Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW05

1) **Least squares method to estimate homographies**

The linear least squares method is used to estimate homographies between two images given one or more pairs of corresponding points. If (x,y,w) and (x',y',w') are the corresponding points in the two images, then

$$A = \begin{bmatrix} 0 & 0 & 0 & -w'x & -w'y & -w'w & y'x & y'y & y'w \\ w'x & w'y & w'w & 0 & 0 & 0 & -x'x & -x'y & -x'w \end{bmatrix}$$

If there are multiple correspondences, they are vertically stacked to create a 2N x 9 matrix A, where N is the number of correspondences. Our objective is to minimize ||Ah||, where h is the homography we are solving for, represented as a 9 x 1 matrix. We constrain ||h|| = 1 to avoid the trivial solution where h is a zero matrix. The solution is given by that eigenvector of A^TA which corresponds to its smallest eigenvalue. The singular value decomposition gives us U, D and V^T. We take the last column of V as the solution homography h.

2) **RANSAC algorithm**

The calculated correspondences using NCC or SSD are likely to have some false correspondences and some noisy correspondences. The Random Sample Consensus algorithm is used to filter out the false correspondences, so that we are left only with actual correspondences which may be noisy.

The algorithm randomly chooses n correspondences and uses linear least squares algorithm to find the homography H. The remaining correspondences x' = Hx are found using this H, and are compared with the actual correspondences that we know. If the Euclidean distance delta is below a certain threshold, then the correspondence is appended to the inlier set.

The above process is repeated N times, and the largest inlier set of correspondences is used to calculate the homography using linear least squares method. Typically we would want the size of the inlier set to be above a certain fraction of the total number of correspondences.

3) **Levenberg-Marquardt algorithm**

LM is a nonlinear least squares method to refine a homography which has been calculated using a linear least squares method. It combines two algorithms: Gradient Descent (GD) and Gauss-Newton (GN).

In Gradient Descent, a descent is made from p_k to p_{k+1} using the formula:

$p_{k+1} = p_k + 2\text{gamma} * J_f^T(p_k) * \text{epsilon}(p_k)$,

where $\text{epsilon}(p_k) = X - f(p)$

and $J_f^T(p_k)$ is the negative of the Jacobian of $\text{epsilon}(p_k)$

In Gauss Newton, the solution is given by

$X = f(p) + J_f(p) * \text{rho}_p$,

Where $J_f(p)$ is the Jacobian of p,

And rho_p is given by solving $J_f(p) * \text{rho}_p = \text{epsilon}(p)$

LM uses the following equation for updates:

$p_{k+1} = p_k + (J_f^T J_f + \text{mu} * I)^{-1} J_f^T \text{epsilon}(p_k)$

mu is called the damping coefficient, a bigger value of which means that the algorithm is leaning more towards GD than towards GN. The initial mu is calculated as:

$\text{mu}_0 = \text{tau} * \max\{\text{diag}(J_f^T J_f)\}$, $0 < \text{tau} <= 1$

Subsequent values of mu are calculated using the following formula:

Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW05

$Mu_{k+1} = mu_k * max\{0.33, 1 - (2*rho^{LM}_{k+1} - 1)^3\}$

4) Image mosaicing steps

a) Five consecutive pictures with significant overlap are taken. Corner points are detected using Harris corner detector.

b) Correspondences are found between these two images using the NCC metric.

c) RANSAC outlier eliminator is used to find true corner points, using which a homography is calculated using linear least squares method.

d) This homography is used as the starting point in the Levenberg-Marquardt algorithm to calculate an even more refined homography. Scipy.optimizer.root implementation is used. This is done for consecutive images.

e) $H13 = H23*H12$
$H43 = H34^{-1}$
$H53 = (H34)^{-1}*(H45)^{-1}$
Where '*' is matrix multiplication.
H<n>3 was applied to image <n>, and stored in a blank image, where n belongs to {1,2,3,4,5}. Application of the homography was carried out in ascending order of n. The resulting image is the output mosaic.

5) Experimental Results

Starting images

Image 1

Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW05

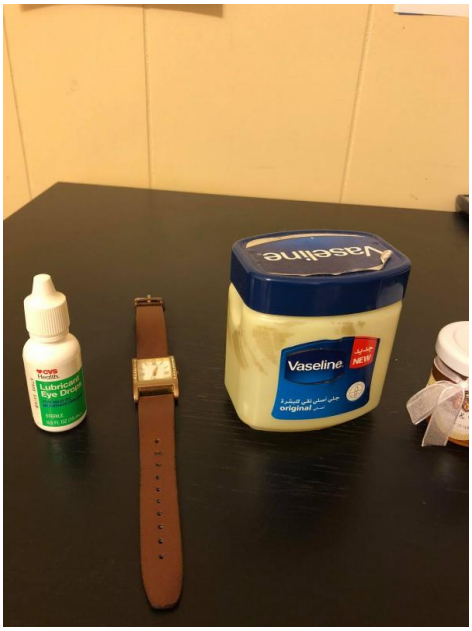


Image 2

Image 3

Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW05

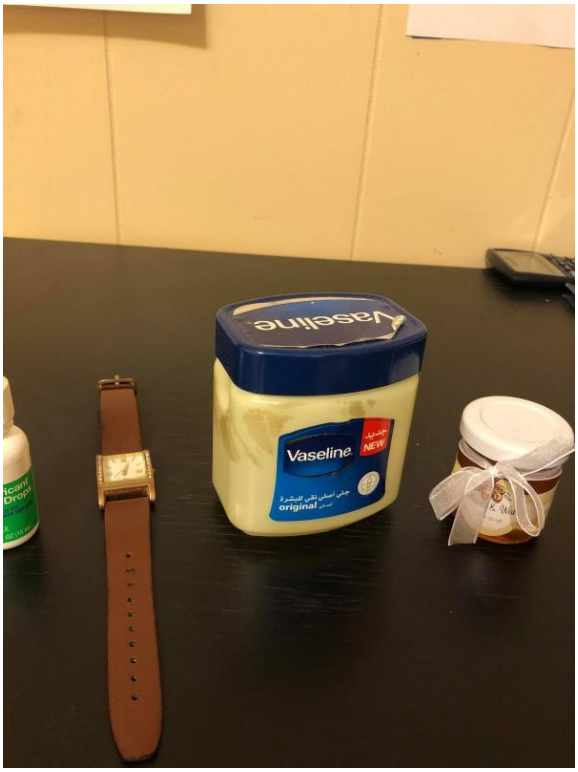
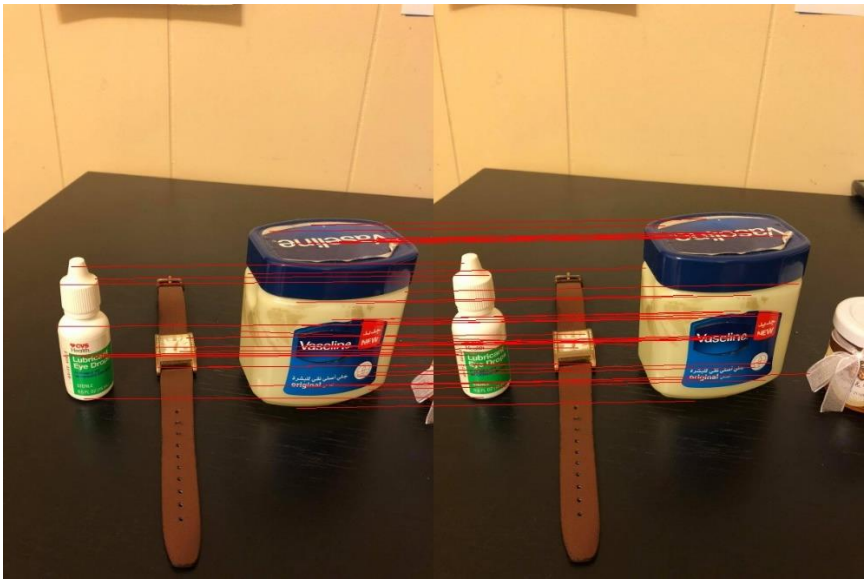


Image 4

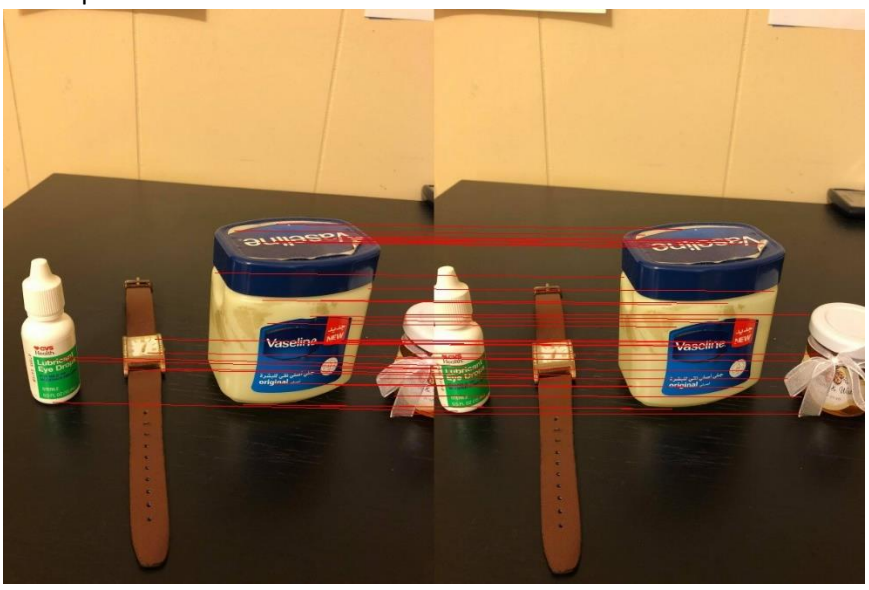
Image 5

Extracted correspondences between sets of adjacent images

Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW05



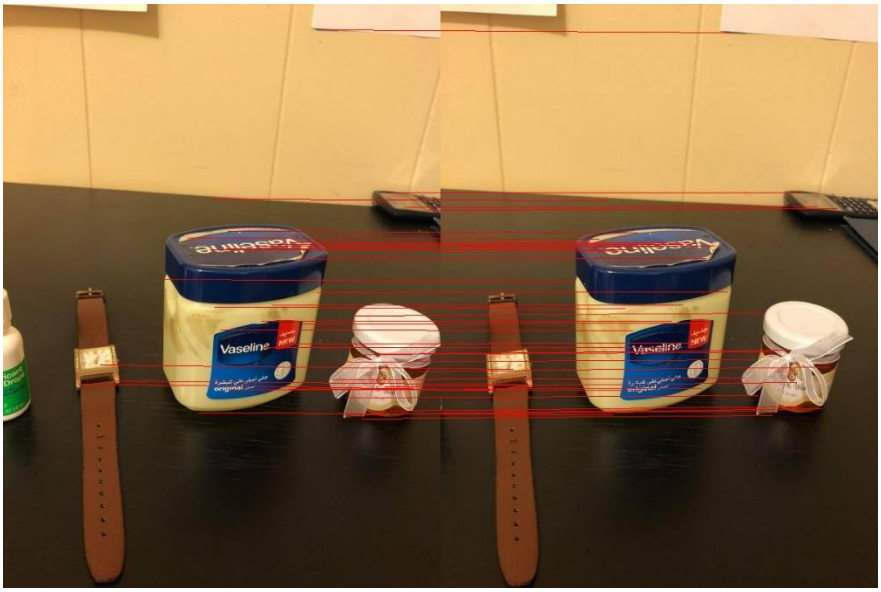
Correspondences between 1 and 2



Correspondences between 2 and 3

Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW05

Correspondences between 3 and 4

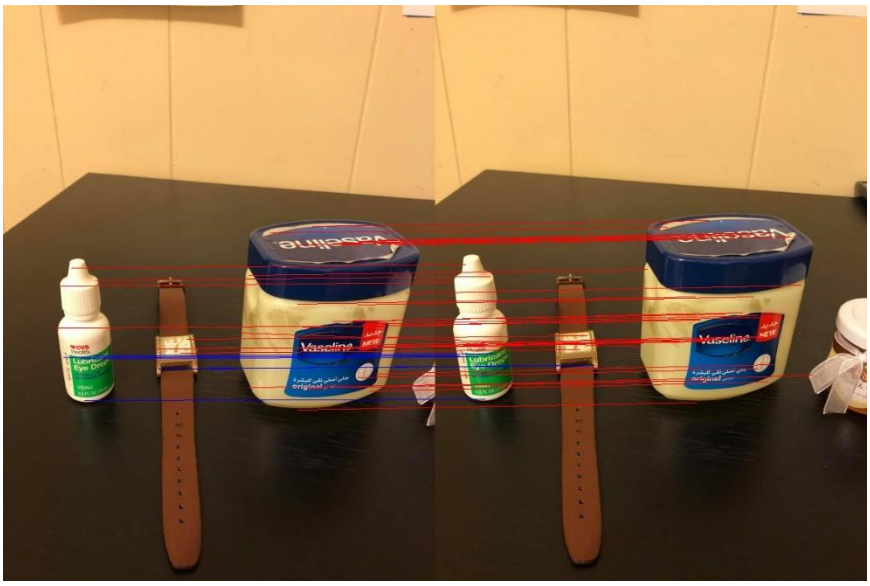


Correspondences between 4 and 5

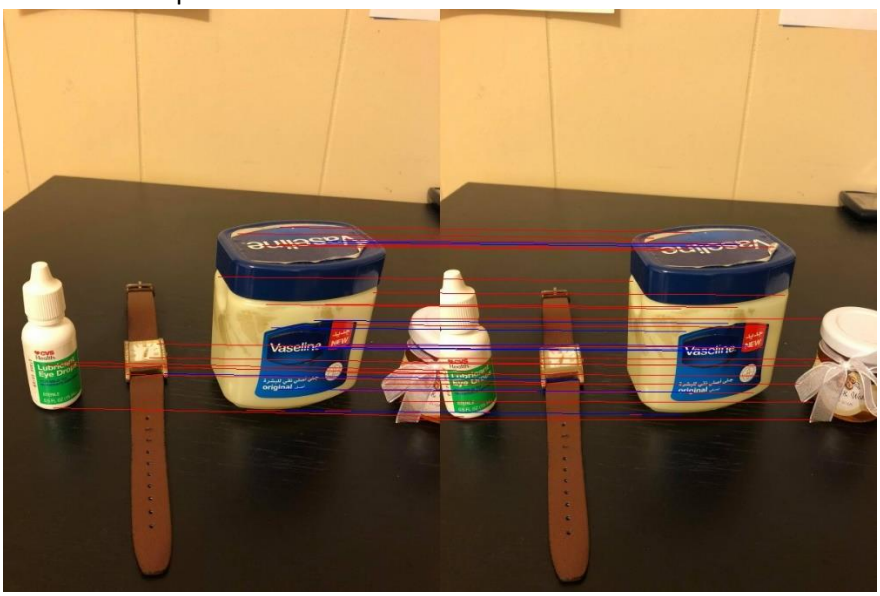
Outliers and selected inliers

The blue lines are inliers, the red lines are outliers

Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW05

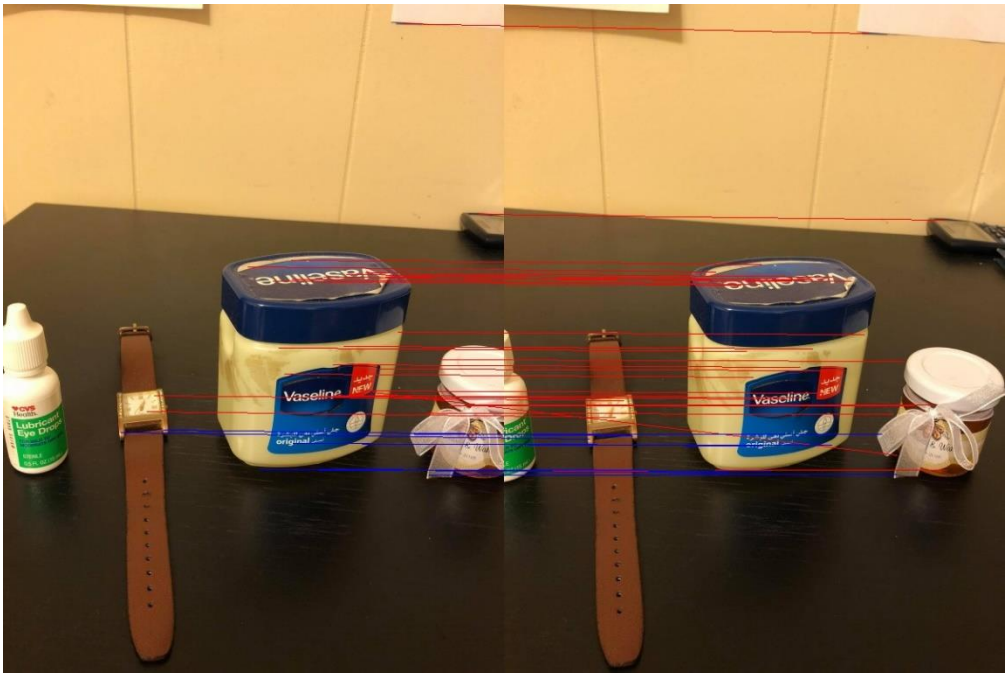


Filtered correspondences between 1 and 2

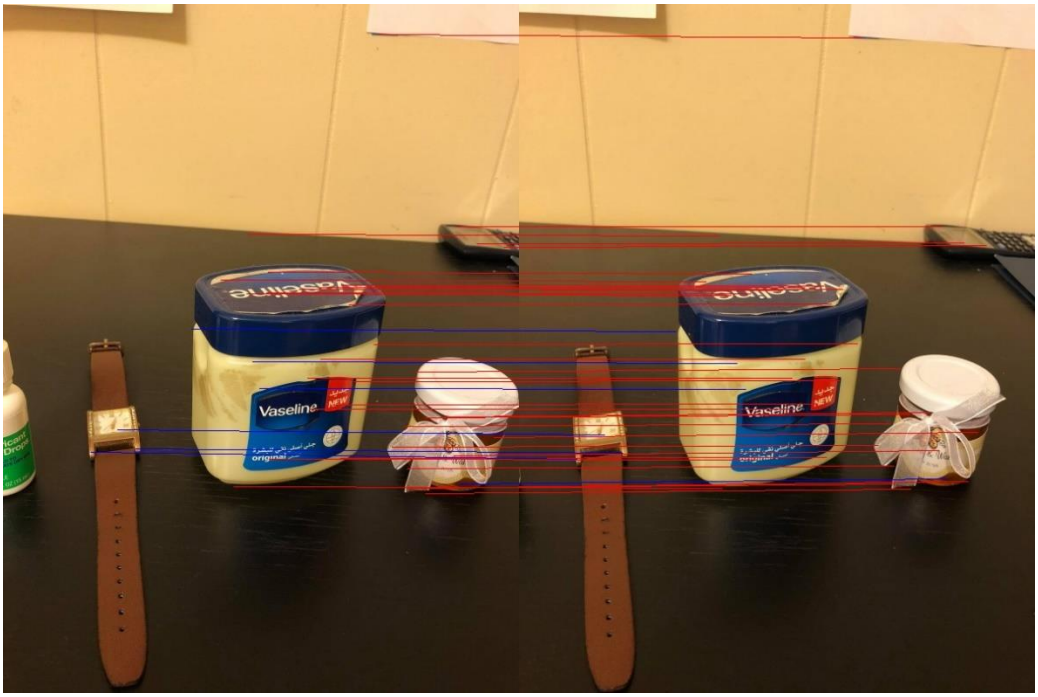


Filtered correspondences between 2 and 3

Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW05



Filtered correspondences between 3 and 4



Filtered correspondences between 4 and 5

Stitched images

Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW05

Images 1 and 2 stitched

Images 2 and 3 stitched

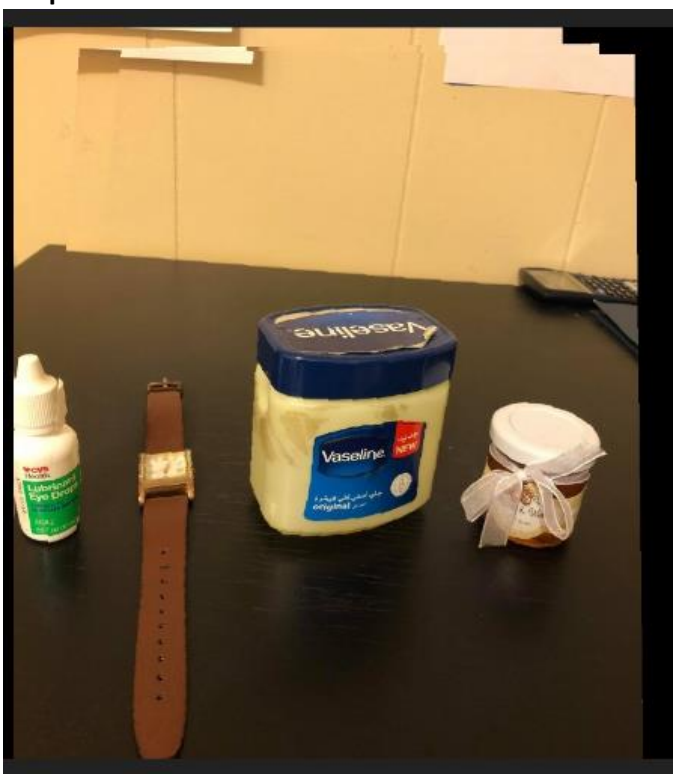
Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW05

Images 3 and 4 stitched

Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW05

Images 4 and 5 stitched

Output mosaic



Output mosaic

6) Parameters chosen for experiments

I chose to scale down the images to the standard low resolution of 600 x 800 to reduce computation time.

The Harris corner detector was used with a sigma of 2.1, which made the window size 11.

For RANSAC, number of trials N was 30, and number of randomly selected correspondences n was also 8. A threshold of 50 pixels was tolerated before a correspondence was rejected from the inlier set.

7) Source code

```
import numpy as np

import cv2, math, scipy.signal

import matplotlib.pyplot as plt

import scipy


#path = '../Users/rmahfuz/Desktop/661/HW05/'
```

Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW05

```
path = ''


#-------------------------------------------------------------------------------------------------------------------------

def harris(img, sigma):
    #finding smallest even integer greater than 4*sigma (size of haar filter)
    size = int(4*sigma) + 1
    if size % 2 != 0:
        size += 1
    assert size % 2 == 0
    haar_x = np.hstack([-1*np.ones((size, int(size/2))), np.ones((size, int(size/2)))])
    haar_y = np.vstack([np.ones((int(size/2), size)), -1*np.ones((int(size/2), size))])
    img_x = scipy.signal.convolve2d(img, haar_x, mode = 'same')
    img_y = scipy.signal.convolve2d(img, haar_y, mode = 'same')
    img_x = np.divide(np.subtract(img_x, np.min(img_x)), np.subtract(np.max(img_x), np.min(img_x)))
    img_y = np.divide(np.subtract(img_y, np.min(img_y)), np.subtract(np.max(img_y), np.min(img_y)))
    print('img_x.shape = ', img_x.shape)
    d_x2 = np.square(img_x)
    d_y2 = np.square(img_y)
    d_xy = np.multiply(img_x, img_y)
    #print(img_x.shape)
    win_size = int(5*sigma)
    if win_size % 2 == 0: #ensuring that window size is even
        win_size += 1
    corner_pts = []
    ratio_store = np.zeros((img.shape[0], img.shape[1]))
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            C = np.zeros((2,2))
            for k in range(i - int(win_size/2), i+int(win_size/2)+1):
```

```
            for l in range(j-int(win_size/2), j+int(win_size/2)+1):
                if k < img.shape[0] and l < img.shape[1]:
                    C[0][0] += d_x2[k][l]
                    C[0][1] += d_xy[k][l]
                    C[1][0] += d_xy[k][l]
                    C[1][1] += d_y2[k][l]
        if np.linalg.matrix_rank(C) == 2:
            ratio_store[i][j] = np.linalg.det(C)/np.square(np.matrix.trace(C))
        else:
            ratio_store[i][j] = -1
    print('finished calculating Cs')
    fil_win = 29 #filter window size
    #thresh = 2*np.mean(ratio_store)
    thresh = 0.01
    #Filtering out non-local maxima
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            x_min = max(0, i-int(fil_win/2))
            x_max = min(i+int(fil_win/2)+1, img.shape[0])
            y_min = max(0,j-int(fil_win/2))
            y_max = min(j+int(fil_win/2)+1, img.shape[1])
            max_ratio = np.max(ratio_store[x_min:x_max,y_min:y_max])
            # if local maximum and greater than threshold
            if ratio_store[i,j] == max_ratio and ratio_store[i,j] > thresh:
                pad_size = int(fil_win/2)
                if i >= pad_size and i < img.shape[0] - pad_size and j >= pad_size and j < img.shape[1] -
pad_size:
                    corner_pts.append((i,j))
    print('finished calculating corner points')
```

```
    return corner_pts

#-------------------------------------------------------------------------------------------------------------------------

def make_gray(img):

    new_img = []

    for i in range(img.shape[0]): #Making the image grayscale

        new_img.append(list(map(lambda x: sum(x)/3, img[i])))

    img = np.array(new_img)

    return img

#-------------------------------------------------------------------------------------------------------------------------

def euclid(x,y):

    #print('in assert: x = ', x, ', y = ', y)

    assert len(x) == len(y)

    #return np.sqrt(np.sum(np.square(np.subtract(x,y))))

    return np.linalg.norm(np.subtract(x,y))

#-------------------------------------------------------------------------------------------------------------------------

def ncc(cor_pts1, cor_pts2, img1, img2):

    #returns a list of point correspondences, where every element is of the form [(pt_x1, pt_y1), (pt_x2,
pt_y2)]

    fil_win = 31 #filter window length

    pad_len = int(fil_win/2)

    ncc_store = np.zeros((len(cor_pts1), len(cor_pts2))) #array to store all NCCs

    ncc_store = ncc_store - 2

    for i in range(len(cor_pts1)):

        for j in range(len(cor_pts2)):

            cor1 = cor_pts1[i]; cor2 = cor_pts2[j]


            x_lo1 = max(0, cor1[0]-pad_len)

            x_hi1 = min(cor1[0]+pad_len+1, img1.shape[0])

            y_lo1 = max(0,cor1[1]-pad_len)
```

```
        y_hi1 = min(cor1[1]+pad_len+1, img1.shape[1])


        x_lo2 = max(0, cor2[0]-pad_len)
        x_hi2 = min(cor2[0]+pad_len+1, img2.shape[0])
        y_lo2 = max(0,cor2[1]-pad_len)
        y_hi2 = min(cor2[1]+pad_len+1, img2.shape[1])


        if x_hi1 - x_lo1 == x_hi2 - x_lo2 and y_hi1 - y_lo1 == y_hi2 - y_lo2:
            mean1 = np.mean(img1[x_lo1:x_hi1,y_lo1:y_hi1])
            mean2 = np.mean(img2[x_lo2:x_hi2,y_lo2:y_hi2])
            term1 = np.subtract(img1[x_lo1:x_hi1,y_lo1:y_hi1], mean1)
            term2 = np.subtract(img2[x_lo2:x_hi2,y_lo2:y_hi2], mean2)
            ncc_store[i,j] = np.divide(np.sum(np.multiply(term1, term2)),
                                np.sqrt(np.multiply(np.sum(np.square(term1)), np.sum(np.square(term2)))))
            #np.sum(np.square(np.subtract(img1[x_lo1:x_hi1,y_lo1:y_hi1],
img2[x_lo2:x_hi2,y_lo2:y_hi2])))


    to_ret = [];nccs = []
    #thresh = np.min(ssd_store[ncc_store > 0])
    track = np.ones(len(cor_pts2))
    for i in range(len(ncc_store)):
        cur = ncc_store[i]
        to_find = cur[cur>=-1]
        if len(to_find) > 0:
            j = np.argmax(to_find)
            if abs(cor_pts1[i][0] - cor_pts2[j][0]) < 100 and abs(cor_pts1[i][1] - cor_pts2[j][1]) < 60 and track[j]
== 1:# and max(to_find) > 0.45:
                #if track[j] == 1:# and max(to_find) > 0.45:
                nccs.append(max(to_find))
```

```
            to_ret.append([cor_pts1[i], cor_pts2[j]])
            track[j] = 0
    sorted_idx = np.argsort(nccs)
    #to_ret = to_ret[sorted_idx]
    return (to_ret, nccs)
#-------------------------------------------------------------------------------------------------------------------
def find_homography(img1, img2, corresp, num):
    N = 30 #number of trials
    n=8 #number of corresps used to build homography using linear least squares
    delta = 50 #pixels of tolerance of discrepancy between homography outputs and actual points
    epsilon = 0.1 #rough estimate of false correspondences
    M = 8 # minimum number of inliers
    master_inliers = []
    max_len = 0; trial = 0; best_inlier = []
    while trial < N:
        #for trial in range(N):
        A = []
        rand_num = np.random.randint(0, len(corresp), 8)
        #for cor in corresp[trial: trial+6]:
        for rnum in rand_num:
            cor = corresp[rnum]
            A.append([0,0,0,-1*cor[0][0], -1*cor[0][1], -1, cor[1][1]*cor[0][0], cor[1][1]*cor[0][1], cor[1][1]])
            A.append([cor[0][0], cor[0][1], 1, 0, 0, 0, -1*cor[1][0]*cor[0][0], -1*cor[1][0]*cor[0][1], -
1*cor[1][0]])
        #Linear least squares
        #print('A = ', A)
        u,d,v_t = np.linalg.svd(A)
        v = v_t.transpose()
        h = v[:,v.shape[1]-1] #last col of v
```

Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW05

```
        h = h/h[-1]
        h = h.reshape((3,3))
        inliers = []
        #print('corresp = ', corresp)
        #remaining_corners = corresp[0:trial] + corresp[trial+6:]
        for cor in corresp:
            prediction = np.matmul(h, cor[1] + [1])
            assert len(prediction) == 3
            prediction = prediction / prediction[2]
            dist = np.linalg.norm(np.subtract(cor[0], prediction[:2]))
            if dist < delta:
                inliers.append(cor)
        #if len(inliers) > M:
        master_inliers.append(inliers)
        if len(inliers) > max_len:
            max_len = len(inliers)
            best_inlier = inliers
        trial += 1
    print('max_len = ', max_len)


    #Plotting inliers and outliers:
    color_img1 = cv2.imread(path + 'pictures/' + str(num) + '.jpg')
    color_img2 = cv2.imread(path + 'pictures/' + str(num+1) + '.jpg')


    stacked_img = np.hstack((color_img1, color_img2))
    for pt in corresp:
        cv2.line(stacked_img, (pt[0][1], pt[0][0]), (pt[1][1] + img1.shape[1], pt[1][0]), color = (0,0,255),
thickness = 1)
    for pt in best_inlier: #good ones
```

```
        cv2.line(stacked_img, (pt[0][1], pt[0][0]), (pt[1][1] + img1.shape[1], pt[1][0]), color = (255,0,0), thickness = 1)
    name = str(num) + str(num+1)
    cv2.imwrite(path + 'pictures/filtered_corresp' + name +'.jpg', stacked_img)


    #Finding best homography from best inlier:
    A = []
    for cor in best_inlier:
        A.append([0,0,0,-1*cor[0][0], -1*cor[0][1], -1, cor[1][1]*cor[0][0], cor[1][1]*cor[0][1], cor[1][1]])
        A.append([cor[0][0], cor[0][1], 1, 0, 0, 0, -1*cor[1][0]*cor[0][0], -1*cor[1][0]*cor[0][1], -1*cor[1][0]])
    #Linear least squares
    u,d,v_t = np.linalg.svd(A)
    v = v_t.transpose()
    h = v[:,v.shape[1]-1]
    h = h/h[-1]
    def to_optim(x):
        return np.matmul(A, x)


    ans = scipy.optimize.root(fun = to_optim, x0 = h, args = (), method = 'lm')
    print('ans = ', ans)
    ans = ans.x.reshape(3,3)
    return h.reshape(3,3)
#-------------------------------------------------------------------------------------------------------
def find_result(im1, im2, pts1, H):
    #Filling im1 with distorted im2:
    new_img = im1
    for i in range(im1.shape[0]): #till 2709
        for j in range(im1.shape[1]): #till 3612
            if True:
```

Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW05

```
            source = np.matmul(H, [[i], [j], [1]]);#print(source, '\n')
            source /= source[2]#; print(source)
            if source[0][0] > 0 and source[1][0] > 0 and source[0][0] < im2.shape[0] and source[1][0] < im2.shape[1]:
                new_img[i][j+600] = im2[int(source[0][0]), int(source[1][0])]
    return new_img
#-------------------------------------------------------------------------------------------------------
def main(sigma):

    imgs = []; color_imgs = []; corners = dict(); corresps = dict()

    #Reading images:
    for i in range(5):
        color_imgs.append(cv2.imread(path + 'pictures/' + str(i+1) + '.jpg'))
        imgs.append(make_gray(color_imgs[i]))
    '''#Finding corners using Harris corner detector, writing into file and plotting:
    for i in range(5):
            name = str(i+1) + str(i+2)
            corners[name] = harris(imgs[i], sigma = sigma)
            #Writing into file:
            with open(path + 'corners_' + name + '.txt', 'w') as fh:
                    for item in corners[name]:
                            fh.write(str(item[0]) + ', ' + str(item[1]) + '\n')
            #Plotting corners:
            for corner in corners[name]:
                    for j in range(corner[0] - 2, corner[0] + 3):
                            for k in range(corner[1] - 2, corner[1] + 3):
                                    if j < imgs[i].shape[0] and k < imgs[i].shape[1]:
                                            color_imgs[i][j,k] = (0,0,255)
```

```
            cv2.imwrite(path + 'pictures/corners_' + name + '.jpg', color_imgs[i])'''
    # Reading Harris corners from files:
    for i in range(5):
        name = str(i+1) + str(i+2)
        with open(path + 'corners_' + name + '.txt', 'r') as fh:
            lines = fh.readlines()
            corners[name] = []
            for line in lines:
                corners[name].append(list(map(lambda x: int(x.strip()), line.split(','))))


    '''# Reading SIFT corner points:
    for i in range(5):
        name = str(i+1) + str(i+2)
        with open(path + 'sift' + str(i+1) + '.txt', 'r') as fh:
            txt = fh.read()
            old_pts1 = np.array(list(map(lambda x: float(x.strip()), txt.split(',')[:-1])))
            pts1 = old_pts1.reshape((int(len(old_pts1)/4), 4))
            corners[name] = list(map(lambda x: [int(x[0]), int(x[1])], pts1))
    #print('corners: ', corners['12'][:7])
    print('Done reading sift corner points')'''
    '''#Finding euclidean distances
    for i in range(4):
        name = str(i+1) + str(i+2)
        nxt_name = str(i+2) + str(i+3)
        corresps[name] = []
        for pt in corners[name]: #for every corner, find correspondence in next image
            cor = corners[nxt_name][0]
            min_dist = euclid(pt, cor)
            for i in range(1,len(corners[nxt_name])):
```

```
            dist = euclid(pt,corners[nxt_name][i])
            if dist < min_dist:
                min_dist = dist
                cor = corners[nxt_name][i]
        #print('min_dist = ', min_dist)
        if min_dist <= 12:
            corresps[name].append([[int(pt[0]), int(pt[1])], [int(cor[0]), int(cor[1])]])'''



    #Plotting correspondences between images
    for i in range(4):
        name = str(i+1) + str(i+2)
        corresp, nccs = ncc(corners[name], corners[str(i+2) + str(i+3)], imgs[i], imgs[i+1]) #correspondences.
ncc
        corresps[name] = corresp
        #print('found correspondences of size ', len(corresp))
        stacked_img = np.hstack((color_imgs[i], color_imgs[i+1]))
        for pt in corresp:
            cv2.line(stacked_img, (pt[0][1], pt[0][0]), (pt[1][1] + imgs[i].shape[1], pt[1][0]), color = (0,0,255),
thickness = 1)
        cv2.imwrite(path + 'pictures/corresp' + name +'.jpg', stacked_img)
    #RANSAC to find best homography:
    h = []
    for i in range(4):
        h.append(find_homography(imgs[i], imgs[i+1], corresps[str(i+1) + str(i+2)], i+1))


    '''#Writing homographies in file:
    with open( path + 'homographies_raw.txt', 'w') as fh:
        for i in h[0].flatten():
```

```
            fh.write(str(i) + ',')
        fh.write('\n')
        for i in h[1].flatten():
            fh.write(str(i) + ',')
        fh.write('\n')
        for i in h[2].flatten():
            fh.write(str(i) + ',')
        fh.write('\n')
        for i in h[3].flatten():
            fh.write(str(i) + ',')'''

    '''#Reading raw homographies from file:
    h = [0,0,0,0]
    with open(path + 'homographies_raw_good.txt', 'r') as fh:
        lines = fh.readlines()
        h[0] = np.array(list(map(lambda x: float(x), lines[0].split(',')[:-1])))
        h[1] = np.array(list(map(lambda x: float(x), lines[1].split(',')[:-1])))
        h[2] = np.array(list(map(lambda x: float(x), lines[2].split(',')[:-1])))
        h[3] = np.array(list(map(lambda x: float(x), lines[3].split(',')[:-1])))
    h[0] = h[0].reshape(3,3)
    h[1] = h[1].reshape(3,3)
    h[2] = h[2].reshape(3,3)
    h[3] = h[3].reshape(3,3)'''

    #Plotting stitched images:
    pts1 = np.array([[0,0,1], [0,imgs[0].shape[1],1], [imgs[0].shape[0], imgs[0].shape[1],1],
[imgs[0].shape[0],0,1]])
    #cv2.imwrite(path + 'pictures/stitch12.jpg', find_result(imgs[0], imgs[1], pts1,h[0]))
    #cv2.imwrite(path + 'pictures/stitch23.jpg', find_result(imgs[1], imgs[2], pts1,h[1]))
```

Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW05

```
    #cv2.imwrite(path + 'pictures/stitch34.jpg', find_result(imgs[2], imgs[3], pts1,h[2]))

    #cv2.imwrite(path + 'pictures/stitch45.jpg', find_result(imgs[3], imgs[4], pts1,h[3]))

    #print('done')


    #Finding stitched homographies:

    h13 = np.matmul(h[1], h[0])

    h23 = h[1]

    h33 = np.array([[1,0,0],[0,1,0],[0,0,1]])

    h43 = np.linalg.inv(h[2])

    h53 = np.matmul(np.linalg.inv(h[2]), np.linalg.inv(h[3]))


    pts1 = np.array([[0,0,1], [0,imgs[0].shape[1],1], [imgs[0].shape[0],
                                        imgs[0].shape[1],1], [imgs[0].shape[0],0,1]])
    blank_img = np.zeros((imgs[0].shape[0],imgs[0].shape[1]*5,3),dtype='uint8') #completely blacked
out'''


    #Creating mosaic:

    step1 = find_result(blank_img, color_imgs[0], pts1, h13)

    cv2.imwrite(path + 'pictures/step1.jpg', step1)

    step2 = find_result(step1, color_imgs[1], pts1, h23)

    cv2.imwrite(path + 'pictures/step2.jpg', step2)

    step3 = find_result(step2, color_imgs[2], pts1, np.linalg.inv(h33))

    cv2.imwrite(path + 'pictures/step3.jpg', step3)


    step4 = find_result(step3, color_imgs[3], pts1, np.linalg.inv(h43))

    cv2.imwrite(path + 'pictures/step4.jpg', step4)

    step5 = find_result(step4, color_imgs[4], pts1, np.linalg.inv(h53))

    cv2.imwrite(path + 'pictures/step5.jpg', step5)
#-------------------------------------------------------------------------------------------------------------------------
```

Rehana Mahfuz (rmahfuz@purdue.edu)
ECE 661 HW05

```
if __name__ == '__main__':
    main(sigma = 2.1)
```