ECE 661: Homework #9

Due on Wednesday, November 21st, 2018

Dr. Avinash Kak

Naveen Madapana

1. Estimation of Fundamental Matrix F

In this homework, a pair of stereo images are used to estimate the fundamental matrix F that enables us to transform the pixels in one image to the other. Another advantage is that, by using F, we can estimate the depth map of the stereo images. Let $\vec{x}$ and $\vec{x}'$ be the points on the left image and right image. Note that there is a one to one correspondence between the left and right images. The constraint satisfied by F is the following: $\vec{x}'^T F \vec{x} = 0$ where $\vec{x}$.

Linear least squares approach is used to estimate the fundamental matrix F. The following images are considered in this regard. The points highlighted in red the manual correspondences that are considered for estimation of F. Look at figure 1

(a) Left Image or First Image

(b) Right Image or Second Image

Figure 1: Manually located eight point correspondences on the left and right images.

The detailed steps are given below:

1. **Normalization:**
   In this step, we want to normalize the images so that that center pixel of both the left and right images correspond to the origin $(0, 0)$. This is achieved by premultiplying the following matrices to the point correspondences of left and right images. Where T is the normalization transformation matrix, W is the width of the image and H is the height of the image.

$$\vec{x}_{norm} = T\vec{x}, \qquad \vec{x}'_{norm} = T\vec{x}'$$

$$T = \begin{bmatrix} 1 & 0 & -W/2 \\ 0 & 1 & -H/2 \\ 0 & 0 & 1 \end{bmatrix}$$

   The normalized correspondences: $(\vec{x}, \vec{x}')$ with $\vec{x} = (x, y, w)$ and $\vec{x}' = (x', y', w')$ will yield one equation per correspondence to estimate F. The following equations are obtained by elaborating $\vec{x}'^T F \vec{x} = 0$ where $\vec{x}$.

$$\text{Let } f = \begin{bmatrix} f_{11} & f_{12} & f_{13} & f_{21} & f_{22} & f_{23} & f_{31} & f_{32} & 1 \end{bmatrix}^T$$

   The last entry is 1 as we are working in homogeneous coordinates.

1. Estimation of Fundamental Matrix F continued on next page...

$$\vec{x}'^T F \vec{x} = 0 \rightarrow \begin{bmatrix} x'x & x'y & x' & y'x & y'y & y' & x & y & 1 \end{bmatrix} f = 0$$

There are 8 unknowns in F. Hence we need atleast eight point correspondences. Next we build the system of linear equations so that we can use linear least squares to solve for parameters in F.

$$Af = 0 \rightarrow \begin{bmatrix} x'_1x_1 & x'_1y_1 & x'_1 & y'_1x_1 & y'_1y_1 & y'_1 & x_1 & y_1 & 1 \\ & & & & \vdots & & & & \\ x'_8x_8 & x'_8y_8 & x'_8 & y'_8x_8 & y'_8y_8 & y'_8 & x_8 & y_8 & 8 \end{bmatrix} \begin{bmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \end{bmatrix} = \vec{0}$$

SVD decomposition is used to find the eigen vector corresponding to the smallest eigen value. This vector would act as solution of F.

2. **Conditioning of F:**
The matrix F is supposed to be singular, meaning one of the eigen values should be zero. Hence we need to condition the F as the linear least squares solution might not yield the F that is singular. In other words, the rank of F is supposed to be 2. The following steps are followed to achieve this.

$$F = UDV^T$$

Now, make the last eigen value in D to zero and construct D'. Now the conditioned matrix F is given as $F = UD'V^T$.
Once the conditioning is done, we need to estimate the de-normalized version of matrix F. The new F would be $F_{new} = T^T F T$

2. Rectification of Images

In this part of the homework, the images will be rectified so that rows of the left image correspond to the rows of the right image. We assume that cameras are in canonical representation i.e. the world coordinate system coincides with the left camera center. In this case the left camera matrix would be :

$$P = \begin{bmatrix} 1,0,0,0 \\ 0,1,0,0 \\ 0,0,1,0 \end{bmatrix}$$

2.1 Computing epipoles

Once we compute the value of F, it is very easy to compute the epipole of the right and left images. The left epipole lies in the null space and the right epipole lies in the left null space of the matrix F. The following steps are followed to obtain the epipoles.

1. Perform SVD Decomposition $F = UDV^T$.
2. The last column of V acts as the epipole of left camera.
3. The last column of U^T acts as the epipole of the right camera.

The camera matrix of the right camera is given as:

$$P' = \big[[e']_\times F | e'\big]$$

2.1 Non linear refinement using LM

In this section, the fundamental matrix and global estimations of the point correspondences are refined using LM algorithm. Specifically, the LM implementation of SCIPY is used in this homework.

To refine F, our goal is to minimize the geometrical error which is given as $Minimize\ d^2_{geom} = ||\vec{V} - \vec{f}(\vec{p})||$.

The loss function is minimized using the LM algorithm. The previously computed values (least squares solution) are used as the starting point for LM.

First, we need to triangulate the point correspondences to obtain the estimation of their world points using least squares. It is given by:

$$\vec{X} = N(A), \qquad \text{with:}\ \ A_{4\times 4} = \begin{bmatrix} x\vec{p}_3^T - \vec{p}_1^T \\ y\vec{p}_3^T - \vec{p}_2^T \\ x'\vec{p}_3'^T - \vec{p}_1'^T \\ y'\vec{p}_3'^T - \vec{p}_2'^T \end{bmatrix}$$

Where:

$$P = \begin{bmatrix} \vec{p}_1^T \\ \vec{p}_2^T \\ \vec{p}_3^T \end{bmatrix}, \quad P' = \begin{bmatrix} \vec{p}_1'^T \\ \vec{p}_2'^T \\ \vec{p}_3'^T \end{bmatrix}, \quad \vec{x} = [x, y, z], \quad \vec{x}' = [x', y', z']$$

The values of $\vec{X}$ can be obtained using the similar procedure of SVD. Once we know the world coordinates of the point correspondences, we use the LM to refine world coordinate estimates and the camera matrix of right camera. The idea is to re-project the world coordinates on to both the cameras using P and P' to get the corresponding pixel coordinates on the camera images $\hat{\vec{x}}$ and $\hat{\vec{x}}'$.

The geometric error is defined as :

$$Min\ d^2_{geom} = \sum_i (||\vec{x}_i - \hat{\vec{x}}_i||^2 + ||\vec{x}'_i - \hat{\vec{x}}'_i||^2)$$

2.2 Image Rectification

The epipolar geometry allows us to limit the search space of the point correspondences of any pixel on the left image or the right image. In more words, the pixel on the right image corresponding to a particular pixel on the left image should lie on the epipolar line of the left pixel point on the right image.

Now, we will estimate the 3 x 3 homographies for both left image (H) and right image (H'). The idea is to estimate the epipole on the images and send them to infinity. In such case, the images will be perfectly aligned with each other with being at a specific angle. It is explained in 2.

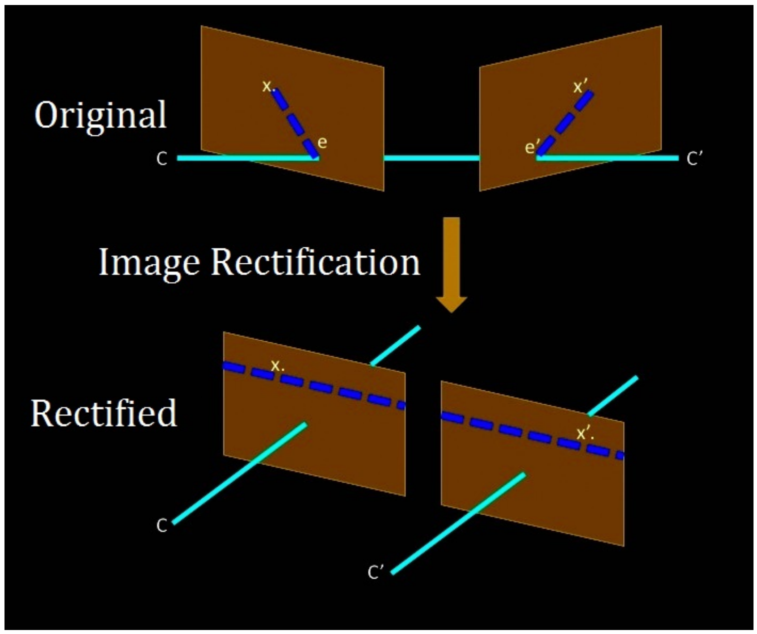

Figure 2: Rectification of images. The line in blue color are epipolar lines. The epipoles are sent to infinity in the rectified images.

Right Image

For the right image, we want to estimate the H' that sends the epipole on the right image to infinity. The steps are given as follows:

1. **Normalization:**
   In this step, we want to normalize the images so that that center pixel of both the left and right images correspond to the origin $(0, 0)$. This is achieved by premultiplying the following matrices to the point correspondences of left and right images. Where T is the normalization transformation matrix, W is the width of the image and H is the height of the image.

$$\vec{x}_{norm} = T\vec{x}, \qquad \vec{x}'_{norm} = T\vec{x}'$$

$$T = \begin{bmatrix} 1 & 0 & -W/2 \\ 0 & 1 & -H/2 \\ 0 & 0 & 1 \end{bmatrix}$$

2. **Rotation Matrix:** Now we want to estimate the rotation matrix

$$R = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

   If the right epipole $e' = [e'_x, e'_y, 1]^T$ in homogenous coordinates. We want to find R so that e' is transformed to $[f, 0, 1]^T$

3. **Epipole to Infinity:**
   The next step is to estimate a homography that transforms $[f, 0, 1]^T$ to infinity $[f, 0, 0]^T$. The following matrix G does this job.

$$G = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{f} & 0 & 1 \end{bmatrix}$$

4. **Final Homography** H'
   The final homography H' is given as

$$H' = T^{-1}GRT$$

Left Image

Once we estimated the H', we want to estimate H for left image that maps the left epipole to infinity. The following steps are followed:

1. **Compute M:**

$$M = P'P^+$$

   Where P^+ is the pseudo inverse of the matrix P.

2. **Decompose H:** H is decomposed as a product of two other matrices H_0 and H_a.

$$H_0 = H'M$$

   H_a takes the following form:

$$H_a = \begin{bmatrix} a & b & c \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

   The values of a, b, c are estimated by minimizing the following the procedure explained in the class.

3. SURF/SIFT Interest Points

SIFT and SURF interest points were extracted using the open source implementations of OpenCV. The number of interest points is reducing the NCC. The RANSAC Algorithm is used to eliminate the outliers as implemented in one of the previous homeworks.

5. Results

Unfortunately, I have implemented the algorithm and debugged it by walking through the implementations of previous best homeworks. I could not get good results.

6. Code

Listing 1: Homework 9 code

```
import cv2
import numpy as np
import os, time, sys
# from NonlinearLeastSquares import NonlinearLeastSquares as NLS
import matplotlib.pyplot as plt
from os.path import basename, dirname, splitext, join
import itertools
from scipy.optimize import least_squares

########################
####### HELPERS #########
########################

########################
### Global Variables ####
########################
x_true = None
x_prime_true = None

def get_null_vec(A):
  U, S, V = np.linalg.svd(A)
  return V.T[:,-1]

def abc_loss_fn(w):
  temp = np.sum(real_to_homo(x_prime_true) * np.array(w), axis = 1) - x_true[:, 0]
  return np.sum(temp ** 2)

def rectify_images(left, right, F):
  '''
  Description:
  Input arguments:
    * left is a dict
      # 'img': image path or np.ndarray of an image
      # 'mps': matching points. 2D np.ndarray. Rows are points. Columns are x and y coordinates.
      # 'e': 1D np.array. Epi pole
      # 'P': Camera matrix of left image. 2D 3 x 4 np.ndarray
    * right is a dict
      # 'img': image path or np.ndarray of an image
      # 'mps': matching points. 2D np.ndarray. Rows are points. Columns are x and y coordinates.
      # 'e': 1D np.array. Epi pole
      # 'P': Camera matrix of left image. 2D 3 x 4 np.ndarray
    * F: 3 x 3 np.ndarray
  Return:
  '''
  global x_true, x_prime_true
  left_img = left['img']
  left_mps = left['mps']
  left_e = left['e']
  P = left['P']
```

```

    right_img = right['img']
    right_mps = right['mps']
    right_e = right['e']
    P_prime = right['P']

    if(isinstance(left_img, str)): left_img = cv2.imread(left_img)
    if(isinstance(right_img, str)): right_img = cv2.imread(right_img)

    l_height, l_width = left_img.shape[0], left_img.shape[1]
    r_height, r_width = right_img.shape[0], right_img.shape[1]

    T_left = np.array([[1, 0, -1*l_width/2.0],[0, 1, -1*l_height/2.0],[0, 0, 1]])
    T_right = np.array([[1, 0, -1*r_width/2.0],[0, 1, -1*r_height/2.0],[0, 0, 1]])

    ## Rectifying the right image
    t_right_e = nmlz(np.dot(T_right, right_e))
    angle = np.arctan(-1*t_right_e[1]/t_right_e[0])
    print np.arctan2(t_right_e[1], -1*t_right_e[0])
    f = t_right_e[0] * np.cos(angle) - t_right_e[1] * np.sin(angle)

    G = np.array([[1, 0, 0],
          [0, 1, 0],
          [-1.0/f, 0, 1]])
    R = np.array([[np.cos(angle), -1*np.sin(angle), 0],
          [np.sin(angle), np.cos(angle), 0],
          [0, 0, 1]])
    H2 = np.dot(np.dot(G, R), T_right)
    # print 'new e right', np.dot(H2, right_e)

    r_center_rect = nmlz(np.dot(H2, [r_width/2.0, r_height/2.0, 1]))
    T2 = np.array([[1, 0, r_width/2.0 - r_center_rect[0]],
          [0, 1, r_height/2.0 - r_center_rect[1]],
          [0, 0, 1]])

    H2 = np.dot(T2, H2)

    ## Rectifying left image.
    M = np.dot(P_prime, np.linalg.pinv(P))
    H0 = np.dot(H2, M)
    left_mps_hat = homo_to_real(np.dot(H0, real_to_homo(left_mps).T).T)
    right_mps_hat = homo_to_real(np.dot(H2, real_to_homo(right_mps).T).T)
    x_prime_true = left_mps_hat
    x_true = right_mps_hat
    x = [0.0, 0., 0.]
    x = least_squares(abc_loss_fn, x).x
    print 'x', x
    # A = real_to_homo(left_mps_hat)
    # b = right_mps_hat[:, 0]
    # x = np.dot(np.linalg.pinv(A), b)

    HA = np.array([[x[0], x[1], x[2]],
          [0, 1, 0],
```

```
        [0, 0, 1]])

  H1 = np.dot(HA, H0)
  H1 = H0
  print 'x', x
  print 'H1', H1

  # l_center_rect = nmlz(np.dot(H1, [l_width/2.0, l_height/2.0, 1]))
  # print 'l_center_rect', l_center_rect
  # print 'l actual center', [l_width/2.0, l_height/2.0, 1]
  # T1 = np.array([[1, 0, l_width/2.0 - l_center_rect[0]],
  #       [0, 1, l_height/2.0 - l_center_rect[1]],
  #       [0, 0, 1]])
  # print 'T1', T1
  # H1 = np.dot(T1, H1)

  ## Find rectified F
  F_rect = np.dot(np.dot(np.linalg.inv(H2.T), F), np.linalg.inv(H1))

  ## Rectified matching points
  left_mps_rect = homo_to_real(np.dot(H1, real_to_homo(left_mps).T).T)
  right_mps_rect = homo_to_real(np.dot(H2, real_to_homo(right_mps).T).T)

  ## Rectified epi poles
  left_e_rect = nmlz(get_null_vec(F_rect))
  right_e_rect = nmlz(get_null_vec(F_rect.T))

  left['mps_rect'] =  left_mps_rect
  left['e_rect'] = left_e_rect
  left['H'] = H1

  right['mps_rect'] = right_mps_rect
  right['e_rect'] = right_e_rect
  right['H'] = H2

  return left, right, F_rect

def loss_fn(w):
  global x_true, x_prime_true
  P = np.zeros((3, 4))
  P[:,:3] = np.eye(3)

  P_prime = np.reshape(w[:12], (3, 4))

  num_points = len(w[12:])/3

  X = np.reshape(w[12:], (num_points, 3))

  x_hat = homo_to_real(np.dot(P, real_to_homo(X).T).T)
  x_prime_hat = homo_to_real(np.dot(P_prime, real_to_homo(X).T).T)

  left_err = np.linalg.norm(x_true - x_hat, axis = 1)**2
  right_err = np.linalg.norm(x_prime_true - x_prime_hat, axis = 1)**2
```

```
  cost = np.sqrt(np.sum(left_err) + np.sum(right_err))

  return cost

def skew(vec):
  try:
    assert len(vec) == 3, 'Error! vec can not have more than three elements.'
  except AssertionError as err:
    print err
    return None
  x, y, z = vec[0], vec[1], vec[2]
  return np.array([[0, -z, y],[z, 0, -x],[-y, x, 0]])

def compute_global_coordinates(left, right, F):
  '''
  Description:
  Input arguments:
    * F: 3 x 3 np.ndarray
    * left_mps: 8 x 2 np.ndarray
    * right_mps: 8 x 2 np.ndarray
    * left_shape: shape of the left image
    * right_shape: shape of the right image
  Return:
  '''
  #########################
  ### Global Variables ####
  #########################
  global x_true, x_prime_true

  left_mps = left['mps']
  left_shape = left['img'].shape

  right_mps = right['mps']
  right_shape = right['img'].shape

  ##################
  ## Left Camera ###
  ##################
  ## Compute e_left (epipole of left image)
  U, S, V = np.linalg.svd(F)
  e_left = nmlz(V.T[:,-1])
  ## Left camera matrix
  P = np.zeros((3, 4))
  P[:,:3] = np.eye(3)

  ##################
  ## Right Camera ##
  ##################
  ## Compute e_prime or e_right (epipole of right image)
  U, S, V = np.linalg.svd(F.T)
  e_right = nmlz(V.T[:,-1])
  M = np.dot(skew(e_right), F)
```

```
P_prime = np.zeros((3, 4))
P_prime[:,:3] = M
P_prime[:, 3] = e_right

# print P_prime

################################
## Estimate World Coordinates ##
################################
G = [] ## Global coordinates
for idx in range(left_mps.shape[0]):
  xl, yl = left_mps[idx, :]
  xr, yr = right_mps[idx, :]
  A = np.array([xl * P[2,:] - P[0, :],
          yl * P[2, :] - P[1, :],
          xr * P_prime[2,:] - P_prime[0, :],
          yr * P_prime[2, :] - P_prime[1, :]])
  U, S, V = np.linalg.svd(A)
  temp = V.T[:,-1]
  G.append(temp)

G = np.array(G)
G = homo_to_real(G)

# print 'Old P Prime: '
# print P_prime
# print 'Old G: '
# print G

## Initial values for non linear least squares
w = np.append(P_prime.flatten(), G.flatten())
x_true = left_mps
x_prime_true = right_mps
w_new = least_squares(loss_fn, w)

w_new = w_new.x

new_P_prime = np.reshape(w_new[:12], (3, 4))
new_G = np.reshape(w_new[12:], (G.shape[0], 3))
# print 'New P Prime: '
# print new_P_prime
# print 'New G: '
# print new_G

## Compute modified parameters
new_e_prime = nmlz(new_P_prime[:, -1])
new_F = np.dot(np.dot(skew(new_e_prime), new_P_prime), np.linalg.pinv(P))
## Compute e_left (epipole of left image)
U, S, V = np.linalg.svd(new_F)
new_e_left = nmlz(V.T[:,-1])

return new_F, new_e_left, new_e_prime, P, new_P_prime, new_G
```

```
def compute_fund_mat(left_mps, right_mps, left_shape = None, right_shape = None):
  '''
  Description:
  Input arguments:
    * left_mps: 8 x 2 np.ndarray
    * right_mps: 8 x 2 np.ndarray
    * left_shape: shape of the left image
    * right_shape: shape of the right image
  '''
  try:
    assert isinstance(left_mps, np.ndarray), 'left_mps should be numpy array'
    assert isinstance(right_mps, np.ndarray), 'right_mps should be numpy array'
    assert left_mps.shape[0] == right_mps.shape[0], 'Error! No. of rows should be same'
  except AssertionError as err:
    print err
    return None

  if(left_shape is not None):
    l_height, l_width = left_shape[0], left_shape[1]
  else:
    l_height, l_width = 0, 0

  if(right_shape is not None):
    r_height, r_width = right_shape[0], right_shape[1]
  else:
    r_height, r_width = 0, 0

  ## Normalization Matrices: Translate origin to center of the image
  T_left = np.array([[1, 0, -1*l_width/2.0],[0, 1, -1*l_height/2.0],[0, 0, 1]])
  T_right = np.array([[1, 0, -1*r_width/2.0],[0, 1, -1*r_height/2.0],[0, 0, 1]])

  ## Transform pixel coordinates so that image's center is the origin
  left_mps_t = homo_to_real(np.dot(T_left, real_to_homo(left_mps).T).T)
  right_mps_t = homo_to_real(np.dot(T_right, real_to_homo(right_mps).T).T)

  ## Form matrix A to estimate the fundamental matrix F
  A = []
  for idx in range(left_mps.shape[0]):
    xl, yl = left_mps_t[idx, :]
    xr, yr = right_mps_t[idx, :]
    temp = [xr*xl, xr*yl, xr, yr*xl, yr*yl, yr, xl, yl, 1]
    A.append(temp)
  A = np.array(A)

  ## Use SVD to solve linear least squares.
  [U, S, V] = np.linalg.svd(A, full_matrices = True)
  f = V.T[:,-1]
  f = f / f[-1]
  F = np.reshape(f, (3, 3))

  ## Condition the F, by making the determinant = 0. Zero the last eigen value.
  [U, S, V] = np.linalg.svd(F, full_matrices = True)
  S[-1] = 0.0
```

```
  F = np.dot(np.dot(U, np.diag(S)), V)

  ## Denormalization
  F = np.dot(np.dot(T_right.T, F), T_left)

  ## Note. x_prime is right and x is the left image.
  # Compute x_prime_transpose * F * x. In theory, it should be equal to zero.
  err_vals = []
  for idx in range(left_mps.shape[0]):
    xl, yl = left_mps[idx, :]
    xr, yr = right_mps[idx, :]
    temp = np.dot(np.dot([xr, yr, 1], F), [xl, yl, 1])
    err_vals.append(temp)

  # print 'Error values: ', err_vals

  return F

def real_to_homo(pts):
  # pts is a 2D numpy array of size _ x 2/3
  # This function converts it into _ x 3/4 by appending 1
  if(pts.ndim == 1):
    return np.append(pts, 1)
  else:
    return np.concatenate((pts, np.ones((pts.shape[0], 1))), axis = 1)

def homo_to_real(pts):
  # pts is a 2D numpy array of size _ x 3/4
  # This function converts it into _ x 2/3 by removing last column
  if(pts.ndim == 1):
    pts = pts / pts[-1]
    return pts[:-1]
  else:
    pts = pts.T
    pts = pts / pts[-1,:]
    return pts[:-1,:].T

def save_mps(event, x, y, flags, param):
  fac, mps = param
  if(event == cv2.EVENT_LBUTTONUP):
    mps.append([int(fac*x), int(fac*y)])
    print(int(fac*x), int(fac*y))

def create_matching_points(img_path, suff = ''):
  npz_path = img_path[:-4]+ suff + '.npz'
  flag = os.path.isfile(npz_path)
  if(not flag):
    img = cv2.imread(img_path)
    fac = max(float(int(img.shape[1]/960)), float(int(img.shape[0]/540)))
    if(fac < 1.0): fac = 1.0
    resz_img = cv2.resize(img, None, fx=1.0/fac, fy=1.0/fac, interpolation = cv2.INTER_CUBIC)
    cv2.namedWindow(img_path)
    mps = []
```

```
    cv2.setMouseCallback(img_path, save_mps, param=(fac, mps))
    cv2.imshow(img_path, resz_img)
    cv2.waitKey(0)
    np.savez(npz_path, mps = np.array(mps))
    cv2.destroyAllWindows()
  return np.load(npz_path)

def nmlz(x):
  assert isinstance(x, np.ndarray), 'x should be a numpy array'
  assert x.ndim > 0 and x.ndim < 3, 'dim of x >0 and <3'
  if(x.ndim == 1 and x[-1]!=0): return x/float(x[-1])
  if(x.ndim == 2 and x[-1,-1]!=0): return x/float(x[-1,-1])
  return x

def rem_transl(H):
  assert isinstance(H, np.ndarray), 'H should be a numpy array'
  assert H.ndim == 2, 'H should be a numpy array of two dim'
  assert H.shape[0] == H.shape[1], 'H should be a square matrix'
  H_clone = np.copy(H)
  H_clone[:-1,-1] = 0
  return H_clone

def hinv(H):
  assert isinstance(H, np.ndarray), 'H should be a numpy array'
  assert H.ndim == 2, 'H should be a numpy array of two dim'
  assert H.shape[0] == H.shape[1], 'H should be a square matrix'
  Hinv = np.linalg.inv(H)
  return Hinv / Hinv[-1,-1]

def apply_homography2(img_path, H, num_partitions = 1, suff = ''):
  if(isinstance(img_path, str)): img = cv2.imread(img_path)
  else:
    img = img_path
    img_path = 'sample.jpg'

  img[0,:], img[:,0], img[-1,:], img[:,-1] = 0, 0, 0, 0

  xv, yv = np.meshgrid(range(0, img.shape[1], img.shape[1]-1), range(0, img.shape[0], img.shape[
  img_pts = np.array([xv.flatten(), yv.flatten()]).T
  trans_img_pts = np.dot(H, real_to_homo(img_pts).T)
  ttt = homo_to_real(trans_img_pts.T).T
  _w = np.max(ttt[0, :]) - np.min(ttt[0, :])
  _h = np.max(ttt[1, :]) - np.min(ttt[1, :])
  l1, l2 = img.shape[1] / _w, img.shape[0] / _h
  K = np.diag([l1, l2, 1])
  H = np.dot(K, H)

  xv, yv = np.meshgrid(range(0, img.shape[1], img.shape[1]-1), range(0, img.shape[0], img.shape[
  img_pts = np.array([xv.flatten(), yv.flatten()]).T
  trans_img_pts = np.dot(H, real_to_homo(img_pts).T)
  trans_img_pts = homo_to_real(trans_img_pts.T).astype(int)

  xmin, ymin = np.min(trans_img_pts[:,0]), np.min(trans_img_pts[:,1])
```

```
  xmax, ymax = np.max(trans_img_pts[:,0]), np.max(trans_img_pts[:,1])
  W_new = xmax - xmin
  H_new = ymax - ymin

  img_new = np.zeros((H_new+1, W_new+1, 3), dtype = np.uint8)
  print 'Shape of new image: ', img_new.shape

  x_batch_sz = int(W_new/float(num_partitions))
  y_batch_sz = int(H_new/float(num_partitions))
  for x_part_idx in range(num_partitions):
    for y_part_idx in range(num_partitions):
      x_start, x_end = x_part_idx*x_batch_sz, (x_part_idx+1)*x_batch_sz
      y_start, y_end = y_part_idx*y_batch_sz, (y_part_idx+1)*y_batch_sz
      xv, yv = np.meshgrid(range(x_start, x_end), range(y_start, y_end))
      xv, yv = xv + xmin, yv + ymin
      img_new_pts = np.array([xv.flatten(), yv.flatten()]).T
      trans_img_new_pts = np.dot(hinv(H), real_to_homo(img_new_pts).T)
      trans_img_new_pts = homo_to_real(trans_img_new_pts.T).astype(int)
      trans_img_new_pts[:,0] = np.clip(trans_img_new_pts[:,0], 0, img.shape[1]-1)
      trans_img_new_pts[:,1] = np.clip(trans_img_new_pts[:,1], 0, img.shape[0]-1)
      img_new_pts = img_new_pts - [xmin, ymin]
      # This is the bottle nect step. It takes the most time.
      img_new[img_new_pts[:,1].tolist(), img_new_pts[:,0].tolist(), :] = img[trans_img_new_pts[:

  fname, ext = tuple(os.path.basename(img_path).split('.'))
  write_filepath = os.path.join(os.path.dirname(img_path), fname+suff+'.'+ext)
  print write_filepath
  cv2.imwrite(write_filepath, img_new)

def apply_homography(img_path, H, num_partitions = 1, suff = ''):
  if(isinstance(img_path, str)): img = cv2.imread(img_path)
  else:
    img = img_path
    img_path = 'sample.jpg'

  img[0,:], img[:,0], img[-1,:], img[:,-1] = 0, 0, 0, 0

  xv, yv = np.meshgrid(range(0, img.shape[1], img.shape[1]-1), range(0, img.shape[0], img.shape[
  img_pts = np.array([xv.flatten(), yv.flatten()]).T
  trans_img_pts = np.dot(H, real_to_homo(img_pts).T)
  trans_img_pts = homo_to_real(trans_img_pts.T).astype(int)

  print 'trans_img_pts'
  print trans_img_pts

  xmin, ymin = np.min(trans_img_pts[:,0]), np.min(trans_img_pts[:,1])
  xmax, ymax = np.max(trans_img_pts[:,0]), np.max(trans_img_pts[:,1])
  W_new = xmax - xmin
  H_new = ymax - ymin

  img_new = np.zeros((H_new+1, W_new+1, 3), dtype = np.uint8)
  print 'Shape of new image: ', img_new.shape
```

```
  x_batch_sz = int(W_new/float(num_partitions))
  y_batch_sz = int(H_new/float(num_partitions))
  for x_part_idx in range(num_partitions):
    for y_part_idx in range(num_partitions):
      x_start, x_end = x_part_idx*x_batch_sz, (x_part_idx+1)*x_batch_sz
      y_start, y_end = y_part_idx*y_batch_sz, (y_part_idx+1)*y_batch_sz
      xv, yv = np.meshgrid(range(x_start, x_end), range(y_start, y_end))
      xv, yv = xv + xmin, yv + ymin
      img_new_pts = np.array([xv.flatten(), yv.flatten()]).T
      trans_img_new_pts = np.dot(hinv(H), real_to_homo(img_new_pts).T)
      trans_img_new_pts = homo_to_real(trans_img_new_pts.T).astype(int)
      trans_img_new_pts[:,0] = np.clip(trans_img_new_pts[:,0], 0, img.shape[1]-1)
      trans_img_new_pts[:,1] = np.clip(trans_img_new_pts[:,1], 0, img.shape[0]-1)
      img_new_pts = img_new_pts - [xmin, ymin]
      # This is the bottle nect step. It takes the most time.
      img_new[img_new_pts[:,1].tolist(), img_new_pts[:,0].tolist(), :] = img[trans_img_new_pts[:

  fname, ext = tuple(os.path.basename(img_path).split('.'))
  write_filepath = os.path.join(os.path.dirname(img_path), fname+suff+'.'+ext)
  print write_filepath
  cv2.imwrite(write_filepath, img_new)

def extract_kps(image, ftype = 'sift', sigma = 1.414):
    ################
    # Description:
    #   Find interest points (keypoints) and descriptors
    # Input:
    #   image: RGB image. 3D ndarray (H x W x 3).
    #   ftype: 'sift' or 'surf'
    #   sigma: scale applied to the image
    # Output:
    #   A tuple (keypoints, features)
    #   keypoints: ndarray (Z x 2). each row has (row_idx, col_idx)
    #   features:  ndarray (Z x desc_size**2)
    #   Z is no. of interest points
    ################

    ## Assertion
    assert image.ndim == 3, 'img is a 3D ndarray (RGB image: H x W x 3)'

    ## convert the image to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    if(ftype.lower() == 'sift'):
        descriptor = cv2.xfeatures2d.SIFT_create()
        # kps (cv2.KeyPoint object) and features (ndarray).
        (kps, features) = descriptor.detectAndCompute(image, None)
        kps = np.float32([kp.pt for kp in kps])
    elif ftype.lower() == 'surf':
        descriptor = cv2.xfeatures2d.SURF_create()
        # kps (cv2.KeyPoint object) and features (ndarray).
        (kps, features) = descriptor.detectAndCompute(image, None)
        kps = np.float32([kp.pt for kp in kps])
```

```
    else:
        raise ValueError('Unknown feature type')

    # kps: ndarray (Z x 2); features: ndarray (Z x 128)
    return (kps, features)

def mean_normalize(M, axis = 0):
    ## First, substract mean and next, normalize the rows/columns.
    # Mean normalize matrix M (_ x k) in rows
    if(axis == 1): M = M.transpose() # (k x _)
    M -= np.mean(M, axis = 0)

    norms = np.linalg.norm(M, axis = 0)
    norms[norms == 0.0] = 1e-10

    M /= norms
    if(axis == 1): M = M.transpose()
    return M

def dist_mat_vec(M, vec, method = 'ncc'):
    # Compute distance between each row of 'M' with 'vec'
    # method: 'ncc', 'dot', 'ssd'
    # M : ndarray ( _ x k); vec: (1 x k)
    # Returns a 1D numpy array of distances.
    if(method.lower() == 'ssd'):
        return np.linalg.norm(M - vec, axis = 1)
    elif(method.lower() == 'ncc'):
        # Mean Normalizing rows of M
        M = mean_normalize(M, axis = 1)
        # Mean normalizing vec
        vec = vec - np.mean(vec)
        vect = vec / np.linalg.norm(vec)
        return np.dot(M, vec)
    elif(method.lower() == 'dot'):
        return np.dot(M, vec)

def dist_mat_mat(M1, M2, method = 'ncc'):
    # M1, M2 --> ndarray (y1 x k) and (y2 x k)
    # Returns y1 x y2 ndarray with the distances.
    # If y1 and y2 are huge, it might run into MemoryError
    D = np.zeros((M1.shape[0], M2.shape[0]))
    if(method.lower() == 'ncc'):
        M1 = mean_normalize(M1, axis = 1)
        M2 = mean_normalize(M2, axis = 1)
        method = 'dot'
    for idx2 in range(M2.shape[0]):
        D[:, idx2] = dist_mat_vec(M1, M2[idx2, :], method = method)
    return D

def filter_kps(kpA, kpB, featuresA, featuresB, method = 'ncc', thresh = 0.97):
    # Filter the keypoints and the descriptors by thresholding.
    # Returns the matches. List of tuples. (a_idx, b_idx). Point correspondences.
    print len(featuresA), len(featuresB)
```

```
    if(method.lower() == 'ncc'):
        featuresA = mean_normalize(featuresA, axis = 1)
        featuresB = mean_normalize(featuresB, axis = 1)
        method = 'dot'

    matches = []
    for idxB in range(featuresB.shape[0]):
        temp = dist_mat_vec(featuresA, featuresB[idxB, :], method = method)
        temp[temp<thresh] = 0.0
        ## Append the ones that pass the threshold
        if(np.max(temp) == 0.0): continue
        else: matches.append((np.argmax(temp), idxB))

    return matches

def draw_matches_one_to_one(imageA, imageB, kpsA, kpsB):
    ########
    # kpsA and kpsB should have same no. of rows.
    # There is one to one correspondence between rows of kpsA and kpsB
    ########
    # initialize the output visualization image
    (hA, wA) = imageA.shape[:2]
    (hB, wB) = imageB.shape[:2]
    vis = np.zeros((max(hA, hB), wA + wB, 3), dtype="uint8")
    vis[0:hA, 0:wA] = imageA
    vis[0:hB, wA:] = imageB

    # loop over the matches
    for ptA, ptB in zip(kpsA, kpsB):
        ptA = (int(ptA[0]), int(ptA[1]))
        ptB = (int(ptB[0]) + wA, int(ptB[1]))
        color = tuple(np.random.randint(0, 255, 3).tolist())
        cv2.line(vis, ptA, ptB, color, 2)

    # return the visualization
    return vis

def draw_matches(imageA, imageB, kpsA, kpsB, matches):
    # initialize the output visualization image
    (hA, wA) = imageA.shape[:2]
    (hB, wB) = imageB.shape[:2]
    vis = np.zeros((max(hA, hB), wA + wB, 3), dtype="uint8")
    vis[0:hA, 0:wA] = imageA
    vis[0:hB, wA:] = imageB

    # loop over the matches
    for queryIdx, trainIdx in matches:
        # only process the match if the keypoint was successfully
        # matched
        # draw the match
        ptA = (int(kpsA[queryIdx][0]), int(kpsA[queryIdx][1]))
        ptB = (int(kpsB[trainIdx][0]) + wA, int(kpsB[trainIdx][1]))
        color = tuple(np.random.randint(127, 255, 3).tolist())
```

```
        cv2.line(vis, ptA, ptB, color, 1)

    # return the visualization
    return vis

def run(image1_path, image2_path, ftype = 'sift', method = 'ncc', \
    thresh = 0.97, sigma = 1.414, write_flag = False):
    img1 = cv2.imread(image1_path)
    img2 = cv2.imread(image2_path)

    kps1, features1 = extract_kps(img1, ftype = ftype, sigma = sigma)
    kps2, features2 = extract_kps(img2, ftype = ftype, sigma = sigma)

    # start = time.time()
    matches = filter_kps(kps1, kps2, features1, features2, method = method, thresh = thresh)
    print 'No. of matches: ', len(matches)
    # print 'Filter Kps: %.02f secs'%(time.time()-start)

    vis = draw_matches(img1, img2, kps1, kps2, matches)

    ## Obtain keypoint matches
    matches = np.array(matches)
    kps1, kps2 = np.array(kps1), np.array(kps2)
    ord_kps1 = kps1[matches[:, 0], :]
    ord_kps2 = kps2[matches[:, 1], :]
    # Format of kp_matches: _ x 4 np.ndarray.
    # Columns 0 and 1 for [x1, y1] of image 1
    # Columns 2 and 3 for [x1, y1] of image 2
    kp_matches = np.append(ord_kps1, ord_kps2, axis = 1)

    delta = 0.5
    while True:
        new_kp_matches, H = ransac(kp_matches, delta = delta, eps = 0.20)
        if H is None: delta = delta * 2
        else: break

    new_kps1 = new_kp_matches[:,:2].tolist()
    new_kps2 = new_kp_matches[:,2:].tolist()

    print 'No. matches: ', len(new_kps1)

    print 'Performing LM: '
    lmres = LM_Minimizer(new_kp_matches, H)

    new_H = np.squeeze(np.asarray(lmres['parameter_values']))
    new_H = np.append(new_H, np.array([1])).reshape(3, 3)
    new_H = nmlz(new_H)

    # mosaic_two_images(img1, img2, new_H)
    # mosaic_two_images(img2, img1, hinv(new_H))

    vis = draw_matches_one_to_one(img1, img2, new_kps1, new_kps2)

```

```
    #####
    if(write_flag):
        fname = splitext(basename(image1_path))[0] + '_' + splitext(basename(image2_path))[0]
        fname = fname + '_' + str(ftype) + '_' + str(int(sigma*1000)) + '_' + str(int(thresh*100
        fname_path = join(dirname(image1_path), fname)
        print 'Writing to: ', fname_path
        cv2.imwrite(fname_path, vis)

    return vis, new_H

def count_inliers(point_corresps, H, delta = 40):
  ####################
  # Input:
  #
  #   point_corresps: np.ndarray of shape _ x 4
  #       Column 0 and 1 correspond to [x_coordinate, y_coordinate] of img1
  #       Column 2 and 3 correspond to [x_coordinate, y_coordinate] of img2
  #       It is point correspondences between two images.
  #
  #   H: Homography (np.ndarray) of shape 3 x 3
  #
  #   delta: decision threshold. Either threshold on SSD or NCC to
  #       determine if a corresp. is an inlier or outlier.
  #       Default value is 40 pixels.
  #
  # Return:
  #
  #   inlier_sz: size of the inlier set. No. of points that are inliers.
  #
  #   inlier_ids: a np array containing indices of points in inlier set
  #
  ####################

  pts1 = point_corresps[:,:2]
  pts2 = point_corresps[:,2:]

  homo_pts2 = real_to_homo(pts2)
  trans_homo_pts2 = np.dot(H, homo_pts2.transpose())
  trans_pts2 = homo_to_real(trans_homo_pts2.transpose())

  err = np.linalg.norm(pts1 - trans_pts2, axis = 1)

  inlier_sz = int(np.sum(err < delta))

  return inlier_sz, np.nonzero(err < delta)[0]

def ransac(point_corresps, param_p = 0.99, eps = 0.1, param_n = 8, delta = 40):
  ####################
  # Input:
  #
  #   point_corresps: np.ndarray of shape _ x 4
  #       Column 0 and 1 correspond to [x_coordinate, y_coordinate] of img1
  #       Column 2 and 3 correspond to [x_coordinate, y_coordinate] of img2
```

```
#       It is point correspondences between two images.
#
#   p: prob. that at least one of N trials will be free of outliers.
#       Default value is 0.99/
#
#   eps: prob. that a pt. corresp. is an outlier
#
#   n: min. no. of point correspondences needed to estimate the homography
#
#   delta: decision threshold. Either threshold on SSD or NCC to
#       determine if a corresp. is an inlier or outlier.
#       Default value is 40 pixels.
#
# Return:
#   H: Homography from 2 --> 1. np.ndarray of shape 3 x 3.
#
#   new_matches: Point correspondences of points in inlier set.
#       Datatype is similar to point_corresps but with only inliers.
#####################

# Determine num_trials (N). No. of trials or times we need to run RANSAC
#   so that at least one trial will contain all inliers
N = np.int16(np.log(1 - param_p) / np.log(1 - (1 - eps)**param_n))

# thresh_inlier_sz (M)
#   A minimum size of inlier set that is acceptable.
n_total = len(point_corresps)
M = np.int((1 - eps) * n_total)

print 'Len. of point_corresps: ', len(point_corresps)
print 'No. of trials (N): ', N
print 'Min. acceptable size of inlier set (M): ', M

trial_info = []

for tr_idx in range(N):
  ## Find 'param_n' point correspondences at random
  tr_match_ids = np.random.randint(0, n_total, param_n)
  # If point correspondences repeat, try until you get unique ids.
  while(len(tr_match_ids) < param_n):
    tr_match_ids = np.random.randint(0, n_total, param_n)

  ## Find homography with the obtained point correspondences
  tr_matches = point_corresps[tr_match_ids, :]
  tr_H, _ = find_homography_2d(tr_matches[:,:2], tr_matches[:,2:])

  ## Find the size of inlier set
  inlier_sz, _ = count_inliers(point_corresps, tr_H, delta = delta)

  # If size of inlier set exceed M, store the trial information.
  if(inlier_sz >= M):
    trial_info.append((inlier_sz, tr_match_ids))
```

```
    if len(trial_info) == 0: return None, None
    # Find the inlier set with maximum inlier size
    inlier_sz_list, inlier_pt_ids = zip(*trial_info)
    best_inlier_tr_idx = np.argmax(inlier_sz_list)
    best_inlier_ids = inlier_pt_ids[int(best_inlier_tr_idx)]

    print '% of inliers:', np.max(inlier_sz_list)/float(n_total)

    ## Estimate the homography with best inlier ids (only param_n corresp.)
    temp_matches = point_corresps[best_inlier_ids, :]
    temp_H, _ = find_homography_2d(temp_matches[:,:2], temp_matches[:,2:])

    ## Find all inlier ids and estimate the homography
    _, all_inlier_ids = count_inliers(point_corresps, temp_H, delta = delta)
    new_matches = point_corresps[all_inlier_ids, :]
    H, _ = find_homography_2d(new_matches[:,:2], new_matches[:,2:])

    return new_matches, H


####################
##### MAIN ##########
####################

import cv2
import numpy as np
from helpers import *

left = {}
right = {}

left_path = 'pair1\\left.jpg'
right_path = 'pair1\\right.jpg'

left['mps'] = create_matching_points(left_path, suff = '')['mps']
right['mps'] = create_matching_points(right_path, suff = '')['mps']

left['img'] = cv2.imread(left_path)
right['img'] = cv2.imread(right_path)
l_height, l_width, _ = left['img'].shape
r_height, r_width, _ = right['img'].shape

F = compute_fund_mat(left['mps'], right['mps'], left['img'].shape, right['img'].shape)
if(F is None): sys.exit('Error! F is None')

F, e_left, e_prime, P, P_prime, G = compute_global_coordinates(left, right, F)

left['e'] = e_left
left['P'] = P

right['e'] = e_prime
right['P'] = P_prime
```

```
left_rect, right_rect, F_rect = rectify_images(left, right, F)

print 'Left'
print nmlz(left_rect['H'])

print 'Right'
print nmlz(right_rect['H'])

H_left = nmlz(left_rect['H'])
H_right = nmlz(right_rect['H'])

apply_homography(left['img'], H_left, suff = '_left')
apply_homography(right['img'], H_right, suff = '_right')
```