code for selecting points.
- It can be helpful when debugging to map and plot the 4 image corners along with the selected points from the source image onto the destination via H.
- For this assignment, you need to work with color images. For geometric operations, one approach is to process each color channel separately and then stack them together to form the result.
- If you take images with your own camera to test your program, it is best to keep the camera at the same position. Between capturing separate images, rotate the camera about its point of projection.

```
import numpy as np
import matplotlib.pyplot as plt
import cv2
import math
import random

#Calculate the geometric distance between estimated points and original points
def geometricDistance(correspondence, h):

    p1 = np.transpose(np.matrix([correspondence[0].item(0), correspondence[0].
item(1), 1]))
    estimatep2 = np.dot(h, p1)
    estimatep2 = (1/estimatep2.item(2))*estimatep2

    p2 = np.transpose(np.matrix([correspondence[0].item(2), correspondence[0].
item(3), 1]))
    error = p2 - estimatep2
    return np.linalg.norm(error)


def ransac(corr, iterations):
    maxInliers = []
    finalH = None
    for i in range(iterations):
        #find 4 random points to calculate a homography
        corr1 = corr[random.randrange(0, len(corr))]
        corr2 = corr[random.randrange(0, len(corr))]
        randomFour = np.vstack((corr1, corr2))
        corr3 = corr[random.randrange(0, len(corr))]
        randomFour = np.vstack((randomFour, corr3))
        corr4 = corr[random.randrange(0, len(corr))]
        randomFour = np.vstack((randomFour, corr4))

        #call the homography function on those points
        H = findHomography(randomFour)

        inliers = []

        for i in range(len(corr)):
            d = geometricDistance(corr[i], H)
            if d < 5:
                inliers.append(corr[i])

        if len(inliers) > len(maxInliers):
            maxInliers = inliers
            finalH = H
        #print ("Corr size: ", len(corr), " NumInliers: ", len(inliers), "Max
inliers: ", len(maxInliers))

        threshold = 3
```

```
        if len(maxInliers) > (len(corr)*threshold):
            break
    return finalH, maxInliers

def findHomography(correspondences):
    #loop through correspondences and create assemble matrix
    aList = []
    for corr in correspondences:
        p1 = np.matrix([corr.item(0), corr.item(1), 1])
        p2 = np.matrix([corr.item(2), corr.item(3), 1])

        a1 = [-p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -
p2.item(2) * p1.item(2), 0, 0, 0,
              p2.item(0) * p1.item(0), p2.item(0) * p1.item(1), p2.item(0) * p
1.item(2)]

        a2 = [0, 0, 0, -p2.item(2) * p1.item(0), -p2.item(2) * p1.item(1), -
p2.item(2) * p1.item(2),
              p2.item(1) * p1.item(0), p2.item(1) * p1.item(1), p2.item(1) * p
1.item(2)]

        aList.append(a1)
        aList.append(a2)

    matrixA = np.asarray(aList)

    #svd composition
    u, s, v = np.linalg.svd(matrixA)

    #reshape the min singular value into a 3 by 3 matrix
    H = np.reshape(v[8], (3, 3))

    #normalize and now we have h
    H = (1/H.item(8)) * H
    return H

def homography_warp(image, image2, H):
    height = len(image2)
    width = len(image2[0])
    new_height = len(image) + len(image2)
    new_width = len(image[0]) + len(image2[0])

    output = np.zeros((new_height, new_width, 3), dtype="uint8")

    for y_counter in range(0, len(image)):
        for x_counter in range(0, len(image[0])):
            curr_pixel = [x_counter,y_counter,1]

            if(curr_pixel[0] < height and curr_pixel[1] < width and curr_pixel
[0] > 0 and curr_pixel[1] > 0):
```

```
                output[x_counter, y_counter] = image[curr_pixel[0], curr_pixel
[1]]

    # H = np.linalg.inv(H)

    for y in range(output.shape[0]):
        for x in range(output.shape[1]):
            src = H.dot([x, y, 1])
            src = (src[:2] / src[2]).astype(int)

            if(0 <= src[0] < image2.shape[1]) and (0 <= src[1] < image2.shape[
0]):
                val = image2[src[1], src[0], :]
                if y < image.shape[0] and x < image.shape[1]:

                    val = (val.astype(int) + image[y, x, :].astype(int)) / 2
                output[y, x, :] = val

    return output

def getCorrespodences(I1, I2):

    # Initialize the ORB detector algorithm
    orb = cv2.ORB_create()

    # Now detect the keypoints and compute
    # the descriptors for the I1 image
    # and I2 image
    I1Keypoints, I1Descriptors = orb.detectAndCompute(I1,None)
    I2Keypoints, I2Descriptors = orb.detectAndCompute(I2,None)

    # Initialize the Matcher for matching
    # the keypoints and then match the
    # keypoints
    matcher = cv2.BFMatcher(cv2.NORM_HAMMING, True)
    matches = matcher.match(I1Descriptors,I2Descriptors)
    matches = sorted(matches, key = lambda x:x.distance)
    matches = matches[:20]

    correspondenceList = []
    keypoints = [I1Keypoints, I2Keypoints]

    #print ('#matches:', len(matches))

    for match in matches:
        (x1, y1) = keypoints[0][match.queryIdx].pt
        (x2, y2) = keypoints[1][match.trainIdx].pt
        correspondenceList.append([x1, y1, x2, y2])

    corrs = np.matrix(correspondenceList)
```

```
    return corrs

#Output mosaic image is Iout
I1 = cv2.imread('wall1.png')
I2 = cv2.imread('wall2.png')

corrs1 = getCorrespodences(I1, I2)

iterations = 1000
#run ransac
finalH, maxInliers = ransac(corrs1, iterations)

#print("Max num of inliers: ", len(maxInliers))
#print("Final Homography with a shape of : ", finalH.shape)
#print(finalH)

Iout = homography_warp(I1, I2,finalH)


#Now for my image tests

myImg1 = cv2.imread('OutsideRight.png')
myImg2 = cv2.imread('OutsideMiddle.png')

corrs2 = getCorrespodences(myImg1, myImg2)

myH, maxInliers = ransac(corrs2, iterations)

Iout2 = homography_warp(myImg1, myImg2, myH)

#save the npy
np.save('outputP5H.npy', finalH)

cv2.imshow("The mosiac image", Iout)
cv2.imwrite('ouputP5wall.png', Iout)

cv2.imshow("Tested transformed image", Iout2)
cv2.imwrite('ouputP5myImages.png', Iout2)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

The wall output:

These are the 2 images I used for my test:

The output: