IMGS 682 Spring 2018 - Homework 3

Nilesh Pandey - jdoe@rit.edu

Due: See MyCourses

Instructions

Your homework submission must cite any references used (including articles, books, code, websites, and personal communications). All solutions must be written in your own words, and you must program the algorithms yourself. If you do work with others, you must list the people you worked with. Submit your solutions as a PDF to the Dropbox Folder on MyCourses.

Your homework solution must be prepared in LaTeXand output to PDF format. I suggest using `http://overleaf.com` or BaKoMa TeXto create your document.

Your programs must be written in Python. The relevant code to the problem should be in the PDF you turn in. If a problem involves programming, then the code should be shown as part of the solution to that problem. One easy way to do this in LaTeXis to use the verbatim environment, i.e., \begin{verbatim} YOUR CODE \end{verbatim}

If you have forgotten your linear algebra, you may find the Matrix Cookbook useful, which can be readily found online. You may wish to use the program MathType, which can easily export equations to AMS LaTeXso that you don't have to write the equations in LaTeXdirectly: `http://www.dessci.com/en/products/mathtype/`

If told to implement an algorithm, don't just call a toolbox.

Problem 1 - Edge Detection

In this problem, you are going to create a simple edge detection algorithm, which is a crude variant of the Canny edge detector.

Part 1 (2 points)

Implement a function `[xgrad, ygrad, mag, ang] = imgrad(img, sigma, k)` that convolves an image with derivative of Gaussian filters. The function takes as its input a monochrome image `img`, the standard deviation of the Gaussian (`sigma`), and the filter size (`k`). The function outputs the x- and y-gradients of the image (`xgrad` and `ygrad`), the gradient magnitude (`mag`) normalized to have a maximum value of 1 by dividing by the maximum value, and the gradient angle (`ang`). Compute the gradient angle only for pixels with non-zero gradient, and the angle returned should be between 0 and 360 degrees (use the `math.atan2` function in Python). You may not use a toolbox to compute the filter, but you may use built-in filtering functions or the functions you wrote in an earlier assignment.

Apply your function three times to `peppers.jpg` after converting it to grayscale using these parameters: `imgrad(img, 3, 9)`, `imgrad(img, 5, 19)`, and `imgrad(img, 9, 25)`. Make a 2×3 image in which the top row shows the magnitude and the bottom row shows the corresponding angle. Make sure to have a caption identifying the images. Comment on how increasing the size of the filter and sigma affect the magnitude.

Solution:

```python
def spatial_1(img, filter_):
    HH,WW = filter_.shape
    H,W = img.shape
    Ph = np.int(np.floor(HH/2.))
    Pw = np.int(np.floor(WW/2.))
    pad_x = np.pad(img, ((Ph,), (Pw,)), 'constant')
    h = int(1 + (H + 2 * Ph - HH) / 1)
    wi = int(1 + (W + 2 * Pw - WW) / 1)

    out = np.zeros((h, wi))

    # for n in range(N):
    for g in range(h):
        for t in range(wi):
            out[g, t] = np.sum(pad_x[g:g + HH, t:t + WW] * filter_)

    return out

def gkern(l=5, sig=5):


    ax = np.arange(-l // 2 + 1., l // 2 + 1.)
    xx, yy = np.meshgrid(ax, ax)

    kernel = np.exp(-(xx**2 + yy**2) / (2. * sig**2))
```

```
    return (kernel / np.sum(kernel))

def get_grad(I,k,sigma):
    f = gkern(k,sigma)
    d = np.array([1,0,-1]).reshape((3,1))
    fx = spatial_1(f,d)*-1
    k = spatial_1(I[:,:,0],fx)*-1
    k2 = spatial_1(I[:,:,0],fx.T)*-1
    M = np.sqrt(k**2 + k2**2)
    M = M/M.max()
    theta = np.arctan2(fx.T, fx)*(180/np.pi)
    angle_i = spatial_1(M,theta)*-1
    return M, angle_i
```

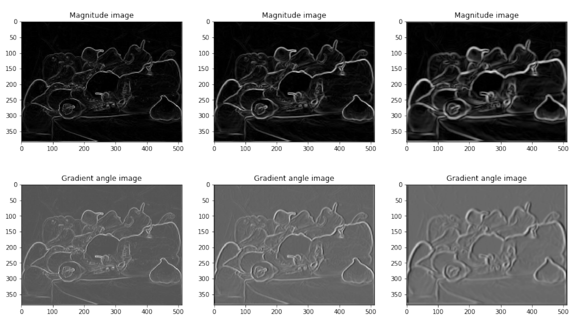


Figure 1: Gradients

As we increase the size more number of pixels come into play, which does increases edge size but makes it more blur.

Part 2 (2 points)

Write a function `out = thinEdge(mag, m)` that takes as input a gradient magnitude image `mag` and a parameter `m`. This function performs a simple form of non-maximal suppression to thin the gradient. To do this, the function looks at each 3×3 window of the image centered at a location (x, y) and it sets `out(x, y)` to `mag(x, y)` if the maximum in the window is less than `mag(i, j)*m`. All other values for `out` are 0.

Compute the gradient magnitude using the `imgrad` function using a sigma value of 1.5 and a 11×11 sized Gaussian filter for coins.jpg and a grayscale version of peppers.jpg. Apply the `ThinEdge` function to the gradient magnitude images using a value of 1.1 for `m`. On the top row, show the original gradient of peppers.jpg on the left and on the right show the version with thinned edges. On the next row, show the same thing for coins.png.

Solution:

```python
def nms(img,m):
    HH,WW = 3,3
    H,W = img.shape
    Ph = np.int(np.floor(HH/2.))
    Pw = np.int(np.floor(WW/2.))
    pad_x = np.pad(img, ((Ph,), (Pw,)), 'constant')
    h = int(1 + (H + 2 * Ph - HH) / 1)
    wi = int(1 + (W + 2 * Pw - WW) / 1)

    out = np.zeros((h, wi))
    for i in range(h):
        for j in range(wi):
            if pad_x[i:i+HH,j:j+WW].max() < img[i,j]*m:
                out[i,j] = img[i,j]
            else:
                out[i,j] = 0
    return out
```

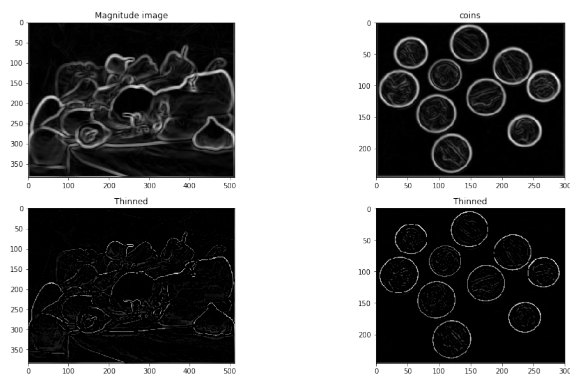


Figure 2: Gradients

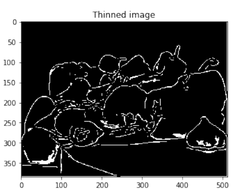
Part 3 (2 points)

Write a function `out = simpleEdge(img, sigma, theta)` that takes as input the image (`img`), the standard deviation of the Gaussian blur (`sigma`), and a threshold (`theta`) that's between 0 and 1. Internally, the function will call the `imgrad` and `thinEdge` functions you wrote. Come up with a way to choose the size of the Gaussian automatically based on sigma. Come up with a heuristic to compute a good value for `m` in `thinEdge`. The function returns a binary image with the edges found by thresholding the thinned magnitude with `theta`.

Apply the function to coins.jpg and peppers.jpg using values for sigma and theta that isolate interesting edges. Show the edge images.

Solution:

```
def getEdges(I,sigma,low,high):
    r = 2*sigma
    k = int(np.floor(2*r+1))
    m,n = get_grad(I,k,sigma)
    i = nms(m,1.1)
    t = apply_hysteresis_threshold(i, low, high)
    return t
```



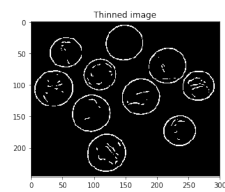


Figure 3: Thinned

Problem 2 - RANSAC

For these problems you must not use a built-in toolbox for RANSAC.

Part 1 (2 points)

For this problem you will need the file `lineData.txt`, which contains 2-d points. The first column has the x coordinates and the second column has the y coordinates.

Implement a function `[line]=fitLineLS(points)` that fits a line using least squares to all of the 2-D data points. It should output the two parameters of a line. Do not use a toolbox for fitting the line.

Plot the points and the line found using your function.

Hint: When doing the regression, augment your data with a vector of 1's.

Solution:

```python
def model(path,iteration,samples_):
    model = []
    for x in range(iteration):
        df = pd.read_csv(path, delim_whitespace=True, engine = 'python', names = range(2))
        df.columns = ['x','y']
        df['b'] = np.ones((df.shape[0],1))
        df = df[['x','b','y']]
        x = df['x'].as_matrix()
        y = df['y'].as_matrix()
        M = df[['x','b']].as_matrix()

        reshuffled_ind = np.random.randint(0,df.shape[0],samples_)
        M = M[reshuffled_ind]
        y = y[reshuffled_ind].reshape((len(reshuffled_ind),1))

        mt = M.T.dot(M)
        mti = np.linalg.pinv(mt)
        mtim = mti.dot(M.T)
        mtimy = mtim.dot(y)
        model.append(mtimy)

    return model

 def line_dist(model,x,y):
    a,c = model
    d = np.abs(a*x + y + c)/np.sqrt(a**2 + 1)
    return d



plt.scatter(X,Y)
plt.plot(X, vall[0][0]*X+vall[0][1], linewidth=2.0)
plt.xlabel('X')
plt.ylabel('Y')
```

Part 2 (4 points)

For this problem you will need the file `lineData.txt` , which contains 2-d points. The first column has the x coordinates and the second column has the y coordinates.

Write a function `[line,inliers]=fitLineRANSAC(points,iter,thr,inlierRatio)` that implements the RANSAC algorithm for fitting a line. The function returns the parameters to the line in `line` and `inliers` is a Boolean array that indicates whether each point is

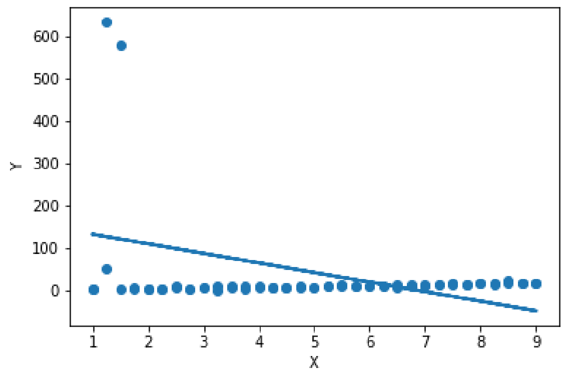


Figure 4: Linear Regression

an inlier (true) or an outlier (false). `points` contains a $n \times 2$ array with the (x, y) data points to be fit, `iter` is an integer indicating the number of iterations to run RANSAC, `thr` is a positive number indicating how close the points need to be to an estimated line to be considered inliers, and `inlierRatio` is the percent of points that need to be classified as inliers for a model to be fit with least squares. Call the function with the parameters `[line,inliers]=fitLineRANSAC(points,20,250,0.9)`.

On a single image, plot the line found using least squares in black, plot the line found using RANSAC in green, plot the inliers as blue circles, and plot the outliers as red squares.

If you have questions on RANSAC, see section 6.1.4 from Szeliski.

Solution:

```
def model(path, iteration, samples_):
    model = []
    for x in range(iteration):
        df = pd.read_csv(path, delim_whitespace=True, engine = 'python', names = range(2))
        df.columns = ['x','y']
#           df['b'] = np.ones((df.shape[0],1))
#           df = df[['x','b','y']]
        x = df['x'].as_matrix()
        y = df['y'].as_matrix()
        M = df[['x']].as_matrix()

        reshuffled_ind = np.random.randint(0,df.shape[0],samples_)
        M = M[reshuffled_ind]
        y = y[reshuffled_ind].reshape((len(reshuffled_ind),1))

        mt = M.T.dot(M)
        mti = np.linalg.pinv(mt)
```

```
        mtim = mti.dot(M.T)
        mtimy = mtim.dot(y)
        model.append(mtimy)

    return model

 import pdb
def line_dist(model,x,y):
#       pdb.set_trace()
    a = model
    d = np.abs(a*x + y)/np.sqrt(a**2 + 1)
    return d


    def find_inliers(model,x,y,threshold,ratio):
    dic = {}
    num = 0
    for i,mod in enumerate(model):
        print(mod)
        i = 0
        mod = np.array(mod)
        for x1,y1 in zip(x,y):
            d = line_dist(mod,x1,y1)
        #       i = 0
            if d < threshold:
                i = i+1
        print("no_of_inliers:{} model:{}".format(i,mod))
        if i*ratio > num:
            num = i*ratio
            best_ = mod
        dic[i] = i*ratio
    return dic,best_
```

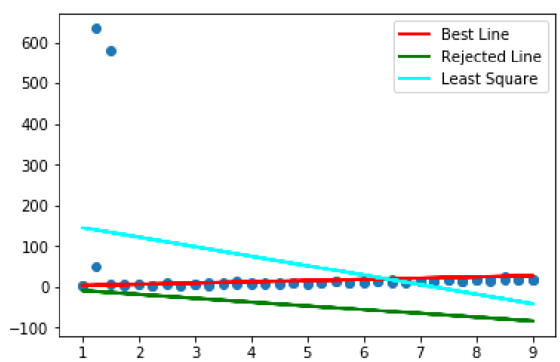


Figure 5: RANSAC Regression

Part 3 (6 points)

For this problem you will need the file `circleData.txt`, which contains 2-d points. The first column has the x coordinates and the second column has the y coordinates.

Write a function `[fit,inliers]=fitCircleRANSAC(points,iter,thr,inlierRatio)` that implements the RANSAC algorithm for fitting a circle. The function returns the parameters to the circle in `fit` and `inliers` is a Boolean array that indicates whether each point is an inlier (true) or an outlier (false). `points` contains a $n \times 2$ array with the (x, y) data points to be fit, `iter` is an integer indicating the number of iterations to run RANSAC, `thr` is a positive number indicating how close the points need to be to an estimated circle to be considered inliers, and `inlierRatio` is the percent of points that need to be classified as inliers for a model to be fit with them. Call the function with these parameters `[fit,inliers]=fitCircleRANSAC(d,10,1,.9)`.

On a single image, plot the circle found using least squares in black, plot the circle found using RANSAC in green, plot the inliers as blue circles, and plot the outliers as red squares. In text, **give the parameters found for the center of the circle and its radius** when you fit it without RANSAC using all of the points, and give the parameters when only the inliers found with RANSAC were used to fit the circle.

Solution:
Parameters found using all points: $(?, ?)$ with $r =?$
Parameters found using RANSAC inliers only: $(?, ?)$ with $r =?$

PUT AN IMAGE/PLOT HERE WITH A LEGEND AND ALSO SOME CODE HERE. YOU MOSTLY JUST NEED TO MODIFY THE CODE YOU WROTE FOR FITTING THE LINE WITH RANSAC

Problem 3 - Corner Detection (10 points)

Write a function `[points]=harris(img, win, theta)` that implements the Harris corner detector as described in class. As input, the function takes an image (`img`), the Harris window size (`win`), and a threshold (`theta`) for determining if a point is a corner. It returns the (x, y) coordinates of the corners (`points`). Remember to perform non-maximal suppression, which you can do with the `thinEdge` function you wrote earlier. Use a value of 0.04 for the Harris sensitivity parameter. Wikipedia contains an excellent article on the Harris corner detector.

Test your code on house.bmp, and show all of the corners. Include in your writeup a zoomed-in figure detailing the results in a selected interesting neighborhood.

Solution:

```
def get_corner(resp_img,threshold,min_dist):
    corner_th = max(resp_img.flatten())*threshold #we calculate threshold
    imt = (resp_img>corner_th) # we filter values below threshold
    corner = imt.nonzero() # we eliminate values which are vacant and we get two array with lo
    n_cords = [ (corner[0][c],corner[1][c]) for c in range(len(corner[0]))] #corner is 1d arra
    # corner[0] == corner[1] length, both contain location [1,4,5],[34,56,3]
    n_corners = [resp_img[c[0]][c[1]] for c in n_cords]
    # we have cords with threshold filter, and we pass this cords to response image to get fea
    index = np.argsort(n_corners)# we get index sorted

    location_to_disp = np.zeros(resp_img.shape)
    location_to_disp[min_dist:-min_dist,min_dist:-min_dist] = 1
    filterted_location = []
    for i in index: #Index sorted for the best match feature
        if location_to_disp[n_cords[i][0]][n_cords[i][1]] == 1:
            filterted_location.append(n_cords[i])#from the matrix that we formed, the only cor
            location_to_disp[(n_cords[i][0]-min_dist):(n_cords[i][0]+min_dist),
                (n_cords[i][1]-min_dist):(n_cords[i][1]+min_dist)] = 0 #we need to make distan
            #since there are many corners detected and overlapped n_cords[i][0] will give the
            #few pixels will help us getting over repeated features
    return filterted_location

#reference http://www.kaij.org/blog/?p=89


def hariss_corner(I,kernel_size,sig,k,threshold,min_dist,color):
    d = np.array([1,0,-1]).reshape((3,1)) #difference filter
    f = gkern(kernel_size,sig)
    Gx = scipy.signal.convolve2d(f,d,mode="same")
    Gy = Gx.T


    Ix = scipy.signal.convolve2d(I[:,:,0],Gx,mode="same")
    Ix = nms(Ix,1.1)
    Iy = scipy.signal.convolve2d(I[:,:,0],Gy,mode="same")
    Iy = nms(Iy,1.1)
    Ix2 = np.square(Ix)
    Iy2 = np.square(Iy)
    Ixy = Ix*Iy

    Sx2 = scipy.signal.convolve2d(Ix2,f,mode="same")
    Sy2 = scipy.signal.convolve2d(Iy2,f,mode="same")
    Sxy = scipy.signal.convolve2d(Ixy,f,mode="same")

    H = np.array(([Sx2,Sxy],[Sxy,Sy2]))

    det = (Sx2 * Sy2) - (Sxy**2)
```

```
    R = det - 0.04*((np.trace(H))**2)

    Im_corner = get_corner(R, threshold, min_dist)

    plt.figure(figsize=(18,19),dpi=80)
    plt.plot([p[1] for p in Im_corner],[p[0] for p in Im_corner],'+',color=color)
    plt.imshow(I)
    return Im_corner


img = cv2.imread('/home/n/images/house.bmp')
I = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

filtered_ = hariss_corner(I,3,2,10,0.04,5,"red")
```

Figure 6: Corner detection

Problem 4 - SIFT and Correspondences

In this problem you will use the images sample-a.jpg and sample-b.jpg. You will find correspondences between the two images and you will indicate which points are best matched together. Here is an example output on some other images.

Part 1 (4 points)

Using the Harris corner detector you implemented earlier, detect corners on both images. Extract feature vectors around each corner (keypoint) by simply taking a $k \times k$ patch of pixels centered at the corner and treating it as a vector. To do this, write a function called `getPatchFeatures(img, corners, k)` that takes as its input an image, a list of n corners, and a window size k, and returns a $ck^2 \times n$ array of features, where c is the number of color channels (e.g., $c = 1$ for grayscale images).

Use the sum squared distance (SSD) to compute similarity between the corners in the two images, and draw lines between the corners that have the lowest (SSD). You may use grayscale or color patches, but discuss in your write-up what you are doing. What value of k did you use? How well did this approach work? If it didn't work, why do you think that happened? Tune your parameters until it matches correspondences as well as possible.

Solution:

```
def comapred_features(features_Vectors1 ,cords_im1 ,features_Vectors2 ,cords_im2):
    dis = []
    loc = []
    cords = []
    s = []
    for v1,c_1 in zip(V_1,c1):
        for v2,c_2 in zip(V_2,c2):
            d = np.sqrt(np.sum(np.square(v2-v1)))
            dis.append(d)
            loc.append((c_1 ,c_2))
        q = np.argmin(dis)
        cords.append(loc[q])
        s.append(dis[q])
        dis = []
        loc = []
    return cords,s

def extract_feature_vector(I,corners,w,c):
    W = int(np.floor(w/2))
    lol = []
    cords = []
    i = 0
    if c == 1:
        I = I[:,:,0]
        j = np.zeros((len(corners),w**2))
    elif c == 2:
        I = I[:,:,0:2]
        j = np.zeros((len(corners),w**2,c))
```

```
    else:
        I = I
        j = np.zeros((len(corners),w**2,3))
    for v in corners:
        lol = []

        xc = [(v[0]+i) for i in range(-W,W+1)]
        yc = [(v[1]+i) for i in range(-W,W+1)]
        for x in xc:
            for y in yc:

                lol.append(I[x][y])
        cords.append((v[0],v[1]))
        k = np.array(lol)
        j[i] = k
        k = 0
        i = i+1
    return j,cords
def get_matches(distance,cords,n,):
    cc = np.argsort(distance)
    ccc = np.array(cc[:n])
    loc = np.array(cords)
    required = loc[ccc]
    uuu = np.zeros((n,1,2))
    uuu[:,[0]] = (0,1000) #change 1000
    uu = required[:,[1]]
    co = uu+ uuu
    required[:,[1]] = co
    return required

numpy_horizontal = np.vstack((I1, I2))
plt.figure(figsize=(18,19),dpi=80)
plt.plot([p[0][1] for p in required[:,[0]]],[p[0][0] for p in required[:,[0]]],'+',color="cyan
plt.plot([p[0][1] for p in required[:,[1]]],[p[0][0] for p in required[:,[1]]],'+',color="red"
plt.plot([[p[0] for p in required[:,[0],[1]]],[p[0] for p in required[:,[1],[1]]]],[[p[0] for


plt.imshow(numpy_horizontal)
```

This method works only if we have applied non maximum suprression, or else it will detect many similar false points.

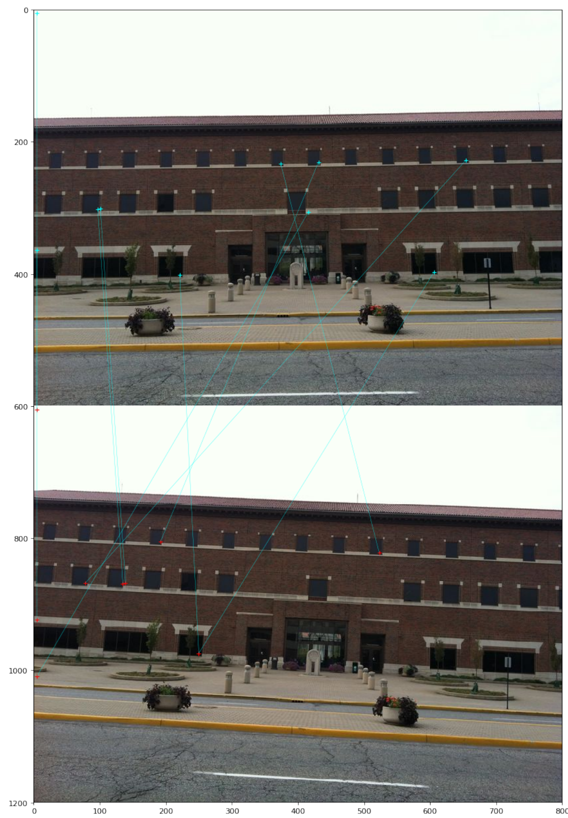


Figure 7: Feature Matching

Part 2 (4 points)

Using the Harris corner detector you implemented earlier, detect corners on both images. Extract feature vectors around each corner by computing a color histogram around the $k \times k \times 3$ patch of pixels centered at each corner and treating it as a vector. To do this, you will need to compute a histogram with d bins for each channel, followed by concatenating the three histograms together, and finally normalizing the $3d$-dimensional feature vector to sum to 1 by dividing by the L_1 norm. To compare a keypoint feature vector $\mathbf{p}$ in one image to a vector $\mathbf{q}$ in the other image use the χ^2 distance, which is used for comparing normalized histograms:

$$D\left(\mathbf{p}||\mathbf{q}\right) = \frac{1}{2}\sum_i \frac{\left(p_i - q_i\right)^2}{p_i + q_i + \epsilon},$$

where ϵ is a very tiny positive number to prevent division-by-zero.

Use RGB color space and the color space of your choice. Show the matches for both color spaces and make sure to label the images. What number of bins did you use? How well did this approach work? If it didn't work, why do you think that happened?

Solution:
Solution:

```
from sklearn import preprocessing

img = cv2.imread('/home/n/images/sample-a.jpg')
I1 = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

img = cv2.imread('/home/n/images/sample-b.jpg')
I2 = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

def get_histogram_features(Image,k,bins,l):
    lol_b = I1[:,:,2]
    lol_g = I1[:,:,1]
    lol_r = I1[:,:,0]
    k = k
    bins = bins
    n = 120
    features_a= []
    filtered_ = hariss_corner(I1,3,2,2,0.001,5,"red")


    for i in (filtered_[:120]):


        V_b = lol_b[(i[0]-k):(i[0]+k+1),(i[1]-k):(i[1]+k+1)]
        V_g = lol_g[(i[0]-k):(i[0]+k+1),(i[1]-k):(i[1]+k+1)]
        V_r = lol_r[(i[0]-k):(i[0]+k+1),(i[1]-k):(i[1]+k+1)]

        hist_b,bins = np.histogram(V_b.flatten(),bins)
        hist_g,_ = np.histogram(V_g.flatten(),bins)
        hist_r,_ = np.histogram(V_r.flatten(),bins)
        coners = np.hstack([hist_b,hist_g,hist_r])

#           print(i)


        norm = np.linalg.norm(con)
        coners = coners/norm
        features_a.append(coners)

    features_a = np.array(features_a)
    return features_a,filtered_
```

```
d = []
sort2 = []
index2 = []

corn_b = []
for i in range(120):
    for k in range(120):
        s =(0.5)*np.sum(np.divide(np.square(np.subtract(V_a[i,:],V_b[k,:])),np.add(V_a[i,:],V_
        d.append(s)
    _ = np.min(d)
    sort2.append(_)
    index2 = d.index(_)
    d = []

    corn_b.append(corner_b[index2])
sortcor1 = []
sortcor2 = []
sort1 = sorted(sort2)




for k in range(120):
    sortcor1.append(corner_a[sort2.index(sort1[k])])
    sortcor2.append(corn_b[sort2.index(sort1[k])])

IMG = np.hstack((I1,I2))
plt.figure(figsize=(18,19),dpi=80)
sortcor2 = sortcor2[1]+800


plt.plot([p[1] for p in sortcor1],[p[0] for p in sortcor1],'*')
plt.plot([p[1] for p in sortcor2],[p[0] for p in sortcor2],'*')



plt.plot([[p[1] for p in sortcor1],[p[1] for p in sortcor2]],[[p[0] for p in sortcor1],[p[0] f

plt.imshow(IMG)
```

number of bins used 9, because of color intensity variation it usually matches wrong points more often.

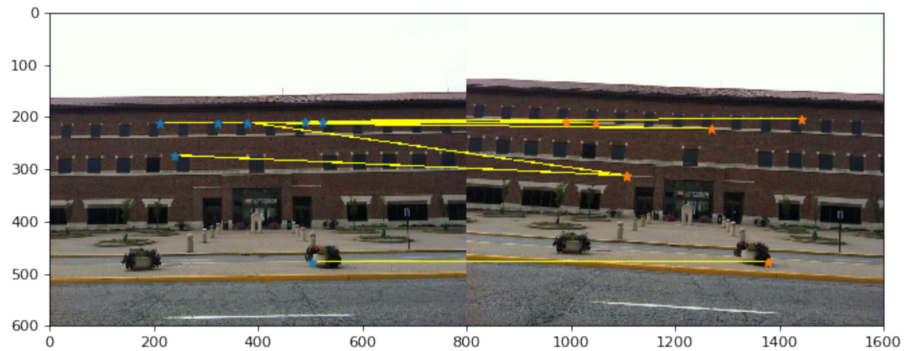


Figure 8: Histogram Corners

Part 3 (4 points)

Use SIFT to extract keypoints from each image and then use SSD to compute matches between the two images. You may use your corner detector, or the SIFT implementation you use may do it automatically for you. For this problem, you may use a toolbox for extracting SIFT features. SIFT is built into OpenCV. Tune SIFT's parameters until it matches correspondences as well as possible. You should discard points that have a high distance compared to other points. For example, you could show only the 50 correspondences that have the lowest distance.

Show the matches. How well did it work compared to the simpler methods you used earlier?

Solution:

```
import cv2
import numpy as np

img1 = cv2.imread('/home/n/images/sample-a.jpg')
gray1= cv2.cvtColor(img1,cv2.COLOR_BGR2GRAY)
img2 = cv2.imread('/home/n/images/sample-b.jpg')
gray2 = cv2.cvtColor(img2,cv2.COLOR_BGR2GRAY)

sift = cv2.xfeatures2d.SIFT_create()
kp1, des1 = sift.detectAndCompute(gray1,None)
kp2, des2 = sift.detectAndCompute(gray2,None)
# kpimg = cv2.drawKeypoints(gray, kp, None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
# plt.figure(figsize=(18,19),dpi=80)

# plt.imshow(kpimg)
im1cords = 0
```

```
im2cords = 0

dis = []
loc = []
cords = []
s = []
for v1,c_1 in zip(des1,im1cords):
    for v2,c_2 in zip(des2,im2cords):
        d = np.sqrt(np.sum(np.square(v2-v1)))
        dis.append(d)
        loc.append((c_1,c_2))
    q = np.argmin(dis)
    cords.append(loc[q])
    s.append(dis[q])
    dis = []
    loc = []

 def get_matches(distance,cords,n):
    cc = np.argsort(distance)
    ccc = np.array(cc[:n])
    loc = np.array(cords)
    required = loc[ccc]
    uuu = np.zeros((n,1,2))
    uuu[:,[0]] = (0,800)
    uu = required[:,[1]]
    co = uu+ uuu
    required[:,[1]] = co
    return required

  lol = get_matches(s,cords,50)


  plt.figure(figsize=(18,19),dpi=80)
kpimg = cv2.drawKeypoints(gray1, kp1, None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
kpimg2 = cv2.drawKeypoints(gray2, kp2, None, flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
numpy_horizontal = np.hstack((kpimg,kpimg2))


plt.plot([[p[0] for p in lol[:,[0],[1]]],[p[0] for p in lol[:,[1],[1]]]],[[p[0] for p in lol[:
plt.imshow(numpy_horizontal,cmap="gray")
```

It is faster compared to many methods that we have come across, it does gives false matching, but for the number of lines we had the other two would have given more mismatching points.

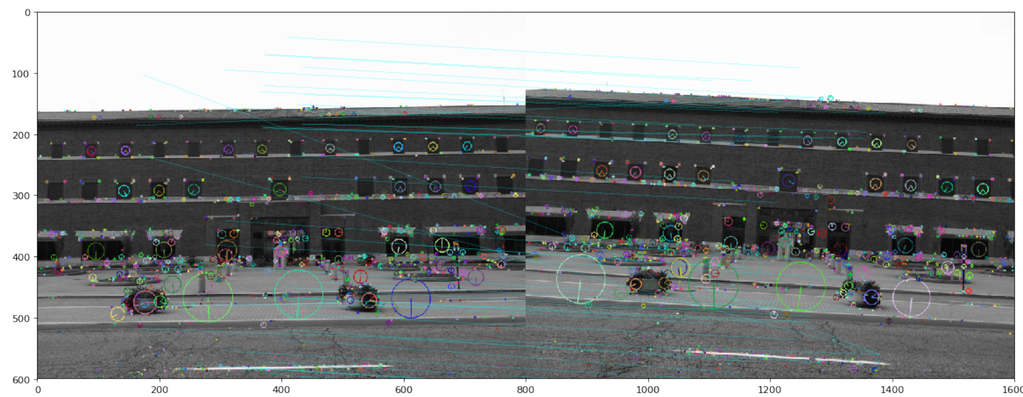


Figure 9: RANSAC Regression

Problem 5 - Homographies

Part 1 (2 points)

Write a function that displays an image and let's the user click on n points. The function will return the n points clicked. Write another function that takes a set of points that are on a quadrilateral and computes (or makes up) a corresponding rectangle.

Test your method on building.gif by clicking on the four points specified in the image and outputting your estimate of what the parameters of a rectangle would be without the projective distortion. Show the points you clicked on the image.

Solution:

```
import cv2
op = []
def clickEvent(event,x,y,flags,param):
    global op

    if event == cv2.EVENT_LBUTTONDBLCLK:
        print(x,y)
        op.append((x,y))


WINDOW_NAME = "win"

image = I1
cv2.namedWindow(WINDOW_NAME, cv2.WINDOW_AUTOSIZE)
# initialtime = time.time()
```

```
cv2.startWindowThread()
cv2.imshow(WINDOW_NAME, image)
# cv.waitKey(1000)
cv2.setMouseCallback("win", clickEvent)
# cv.waitKey(1)
# cv.destroyAllWindows()
c = cv2.waitKey(0)
if 'q' == chr(c & 255):
    cv2.destroyAllWindows()


def convert_rectangle(img, points):
    points = np.array(points)
    points[1][1] = points[0][1]
    points[2][0] = points[1][0]
    points[3][0] = points[0][0]
    points[3][1] = points[2][1]
    c =img[points[0][1]:points[2][1], points[0][0]:points[1][0]]
    plt.imshow(c)

    return points, c
print(op):
[(374, 284), (488, 273), (498, 338), (372, 338)]
```

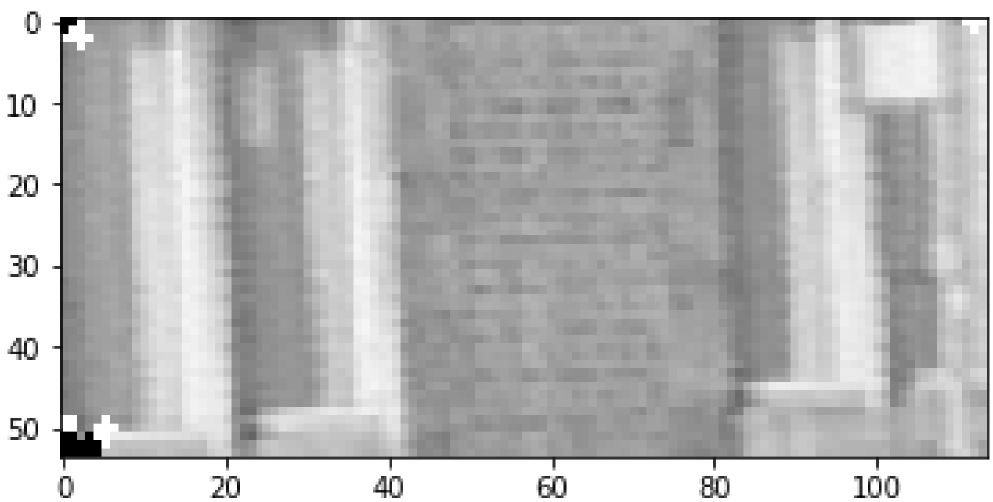


Figure 10: Rect Image clicked

Part 2 (10 points)

Implement the four-point algorithm for a planar scene to estimate the homography matrix **H** between two images. The function will take as input two sets of correspondences and output the homography matrix **H**. Call the function `fourPointAlgorithm`.

Write another function called `removePerspectiveProjection` that takes as input an image and a list of four points that should be on a rectangular region that has been distorted by perspective projection. It will output the image with the perspective projection removed by calling the `fourPointAlgorithm` function you wrote. To generate the list of four points, use $n \geq 4$ hand-clicked points on a rectangle altered by projective distortion. Apply your method to two images: building.gif and floor.png. Make sure your program works on color images. **Show the images with the distortion removed with the points that were clicked shown on the new image and provide the homography matrices for both images.**

Hint: To do this you will need to make up the correspondences by determining what the points would have been without the perspective projection and if they were actually rectangular. You will need to apply the **H** matrix you found to all of the pixels in the original images to compute the new image.

Solution:

```
def gethomography(p1, p2):
    p1 = np.array(p1)
    p2 = np.array(p2)
    lol = []
    for i in range(4):
        x, y = p1[i][0], p1[i][1]
        u, v = p2[i][0], p2[i][1]
        lol.append([x, y, 1, 0, 0, 0, -u*x, -u*y, -u])
        lol.append([0, 0, 0, x, y, 1, -v*x, -v*y, -v])
    lol = np.array(lol)
    U, S, v = np.linalg.svd(lol)
    L = v[-1,:] / v[-1,-1]
    H = L.reshape(3, 3)
    return H

o = cv2.warpPerspective(I1,H,(305, 344))

# First Image H
array([[ 3.60843672e-01, -2.35128975e-01,  1.72152579e+02],
       [-1.49839285e-01,  3.91922497e-01,  1.26937675e+02],
       [-5.88399447e-04, -4.75969585e-04,  1.00000000e+00]])

# Second Image H
```

```
array([[ 2.02978972e+00,  6.95857737e-01, -1.61747271e+02],
       [ 4.32598039e-01,  2.52199947e+00, -1.98709215e+02],
       [ 1.09985177e-03,  3.39663349e-03,  1.00000000e+00]])
```

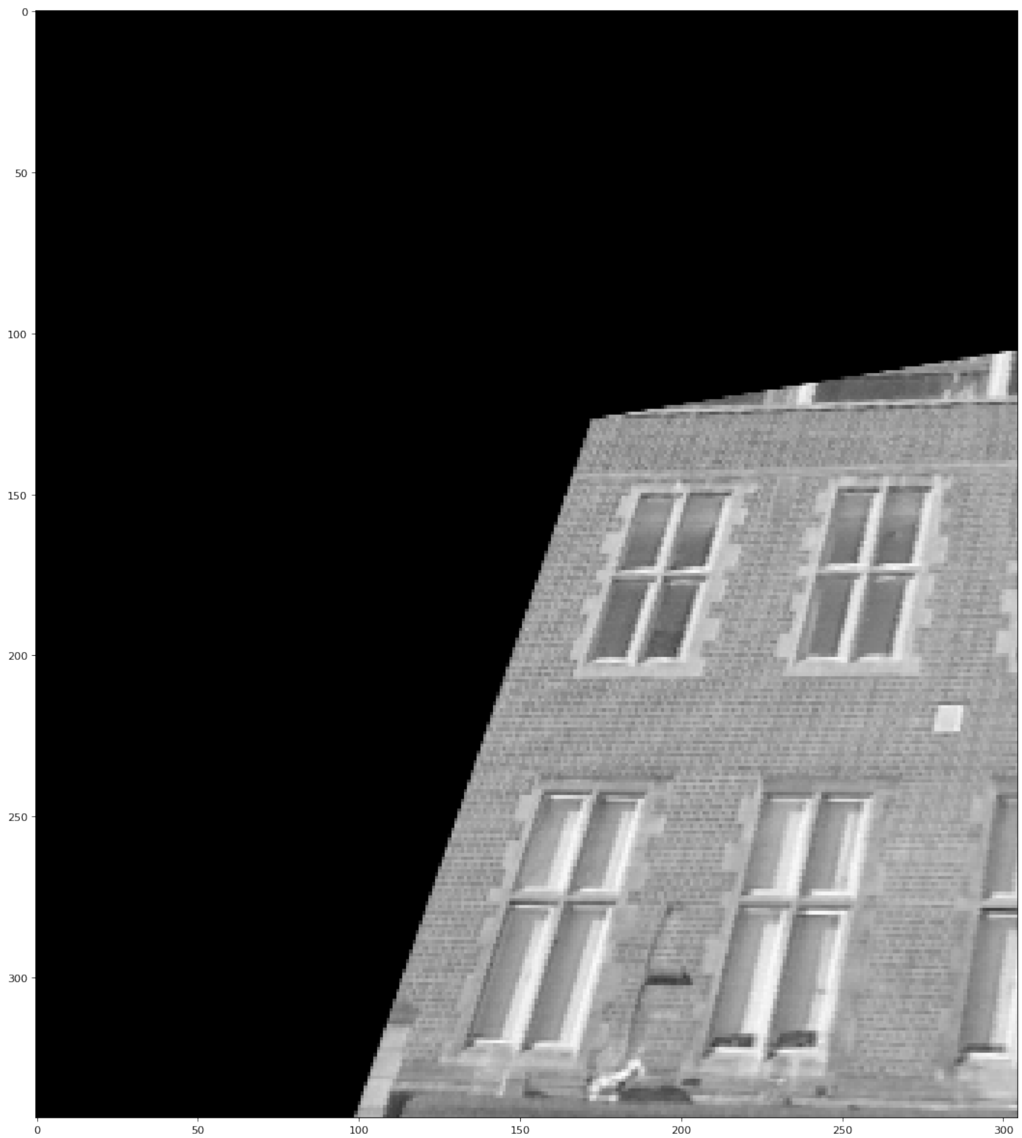


Figure 11: 1st Rect Image Converted

Figure 12: 2ndRect Image Converted

Problem 6 - Automatic Image Stitching

For these problems you must not use a built-in toolbox for RANSAC, computing homographies, or image stitching. I suggest that you use a toolbox for SIFT to compute interest points and descriptors, but you could also use the Harris detector you implemented earlier.

You should make it easy to swap out descriptors and interest point detectors.

To receive credit, your method must do a qualitatively good job of stitching the images together automatically (i.e., all the parts are in the right places). No credit will be given if you use a toolbox to do the stitching. This problem may require some trial and error in putting together the pieces that you have written already.

You may use toolbox functions that do the warping when supplied with your homography.

Part 1 (8 points)

Devise and implement a RANSAC-based method for automatically estimating the homography $\mathbf{H}$ between two images and then using it to stitch the two images together. Explain your algorithm and provide the code.

Apply your algorithm to the image pair desk0.gif and desk1.gif, which was acquired by a camera rotating about its optical center to compute $\mathbf{H}$. Show the matrix $\mathbf{H}$, and stitch the images together using $\mathbf{H}$.

Solution:

```
import imageio
# import urllib.request

# url = "/home/n/images/desk0.gif"
# url1 = "/home/n/images/desk1.gif"
# fname = "tmp.gif"

## Read the gif from the web, save to the disk
# imdata = urllib.request.urlopen(url).read()

## Read the gif from disk to `RGB`s using `imageio.miread`
gif1 = imageio.imread("/home/n/images/desk0.gif")
gif2 = imageio.imread("/home/n/images/desk1.gif")

# convert form RGB to BGR

import cv2
import numpy as np

# img1 = cv2.imread('/home/n/images/desk0.gif')

sift = cv2.xfeatures2d.SIFT_create()
kp1, des1 = sift.detectAndCompute(gif1,None)
kp2, des2 = sift.detectAndCompute(gif2,None)
```

```
im1cords = [p.pt for p in kp1]
im2cords = [p.pt for p in kp2]

im1cords = np.array(im1cords)
im2cords = np.array(im2cords)

dis = []
loc = []
cords = []
s = []
for v1,c_1 in zip(des1,im1cords):
    for v2,c_2 in zip(des2,im2cords):
        d = np.sqrt(np.sum(np.square(v2-v1)))
        dis.append(d)
        loc.append((c_1,c_2))
    q = np.argmin(dis)
    cords.append(loc[q])
    s.append(dis[q])
    dis = []
    loc = []

 cords_sorted = cords[index]
 lol = np.ndarray.tolist(cords_sorted)
import random
```

Part 2 (6 points)

Generalize your algorithm for creating an image mosaic to handle n images by chaining together the homographies. Create a mosaic out of the images. You may need to come up with a scheme to appropriately alter an image's color and intensity to match the other images, which you can do using the correspondences.

Test your method on two datasets. The first dataset consists of 3 images: r1.jpg, r2.jpg, and r3.jpg. The second consists of 7 images: m1.jpg – m7.jpg. Explain your method, provide code, and show the results of the images after automatically stitching them together. To receive credit, your method should do a decent job of stitching the images together automatically, but I don't expect you to correct the colors.

Solution:
EXPLAIN YOUR SOLUTION, SHOW THE IMAGES, AND GIVE SOME CODE