Homework - 8

Sriram Karthik Badam

December 4, 2012

1. **Projective Reconstruction**

In this homework, we reconstruct the 3D structure of an image using the point correspondences obtained by edge detection and NCC measure. There are a series of steps involved to reconstruct the 3D structure which will be discussed in the next section.

Algorithm: Although its possible to reconstruct the 3D structure using multiple images, for the sake of simplicity, I am using two images taken using my mobile camera.

(a) Estimate the fundamental matrix (F) using the correspondences inputted by the user manually. Let the correspondences be $(\vec{x_i}, \vec{x_i}')$, then

$$\vec{x_i}' F \vec{x_i} = 0$$

$$\begin{bmatrix} x'x & x'y & x' & xy' & y'y & y' & x & y & 1 \end{bmatrix} \vec{f} = 0$$

where $\vec{f} = (F_{11}, F_{12}, F_{13}, F_{21}, F_{22}, F_{23}, F_{31}, F_{32}, F_{33})^T$ are the elements of the fundamental matrix. We require 8 or more such correspondences to compute F.
We build a matrix A with rows as the left hand elements of above expression for each user inputted correspondence. Note:

i. Its required to do some normalization before doing the above step to ensure a better estimate of F for various scales of input image. We find the normalization matrices T_1 , T_2 which transform the points $(\vec{x_i}, \vec{x_i}')$ to $(\vec{\hat{x}_i}, \vec{\hat{x}_i}')$.

ii. When the points are normalized, after the computation of F, it has to be denormalized.

$$F = T_2^T F T_1$$

iii. The rank of the Fundamental matrix should be 2 which can be ensured by doing an Singular Value Decomposition and then making the last eigen value zero.

$$F = UDV^T$$

(b) Compute the epipoles $(\vec{e}, \vec{e}')$ using the Fundamental matrix (F). They are the right and left null vectors of the Fundamental matrix and can be calculated using Singular Value Decomposition.

(c) Calculate the Camera matrices P, P'

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

$$P' = \begin{bmatrix} 0 & -e'_3 & e'_2 & e'_1 \\ e'_3 & 0 & -e'_1 & e'_2 \\ -e'_2 & -e'_1 & 0 & e'_3 \end{bmatrix}$$

where e' = $(e'_1, e'_2, e'_3)^T$ is the second epipole.

(d) Image Rectification:
To do an efficient reconstruction, its required to perform rectification. Through image rectification, we transform the epipoles to infinity(on x-axis). This makes the epipolar lines to be parallel to one other and also parallel to x axis.

i. Shift the second image centre to origin.(T)

ii. Rotate the second epipole such that its on x-axis.(R)

$$e' = \begin{bmatrix} f \\ 0 \\ 1 \end{bmatrix}$$

iii. Transform the epipole to a point on infinity on x-axis(G)

iv. Restore the centre of the image back to the original (T_2)
The homography that the above steps (H_2)

$$H_2 = T_2GRT$$

v. For the first image, compute H_1 such that it minimizes the least-squares method.

$$\sum_i distance(H_1 x_i, H_2 x'_i)$$

This is explained in detail in Section 11.12.2 in the text book, and the relevant expressions that guide this process are

$$M = P'P^+$$

$$H_0 = H_2 M$$

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

where (a, b, c) minimize the following sum. If $\vec{\hat{x}_i} = (\hat{x_i}, \hat{y_i})^T$ and $\vec{\hat{x}'_i} = (\hat{x_i}', \hat{y_i}')^T$ are the points on two images.

$$\sum_i (a\hat{x_i} + b\hat{y_i} + c - \hat{x_i}')^2$$

Then the value of H_1 is $H_A H_0$

(e) Interest Point Detection:

i. The interest points in the image are computed by using a canny edge detector. The idea is to find correspondences between interest points of both images and use Triangulation method to compute 3D points.

ii. Once the interest points are found, we can filter them for correspondences by applying the following constraints on them.

A. Epipolar constraint: If (x_i, x_i') are two feature points then

$$x_i'^T F x_i = 0$$

B. Thresholding on the NCC score - Normalized Cross Correlation (discussed in hw4)

C. A Threshold on the distance between the point coordinates.

(f) Triangulation Method:

i. If $\vec{x_i} = (x_i, y_i)^T$ and $\vec{x_i'} = (x_i', y_i')^T$ are the points on two images, then using the parameters P, P' we can compute the 3D coordinates X_i. In fact, its the null vector of the following matrix

$$A = \begin{bmatrix} x_i\vec{p}^{3T} & -\vec{p}^{1T} \\ y_i\vec{p}^{3T} & -\vec{p}^{2T} \\ x_i'\vec{p}^{3T} & -\vec{p}^{1T} \\ y_i'\vec{p}^{3T} & -\vec{p}^{2T} \end{bmatrix}$$

where $(\vec{p}^1\vec{p}^2\vec{p}^3)$ are the columns of P'.
Note: Since P corresponds to a camera at origin it never appears in any expressions obviously.

ii. The parameters can be improved upon using Levenberg Marquardt method. The error function for LM Method compares the reprojection of the world points on image planes using P, P'. I used Levmar, a Levenberg Marquardt method implementation in C. A python wrapper was integrated to access the native levmar methods.

iii. The 3D points are shown using the matPlotLib library of python to make 3D plots.

(g) Observations:

i. The program runs really slow with images of high resolution because the number of interest points explode and the process of correspondence finding is exhaustive.

ii. The Image rectification is dependent on the user-inputted correspondences and a bad set of input points leads to a bad orientation of the rectified images.

iii. A better(maybe)/faster way is to use SURF.

iv. The final 3D reconstruction was found to have 1 pixel error in average for every correspondence. **This means that every world point when reprojected onto image planes, was off by atmost one pixel.**

2. Results:

Initial Fundamental Matrix

[[-2.86226712e-07 -1.57753788e-05 -4.11866949e-04]

[1.04282323e-05 -3.20950250e-06 1.22212416e-02]

[1.69240610e-04 -1.29329886e-02 1.36281928e-01]]

Initial epipole e:

[[-1.17342077e+03]

[-4.81779742e+00]

[1.00000000e+00]]

Initial epipole e':

[[-811.98505384]

[-38.51586846]

[1.]]

Initial first Camera matrix P:

[[1. 0. 0. 0.]

[0. 1. 0. 0.]

[0. 0. 1. 0.]]

Initial second image Camera matrix P':

[[-6.52887731e-03 4.98128497e-01 -5.26123805e+00 -8.11985054e+02]

[1.37420560e-01 -1.05014092e+01 1.10658477e+02 -3.85158685e+01]

[-8.47859301e-03 1.99846565e-03 -9.93932894e+00 1.00000000e+00]]

Number Of Iterations done by levmar for the user inputted correspondences= 298

Final Fundamental Matrix:

[[1.47403585e-21 -1.89583851e-19 -5.61882820e-06]

[-9.14795583e-20 1.10167223e-18 -1.39352125e-02]

[3.78248296e-07 1.39358528e-02 2.68811330e-02]]

Final second image camera matrix P':

[[-6.16185789e-01 7.21230782e-01 -6.52341659e+00 -8.16449871e+02]

[-1.55112288e+00 -1.07358699e+01 1.10008672e+02 1.05483630e+02]

[-3.99984030e-02 5.07387752e-03 -1.01384341e+01 3.66357161e+00]]

Figure 1: input image 1

Figure 2: input image 2

Note: The background is deliberately made white, even for the rectified images, to fool the corner detector to not recognize an edge in background.

.

Figure 3: User Selected points

Figure 4: User Selected points

Figure 5: Rectified Image 1

Figure 6: Rectified image 2

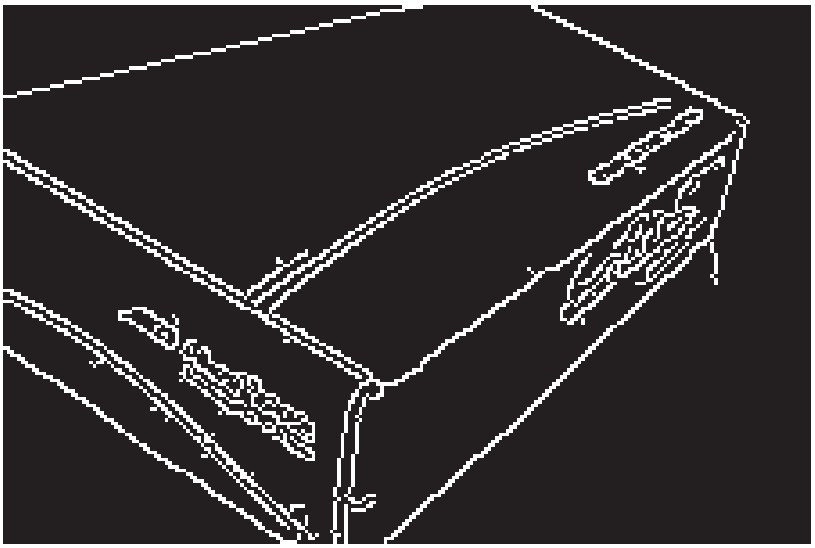

Figure 7: Edges/Interest points in Image 1

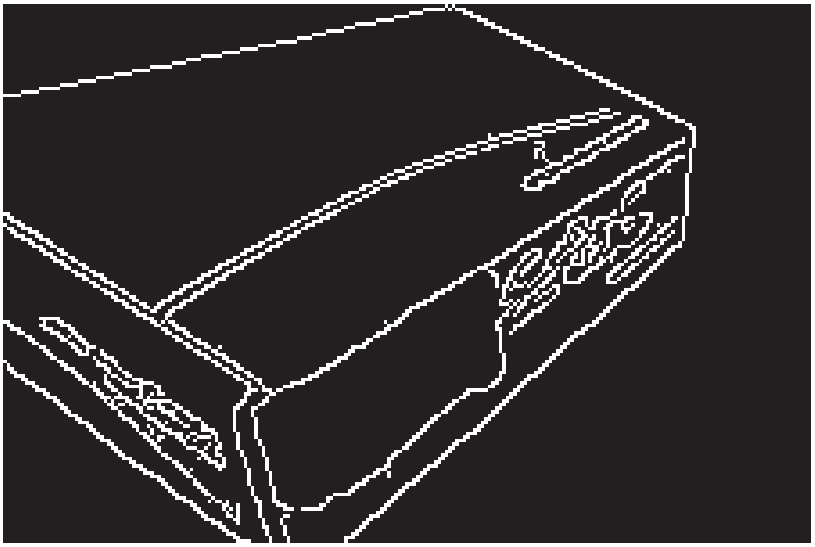

Figure 8: Edges/Interest points in image 2

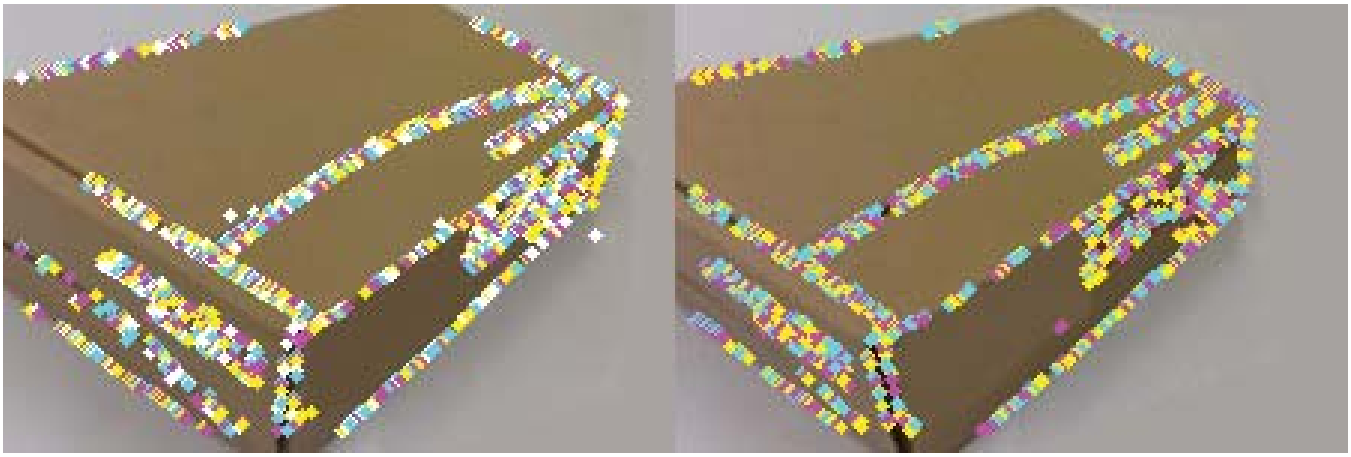

Figure 9: The Correspondences in images are shown here. Only the corresponding points are plotted and these points are used for 3D reconstruction

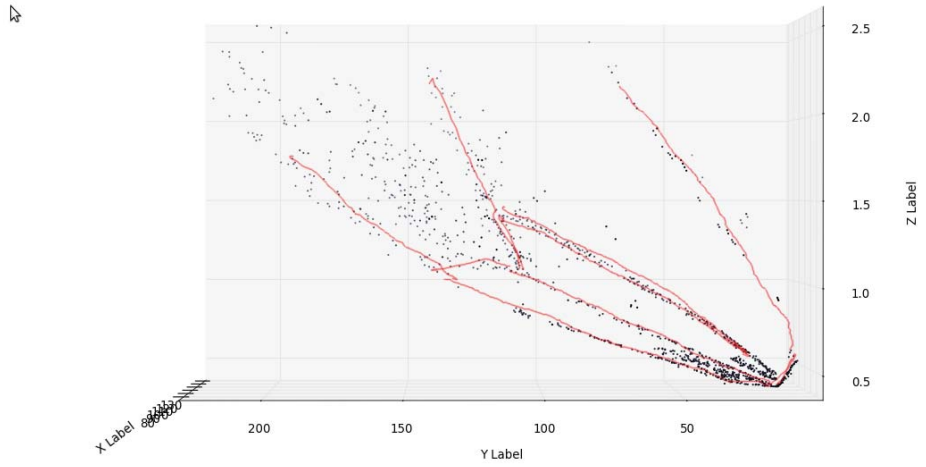

Figure 10: 3D plot done on python,. Explained in next image

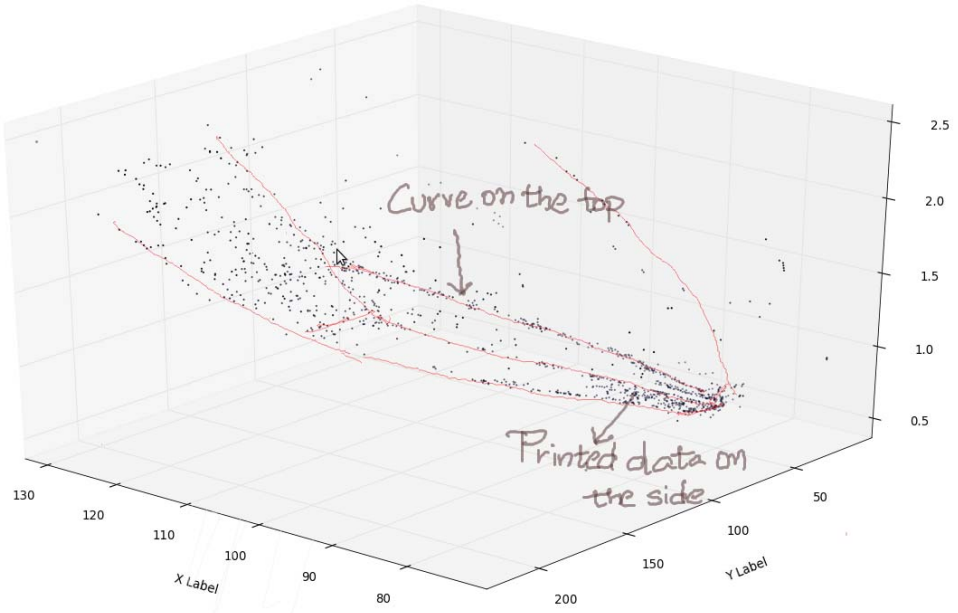

Figure 11: Note: **The correspondences found don't include most part of the back edges of the box, so they are not plotted. Only lines that are visible are the front ones and the curve on the top**

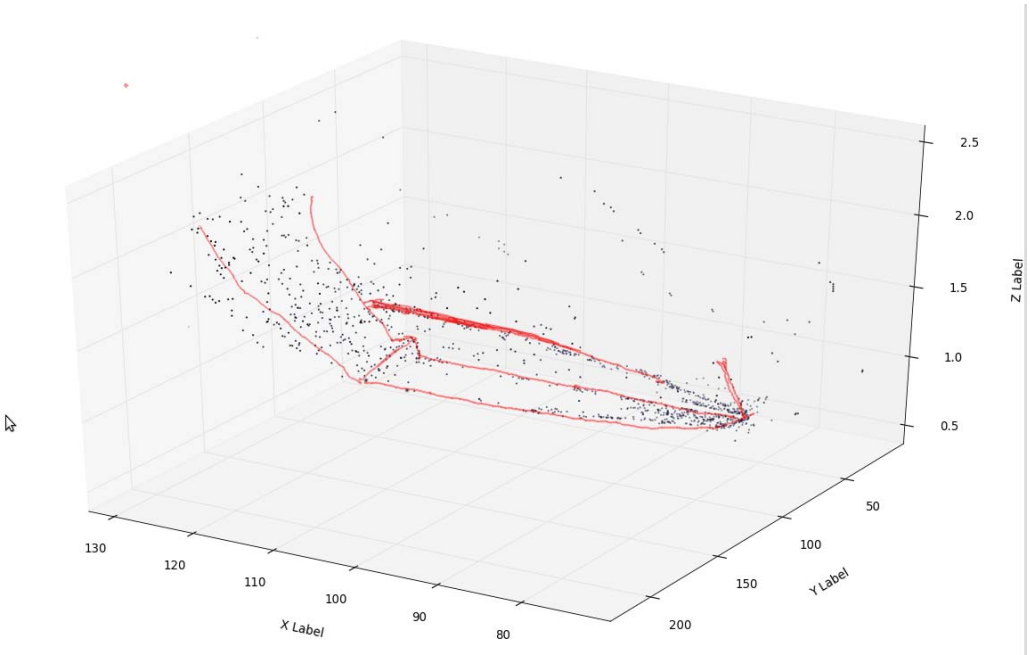

Figure 12: Other views: 3D plots done on python

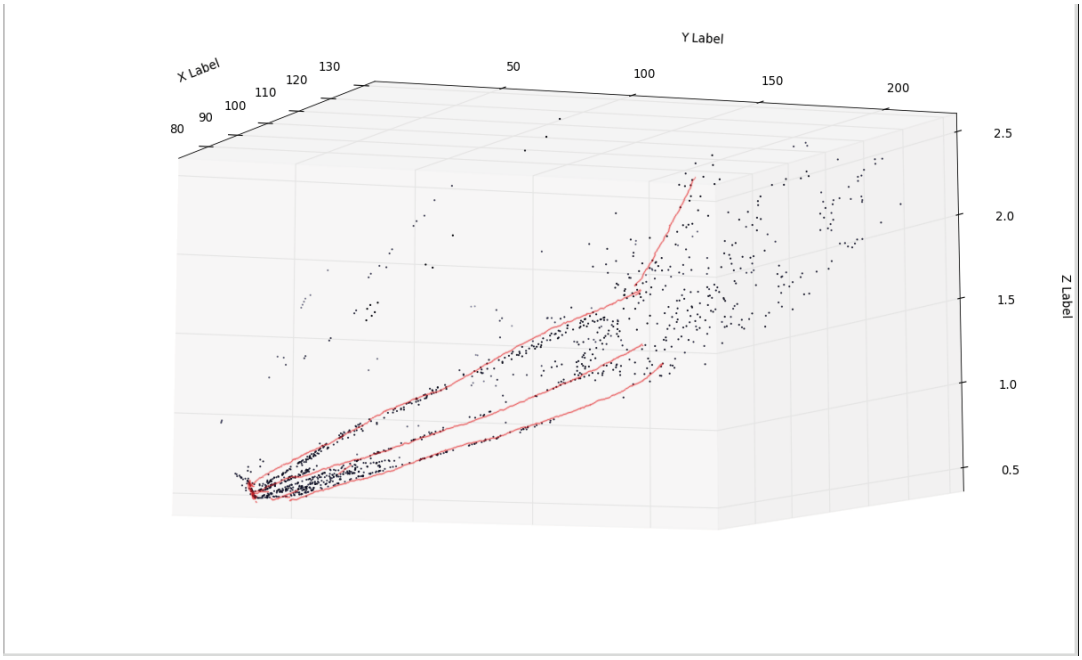

Figure 13: Other views: 3D plots done on python

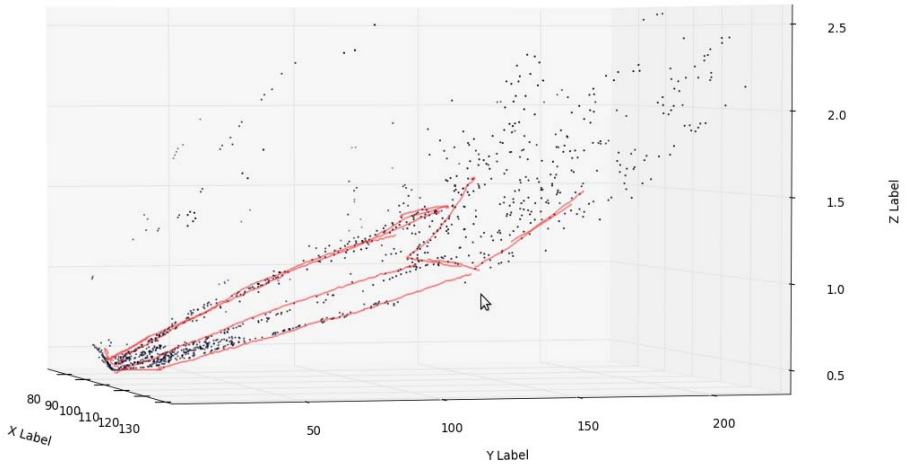

Figure 14: Other views: 3D plots done on python

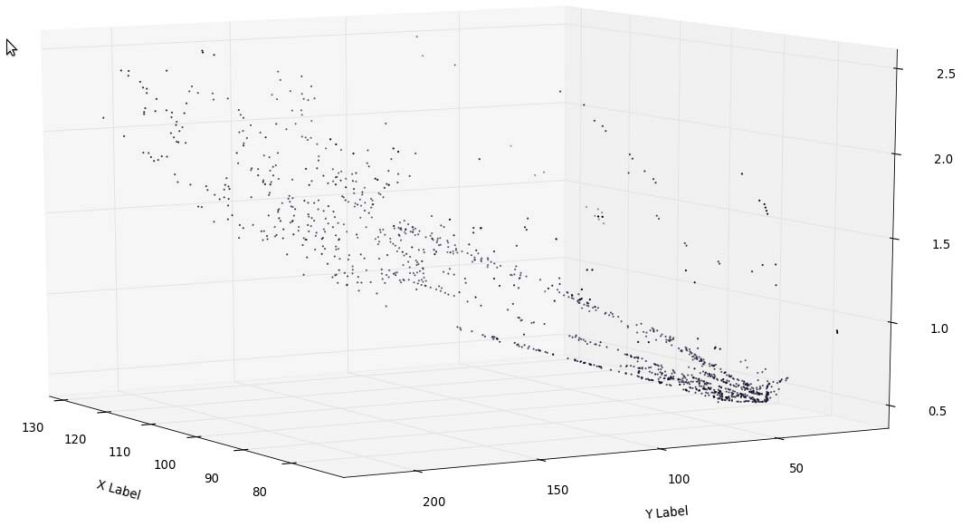

Figure 15: Other views: 3D plots done on python

3. Code : reconstruction.py

```
#!/usr/bin/python
#
# Author: Sriram Karthik Badam
# Date: Nov 25, 2012
#
import sys, os
import cv
import numpy as np
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt
import levmar


#CONSTANTS, AND THRESHOLDS
TOTAL_POINTS = 12
left_image = 0
right_image = 0
pointCount = 0

NCC_WINDOW_SIZE = 13
NCC_THRESHOLD = 0.7

#feature class
class Feature(object):
```

```
  #initializing
  def __init__(self, x, y):
    self.x = x
    self.y = y
    self.match = -1
    self.score = -1
    self.neighbor = 0


"""
This function collects the coordinates of the points selected by the user
"""

def on_mouse (event, x, y, flags, param):
  global TOTAL_POINTS, pointCount
  (image, points) = param
  #got a new point
  font = cv.InitFont(cv.CV_FONT_HERSHEY_SIMPLEX, 1, 0.5, 0, 1)
  if event == cv.CV_EVENT_LBUTTONDOWN:
    if pointCount < TOTAL_POINTS:
      point = (int(x),int(y))
      cv.Circle(image, point, 2, (0, 255, 0), 1, 8, 0)
      #cv.PutText(image,`pointCount+1`, point, font, 0)
      cv.mSet(points, pointCount, 0, int(x))
      cv.mSet(points, pointCount, 1, int(y))
      cv.mSet(points, pointCount, 2, 1)
      cv.ShowImage("Input Image", image)
      pointCount = pointCount + 1

"""
Computes the correspondence score of two interest points
The higher the score better is the correspondence
"""

def nccScore(f1, f2):
  global NCC_WINDOW_SIZE #NCC_WINDOW_SIZE of NCC window
  mean1 = cv.Avg(f1)
  mean2 = cv.Avg(f2)
  f1_sub_mean = cv.CreateMat(NCC_WINDOW_SIZE, NCC_WINDOW_SIZE, cv.CV_64FC1)
  f2_sub_mean = cv.CreateMat(NCC_WINDOW_SIZE, NCC_WINDOW_SIZE, cv.CV_64FC1)
  cv.SubS(f1, mean1, f1_sub_mean);
  cv.SubS(f2, mean2, f2_sub_mean);

  #calculates numerator of the ncc score fraction
  ncc_numerator = cv.CreateMat(NCC_WINDOW_SIZE, NCC_WINDOW_SIZE, cv.CV_64FC1)
  cv.Mul(f1_sub_mean, f2_sub_mean, ncc_numerator)
  numerator = cv.Sum(ncc_numerator)

  #calculates denominator
  magnitude1 = cv.CreateMat(NCC_WINDOW_SIZE, NCC_WINDOW_SIZE, cv.CV_64FC1)
  magnitude2 = cv.CreateMat(NCC_WINDOW_SIZE, NCC_WINDOW_SIZE, cv.CV_64FC1)
  cv.Pow(f1_sub_mean, magnitude1, 2)
```

```
  sum1 = cv.Sum(magnitude1)
  cv.Pow(f2_sub_mean, magnitude2, 2)
  sum2 = cv.Sum(magnitude2)
  denominator = np.sqrt(sum1[0]*sum2[0])

  score = numerator[0]/denominator
  return score


"""
prints a cvMat variable
"""
def printMatrix(mat):
  temp_array = np.asarray(mat[:,:])
  print temp_array


"""
fills the 'neighbor' variable with the neighboring pixel values of a given pixel.
"""
def getNeighbor(x, y, grey_scale_image, neighbor):
  global NCC_WINDOW_SIZE
  for i in range(-int(NCC_WINDOW_SIZE/2), int(NCC_WINDOW_SIZE/2+1)):
    for j in range(-int(NCC_WINDOW_SIZE/2), int(NCC_WINDOW_SIZE/2+1)):
      cv.mSet(neighbor, int(NCC_WINDOW_SIZE/2) + j, int(NCC_WINDOW_SIZE/2) + i,
          grey_scale_image[y+j, x+i])


"""
Normalizes the user inputed points to calculate Fundamental matrix F
"""
def normalize(points):
  global TOTAL_POINTS
  T = cv.CreateMat(3, 3, cv.CV_64FC1)
  mean_x = 0.0
  mean_y = 0.0
  for i in range(TOTAL_POINTS):
    mean_x+=cv.mGet(points, i, 0)
    mean_y+=cv.mGet(points, i, 1)
  mean_x/=float(TOTAL_POINTS)
  mean_y/=float(TOTAL_POINTS)

  #its not really a variance but kinda standard derviation ... lets just name it
  variance = 0.0
  for i in range(TOTAL_POINTS):
    variance+=np.sqrt((cv.mGet(points, i, 0) - mean_x)**2 + (cv.mGet(points, i,
      1) - mean_y)**2)
  variance/=float(TOTAL_POINTS)

  scale = np.sqrt(2)/variance
  translate_x = -scale*mean_x
  translate_y = -scale*mean_y
```

```
  cv.Zero(T)
  cv.mSet(T, 0, 0, scale)
  cv.mSet(T, 0, 2, translate_x)
  cv.mSet(T, 1, 2, translate_y)
  cv.mSet(T, 1, 1, scale)
  cv.mSet(T, 2, 2, 1)

  T_transpose = cv.CreateMat(3, 3, cv.CV_64FC1)
  cv.Transpose(T, T_transpose)
  #Note: points is a total_points x 3 matrix so the below way works
  cv.MatMul(points, T_transpose, points)

  return T

"""
Computes the fundamental Matrix from the user inputed point correspondences
"""
def computeF(p1, p2):
  global TOTAL_POINTS
  points1 = cv.CreateMat(TOTAL_POINTS, 3, cv.CV_64FC1)
  points2 = cv.CreateMat(TOTAL_POINTS, 3, cv.CV_64FC1)
  cv.Copy(p1, points1)
  cv.Copy(p2, points2)

  A = cv.CreateMat(TOTAL_POINTS, 9, cv.CV_64FC1)
  T1 = normalize(points1)
  T2 = normalize(points2)

  #Normalization matrices -> make the code work for images of different scales
  print "T1, T2 are"
  printMatrix(T1)
  printMatrix(T2)

  #Building the A matrix based on the textbook method
  for i in range(TOTAL_POINTS):
    cv.mSet(A, i, 0, cv.mGet(points2, i, 0)*cv.mGet(points1, i, 0))
    cv.mSet(A, i, 1, cv.mGet(points2, i, 0)*cv.mGet(points1, i, 1))
    cv.mSet(A, i, 2, cv.mGet(points2, i, 0))
    cv.mSet(A, i, 3, cv.mGet(points2, i, 1)*cv.mGet(points1, i, 0))
    cv.mSet(A, i, 4, cv.mGet(points2, i, 1)*cv.mGet(points1, i, 1))
    cv.mSet(A, i, 5, cv.mGet(points2, i, 1))
    cv.mSet(A, i, 6, cv.mGet(points1, i, 0))
    cv.mSet(A, i, 7, cv.mGet(points1, i, 1))
    cv.mSet(A, i, 8, 1)

  F = cv.CreateMat(3, 3, cv.CV_64FC1)
  D = cv.CreateMat(TOTAL_POINTS, 9, cv.CV_64FC1)
  V = cv.CreateMat(9, 9, cv.CV_64FC1)
  #set the flags in SVD to make it work faster
  cv.SVD(A, D, None, V, cv.CV_SVD_V_T)
```

```
 #last row of V-Transpose
  for i in range(3):
    for j in range(3):
      cv.mSet(F, i, j, cv.mGet(V, 8, i*3+j))

  #Reduce the Rank of the fundamental matrix
  U = cv.CreateMat(3, 3, cv.CV_64FC1)
  U_t = cv.CreateMat(3, 3, cv.CV_64FC1)
  D = cv.CreateMat(3, 3, cv.CV_64FC1)
  V = cv.CreateMat(3, 3, cv.CV_64FC1)
  cv.SVD(F, D, U, V, cv.CV_SVD_U_T|cv.CV_SVD_V_T)
  cv.Transpose(U, U_t)
  cv.mSet(D, 2, 2, 0)
  cv.MatMul(U_t, D, F)
  cv.MatMul(F, V, F)

  #Denormalizes
  T2_transpose = cv.CreateMat(3, 3, cv.CV_64FC1)
  cv.Transpose(T2, T2_transpose)
  cv.MatMul(T2_transpose, F, F)
  cv.MatMul(F, T1, F)

  #checks if fundamental matrix is correct - result should be nearly zero
  """
  print "Debug - check - F"
  point1 = cv.CreateMat(3, 1, cv.CV_64FC1)
  point2_transpose = cv.CreateMat(1, 3, cv.CV_64FC1)
  point2 = cv.CreateMat(3, 1, cv.CV_64FC1)

  for j in range(TOTAL_POINTS):
    result1 = cv.CreateMat(1, 3, cv.CV_64FC1)
    result2 = cv.CreateMat(1, 1, cv.CV_64FC1)
    for i in range(3):
      cv.mSet(point1, i, 0, cv.mGet(p1, j, i))
      cv.mSet(point2, i, 0, cv.mGet(p2, j, i))
    cv.Transpose(point2, point2_transpose)
    cv.MatMul(point2_transpose, F, result1)
    cv.MatMul(result1, point1, result2)
    printMatrix(result2)
  """
  print "Initial Fundamental Matrix:"
  printMatrix(F)
  return F

"""
Computes epipoles and camera Matrices
"""

def findEpipoles_CameraMatrix(F):
  U = cv.CreateMat(3, 3, cv.CV_64FC1)
  D = cv.CreateMat(3, 3, cv.CV_64FC1)
  V = cv.CreateMat(3, 3, cv.CV_64FC1)
```

```
  cv.SVD(F, D, U, V, cv.CV_SVD_U_T|cv.CV_SVD_V_T)

  #Epipole 1 and 2
  e1 = cv.CreateMat(3, 1, cv.CV_64FC1)
  e2 = cv.CreateMat(3, 1, cv.CV_64FC1)
  #right and left null vectors of F
  for i in range(3):
    cv.mSet(e1, i, 0, cv.mGet(V, 2, i)/cv.mGet(V, 2, 2))
    cv.mSet(e2, i, 0, cv.mGet(U, 2, i)/cv.mGet(U, 2, 2))

  #Camera matrices
  #Note: the P, Pp is the notation followed by textbook for these
  P1 = cv.CreateMat(3, 4, cv.CV_64FC1)
  P2 = cv.CreateMat(3, 4, cv.CV_64FC1)

  #epipolar vector in matrix representation
  Ex = cv.CreateMat(3, 3, cv.CV_64FC1)
  cv.Zero(Ex)
  cv.mSet(Ex, 0, 1, -cv.mGet(e2, 2, 0))
  cv.mSet(Ex, 0, 2, cv.mGet(e2, 1, 0))
  cv.mSet(Ex, 1, 0, cv.mGet(e2, 2, 0))
  cv.mSet(Ex, 1, 2, -cv.mGet(e2, 0, 0))
  cv.mSet(Ex, 2, 0, -cv.mGet(e2, 1, 0))
  cv.mSet(Ex, 2, 1, cv.mGet(e2, 0, 0))

  #Camera Matrix P1
  cv.Zero(P1)
  cv.mSet(P1, 0, 0, 1)
  cv.mSet(P1, 1, 1, 1)
  cv.mSet(P1, 2, 2, 1)

  #Camera Matrix P2
  Ex_F = cv.CreateMat(3, 3, cv.CV_64FC1)
  cv.MatMul(Ex, F, Ex_F)
  for i in range(3):
    for j in range(3):
      cv.mSet(P2, i, j, cv.mGet(Ex_F, i, j))
  for i in range(3):
    cv.mSet(P2, i, 3, cv.mGet(e2, i, 0))

  return (e1, e2, P1, P2)

"""
Computes the homography matrices H2 which sends e2 to [k, 0, 0]^T and H1 which
adjusts left image correspondingly
"""

def imageRectify(left_image, right_image, points1, points2, e1, e2, P1, P2, H1, H2, F):
  global TOTAL_POINTS

  #H2 = GRT which represent scaling, rotation and translation
  G = cv.CreateMat(3, 3, cv.CV_64FC1)
```

```
R = cv.CreateMat(3, 3, cv.CV_64FC1)
T = cv.CreateMat(3, 3, cv.CV_64FC1)
T2 = cv.CreateMat(3, 3, cv.CV_64FC1)

cv.Zero(G)
cv.Zero(R)
cv.Zero(T)
cv.Zero(H2)
image_width = left_image.width
image_height = left_image.height


angle = np.arctan(-(image_height/2 - cv.mGet(e2, 1, 0)/cv.mGet(e2, 2, 0))/(image_width/2
    - cv.mGet(e2, 0, 0)/cv.mGet(e2, 2, 0)))
f = np.cos(angle)*(-image_width/2 + cv.mGet(e2, 0, 0)/cv.mGet(e2, 2, 0)) -
    np.sin(angle)*(-image_height/2 + cv.mGet(e2, 1, 0)/cv.mGet(e2, 2,0))

#print "angle and f", angle, f
cv.mSet(R, 0, 0, np.cos(angle))
cv.mSet(R, 0, 1, -np.sin(angle))
cv.mSet(R, 1, 0, np.sin(angle))
cv.mSet(R, 1, 1, np.cos(angle))
cv.mSet(R, 2, 2, 1)
cv.mSet(G, 2, 0, -1/f)
cv.mSet(G, 0, 0, 1)
cv.mSet(G, 1, 1, 1)
cv.mSet(G, 2, 2, 1)
cv.mSet(T, 0, 0, 1)
cv.mSet(T, 1, 1, 1)
cv.mSet(T, 2, 2, 1)
cv.mSet(T, 0, 2, -image_width/2)
cv.mSet(T, 1, 2, -image_height/2)

"""
print "G and R"
printMatrix(G)
printMatrix(R)
printMatrix(T)
"""
cv.MatMul(G, R, H2)
cv.MatMul(H2, T, H2)

#check if H2 is correct -> result should be of the form [f, 0, 0]^T
"""
result = cv.CreateMat(3, 1, cv.CV_64FC1)
cv.MatMul(H2, e2, result)
print "The value is "
printMatrix(result)
"""

#preserves center after transformation with H2
centre_point = cv.CreateMat(3, 1, cv.CV_64FC1)
```

```
new_centre = cv.CreateMat(3, 1, cv.CV_64FC1)
cv.mSet(centre_point, 0, 0, image_width/2)
cv.mSet(centre_point, 1, 0, image_height/2)
cv.mSet(centre_point, 2, 0, 1)
cv.MatMul(H2, centre_point, new_centre)

cv.Zero(T2)
cv.mSet(T2, 0, 0, 1)
cv.mSet(T2, 1, 1, 1)
cv.mSet(T2, 2, 2, 1)
cv.mSet(T2, 0, 2, image_width/2 - cv.mGet(new_centre, 0, 0)/cv.mGet(new_centre, 2, 0))
cv.mSet(T2, 1, 2, image_height/2 - cv.mGet(new_centre, 1, 0)/cv.mGet(new_centre, 2, 0))
cv.MatMul(T2, H2, H2)


#computes H1
P_pseudo_inverse = cv.CreateMat(4, 3, cv.CV_64FC1)
cv.Invert(P1, P_pseudo_inverse, cv.CV_SVD)
M = cv.CreateMat(3, 3, cv.CV_64FC1)

H_temp1 = cv.CreateMat(3, 3, cv.CV_64FC1)
cv.MatMul(P2, P_pseudo_inverse, M)
cv.MatMul(H2, M, H_temp1)

A = cv.CreateMat(TOTAL_POINTS, 3, cv.CV_64FC1)
b = cv.CreateMat(TOTAL_POINTS, 1, cv.CV_64FC1)

x1 = cv.CreateMat(3, 1, cv.CV_64FC1)
new_x1 = cv.CreateMat(3, 1, cv.CV_64FC1)
x2 = cv.CreateMat(3, 1, cv.CV_64FC1)
new_x2 = cv.CreateMat(3, 1, cv.CV_64FC1)

for i in range(TOTAL_POINTS):
  cv.mSet(x1, 0, 0, cv.mGet(points1, i, 0))
  cv.mSet(x1, 1, 0, cv.mGet(points1, i, 1))
  cv.mSet(x1, 2, 0, 1)
  cv.MatMul(H_temp1, x1, new_x1)
  cv.Scale(new_x1, new_x1, 1/cv.mGet(new_x1, 2, 0))
  cv.mSet(x2, 0, 0, cv.mGet(points2, i, 0))
  cv.mSet(x2, 1, 0, cv.mGet(points2, i, 1))
  cv.mSet(x2, 2, 0, 1)
  cv.MatMul(H2, x2, new_x2)
  cv.Scale(new_x2, new_x2, 1/cv.mGet(new_x2, 2, 0))
  cv.mSet(A, i, 0, cv.mGet(new_x1, 0, 0))
  cv.mSet(A, i, 1, cv.mGet(new_x1, 1, 0))
  cv.mSet(A, i, 2, 1)
  cv.mSet(b, i, 0, cv.mGet(new_x2, 0, 0))

h_values = cv.CreateMat(3, 1, cv.CV_64FC1)
cv.Solve(A, b, h_values, cv.CV_LU)

H_temp2 = cv.CreateMat(3, 3, cv.CV_64FC1)
```

```
  cv.Zero(H_temp2)
  cv.mSet(H_temp2, 0, 0, cv.mGet(h_values, 0, 0))
  cv.mSet(H_temp2, 0, 1, cv.mGet(h_values, 1, 0))
  cv.mSet(H_temp2, 0, 2, cv.mGet(h_values, 2, 0))
  cv.mSet(H_temp2, 1, 1, 1)
  cv.mSet(H_temp2, 2, 2, 1)

  #H1 = H_temp2 x H_temp1
  cv.MatMul(H_temp2, H_temp1, H1)

  #preserves center after transformation with H1
  centre_point = cv.CreateMat(3, 1, cv.CV_64FC1)
  new_centre = cv.CreateMat(3, 1, cv.CV_64FC1)
  cv.mSet(centre_point, 0, 0, image_width/2)
  cv.mSet(centre_point, 1, 0, image_height/2)
  cv.mSet(centre_point, 2, 0, 1)
  cv.MatMul(H1, centre_point, new_centre)

  cv.Zero(T2)
  cv.mSet(T2, 0, 0, 1)
  cv.mSet(T2, 1, 1, 1)
  cv.mSet(T2, 2, 2, 1)
  cv.mSet(T2, 0, 2, image_width/2 - cv.mGet(new_centre, 0, 0)/cv.mGet(new_centre, 2, 0))
  cv.mSet(T2, 1, 2, image_height/2 - cv.mGet(new_centre, 1, 0)/cv.mGet(new_centre, 2, 0))
  cv.MatMul(T2, H1, H1)

  #updates Fundamental Matrix with respect to the recification homographies
  H1_inv = cv.CreateMat(3, 3, cv.CV_64FC1)
  H2_inv = cv.CreateMat(3, 3, cv.CV_64FC1)
  H2_inv_t = cv.CreateMat(3, 3, cv.CV_64FC1)
  cv.Invert(H1, H1_inv, cv.CV_SVD)
  cv.Invert(H2, H2_inv, cv.CV_SVD)
  cv.Transpose(H2_inv, H2_inv_t)
  cv.MatMul(H2_inv_t, F, F)
  cv.MatMul(F, H1_inv, F)

  #rectifies both images
  left_image = createNewImage(left_image, H1, H1_inv, 1)
  right_image = createNewImage(right_image, H2, H2_inv, 2)

  return (left_image, right_image)

"""
Creates a new image when the homography matrices are supplied
"""
def createNewImage(image, H, H_inv, index):

  im_result = cv.CreateImage((image.width, image.height), image.depth ,3)
  cv.Zero(im_result)
  #fill the image using intepolation
  image_coordinates1 = cv.CreateMat(3, 1, cv.CV_64FC1)
  image_coordinates2 = cv.CreateMat(3, 1, cv.CV_64FC1)
```

```
  cv.mSet(image_coordinates1, 2, 0, 1)

  for i in range(image.width):
    cv.mSet(image_coordinates1, 0, 0, i)
    for j in range(image.height):
      cv.mSet(image_coordinates1, 1, 0, j)
      cv.MatMul(H_inv, image_coordinates1, image_coordinates2)
      x = cv.mGet(image_coordinates2, 0, 0)/cv.mGet(image_coordinates2, 2, 0)
      y = cv.mGet(image_coordinates2, 1, 0)/cv.mGet(image_coordinates2, 2, 0)
      #Interpolation
      if x < image.width-1 and y < image.height-1 and x >= 0 and y>= 0:
        temp = [0,0,0]
        for k in range(3):
          temp[k] = 0
          temp[k] = temp[k]+(x-int(x))*(y-int(y))*(image[int(y+1),int(x+1)][k])
          temp[k] = temp[k]+(1.0-(x-int(x)))*(y-int(y))*(image[int(y+1),int(x)][k])
          temp[k] = temp[k]+(x-int(x))*(1.0-(y-int(y)))*(image[int(y),int(x+1)][k])
          temp[k] = temp[k]+(1.0-(x-int(x)))*(1.0-(y-int(y)))*(image[int(y),int(x)][k])
        im_result[j,i] = temp
      else :
        #fools the edge detector into not recognizing the back ground edges by
        #filling the background white
        im_result[j, i] = image[image.height - 20, image.width - 20]


  cv.SaveImage("result"+`index`+".png", im_result)
  return im_result

"""
Applies canny edge detection to detect image edges which are the interest
points in our case
"""
def getFeatures(image, features, index):
  image_edges = cv.CreateImage((image.width, image.height), image.depth, 1)
  aperture_size = 3
  threshold1 = 10
  threshold2 = 100
  #Applies canny Operator to get the image edges
  cv.Canny(image, image_edges, threshold1, threshold2, aperture_size)

  count = 0
  global NCC_WINDOW_SIZE
  for i in range(int(NCC_WINDOW_SIZE/2), image.width - int(NCC_WINDOW_SIZE/2 + 1)):
    for j in range(int(NCC_WINDOW_SIZE/2), image.height - int(NCC_WINDOW_SIZE/2 + 1)):
      if image_edges[j, i] != 0:
        features.append(Feature(i, j))
        count+=1
      else:
        image_edges[j, i] = 0
  print "Number of Features in image", index,":", count
  cv.SaveImage("edges"+`index`+".png", image_edges)
```

```
"""
FInds correspondences between interest points in two images using NCC and
epipolar constraint
"""
def findCorrespondences(features1, features2, F):

  print "Finding Correspondences"
  global NCC_THESHOLD
  for i in range(len(features1)):
    #epipolar constraint
    x1 = cv.CreateMat(3, 1, cv.CV_64FC1)
    x2 = cv.CreateMat(1, 3, cv.CV_64FC1)
    cv.mSet(x1, 0, 0, features1[i].x)
    cv.mSet(x1, 1, 0, features1[i].y)
    cv.mSet(x1, 2, 0, 1)
    constraint = cv.CreateMat(1, 1, cv.CV_64FC1)
    score = 0
    score_max = -1e10
    score_second_max = -1e10
    match_constraint = 1
    for j in range(len(features2)):
      cv.mSet(x2, 0, 0, features2[j].x)
      cv.mSet(x2, 0, 1, features2[j].y)
      cv.mSet(x2, 0, 2, 1)

      cv.MatMul(x2, F, x2)
      # x2.F.x1 = 0
      cv.MatMul(x2, x1, constraint)

      if (np.abs(features1[i].x - features2[j].x) > 30 or
          np.abs(features1[i].y - features2[j].y) > 40):
        continue

      #print cv.mGet(constraint, 0, 0)
      #maynot be equal to zero so lets put a threshold
      if (np.abs(cv.mGet(constraint, 0, 0)) > 0.055):
       continue

      score = nccScore(features1[i].neighbor, features2[j].neighbor)
      if score > score_max:
        score_max = score
        features1[i].score = score_max
        features1[i].match = j
        features2[j].score = score_max
        features2[j].match = i
        match_constraint = cv.mGet(constraint, 0, 0)

    for j in range(i):
      if features1[j].match == features1[i].match:
        if features1[j].score < features1[i].score:
          features1[j].match = -1
          features1[j].score = -1
```

```
        else:
          features1[i].match = -1
          features1[i].score = -1
        break

"""
Sorts a given set of interest points using Insertion sort
"""
def sortPoints(points1, points2, size):
  #insertion sort
  for j in range(1, size):
    key1 = (cv.mGet(points1, j, 0), cv.mGet(points1, j, 1), cv.mGet(points1, j, 2))
    key2 = (cv.mGet(points2, j, 0), cv.mGet(points2, j, 1), cv.mGet(points2, j, 2))
    i = j-1
    while i >= 0 and cv.mGet(points1, i, 0) > key1[0]:
      for k in range(3):
        cv.mSet(points1, i+1, k, cv.mGet(points1, i, k))
        cv.mSet(points2, i+1, k, cv.mGet(points2, i, k))
      i -= 1
    for k in range(3):
      cv.mSet(points1, i+1, k, key1[k])
      cv.mSet(points2, i+1, k, key2[k])

"""
Error Function for levmar
Essentially reprojects the world points to image space to find the error
"""

def errorFunc(parameters, measurement, data):

  final_values = []
  P1 = cv.CreateMat(3, 4, cv.CV_64FC1)
  P2 = cv.CreateMat(3, 4, cv.CV_64FC1)
  #Camera Matrix P1
  cv.Zero(P1)
  cv.mSet(P1, 0, 0, 1)
  cv.mSet(P1, 1, 1, 1)
  cv.mSet(P1, 2, 2, 1)

  #Unpacking the parameter vector
  for i in range(3):
    for j in range(4):
      cv.mSet(P2, i, j, parameters[i*4+j])

  number_of_points = (len(parameters) - 12)/3
  X = cv.CreateMat(number_of_points, 4, cv.CV_64FC1)

  for i in range(number_of_points):
    cv.mSet(X, i, 0, parameters[12 + 3*i])
    cv.mSet(X, i, 1, parameters[13 + 3*i])
    cv.mSet(X, i, 2, parameters[14 + 3*i])
    cv.mSet(X, i, 3, 1)
```

```
  P1_transpose = cv.CreateMat(4, 3, cv.CV_64FC1)
  P2_transpose = cv.CreateMat(4, 3, cv.CV_64FC1)
  cv.Transpose(P1, P1_transpose)
  cv.Transpose(P2, P2_transpose)

  x1 = cv.CreateMat(number_of_points, 3, cv.CV_64FC1)
  x2 = cv.CreateMat(number_of_points, 3, cv.CV_64FC1)

  cv.MatMul(X, P1_transpose, x1)
  cv.MatMul(X, P2_transpose, x2)

  for i in range(number_of_points):
    final_values.append(cv.mGet(x1, i, 0)/cv.mGet(x1, i, 2))
    final_values.append(cv.mGet(x1, i, 1)/cv.mGet(x1, i, 2))
    final_values.append(cv.mGet(x2, i, 0)/cv.mGet(x2, i, 2))
    final_values.append(cv.mGet(x2, i, 1)/cv.mGet(x2, i, 2))

  return final_values

"""
Optimizes the parameters using Levmar which is a levenberg marquardt
implementation
"""
def LMOptimize(points1, points2, F, P1, P2, world_points, e1, e2,
    number_of_points):

  #packing all parmeters into one huge parameter vector
  parameters = np.zeros(12+3*number_of_points, float)
  data = []
  for i in range(3):
    for j in range(4):
      parameters[i*4+j] = cv.mGet(P2, i, j)
  for i in range(number_of_points):
    parameters[12+i*3] = cv.mGet(world_points, i, 0)
    parameters[13+i*3] = cv.mGet(world_points, i, 1)
    parameters[14+i*3] = cv.mGet(world_points, i, 2)

  count = 0
  for i in range(number_of_points):
    data.append(cv.mGet(points1, i, 0))
    data.append(cv.mGet(points1, i, 1))
    data.append(cv.mGet(points2, i, 0))
    data.append(cv.mGet(points2, i, 1))
    count+=4

  #finds the initial initial Error
  values = errorFunc(parameters, data, "dummy")
  error = 0
  for i in range(len(values)):
    error += (data[i] - values[i])**2
  print "Initial Error before LM is :", error
```

```
    """
    if there are too many parameters better to have less iterations or levmar
    will crash
    """
    iterations = 1000
    if number_of_points > 20:
      iterations = 6

    # note that data is dummy variable because everything is packed in
    # 'parameters'
    result = levmar.ddif(errorFunc, parameters, data, iterations, data = "dummy")

    #print result

    #Unpack the results obtained to use in the futher steps
    for i in range(3):
      for j in range(4):
        cv.mSet(P2, i, j, result[0][4*i+j])

    for i in range(number_of_points):
      cv.mSet(world_points, i, 0, result[0][12+i*3])
      cv.mSet(world_points, i, 1, result[0][13+i*3])
      cv.mSet(world_points, i, 2, result[0][14+i*3])

    print "Number Of Actual Iterations done by levmar ", result[1]

    for i in range(3):
      cv.mSet(e2, i, 0, cv.mGet(P2, i, 3))

    #epipolar vector in matrix representation
    Ex = cv.CreateMat(3, 3, cv.CV_64FC1)
    cv.Zero(Ex)
    cv.mSet(Ex, 0, 1, -cv.mGet(e2, 2, 0))
    cv.mSet(Ex, 0, 2, cv.mGet(e2, 1, 0))
    cv.mSet(Ex, 1, 0, cv.mGet(e2, 2, 0))
    cv.mSet(Ex, 1, 2, -cv.mGet(e2, 0, 0))
    cv.mSet(Ex, 2, 0, -cv.mGet(e2, 1, 0))
    cv.mSet(Ex, 2, 1, cv.mGet(e2, 0, 0))

    #Recomputes F from the new parameters
    M = cv.CreateMat(3, 3, cv.CV_64FC1)
    for i in range(3):
      for j in range(3):
        cv.mSet(M, i, j, cv.mGet(P2, i, j))
    cv.MatMul(Ex, M, F)
    cv.Scale(F, F, 1/cv.mGet(F, 2, 2))

"""
Finds the 3D points using the P1, P2 camera matrices
"""
def find3DPoints(features1, features2, P1, P2, number_of_points):
```

```
  A = cv.CreateMat(4, 4, cv.CV_64FC1)
  D = cv.CreateMat(4, 4, cv.CV_64FC1)
  V = cv.CreateMat(4, 4, cv.CV_64FC1)
  world_points = cv.CreateMat(number_of_points, 3, cv.CV_64FC1)


  for i in range(number_of_points):
    for j in range(4):
      cv.mSet(A, 0, j, cv.mGet(features1, i, 0) * cv.mGet(P1, 2, j) - cv.mGet(P1, 0, j))
      cv.mSet(A, 1, j, cv.mGet(features1, i, 1) * cv.mGet(P1, 2, j) - cv.mGet(P1, 1, j))
      cv.mSet(A, 2, j, cv.mGet(features2, i, 0) * cv.mGet(P2, 2, j) - cv.mGet(P2, 0, j))
      cv.mSet(A, 3, j, cv.mGet(features2, i, 1) * cv.mGet(P2, 2, j) - cv.mGet(P2, 1, j))
    cv.SVD(A, D, None, V)
    cv.mSet(world_points, i, 0, cv.mGet(V, 0, 3)/cv.mGet(V, 3, 3))
    cv.mSet(world_points, i, 1, cv.mGet(V, 1, 3)/cv.mGet(V, 3, 3))
    cv.mSet(world_points, i, 2, cv.mGet(V, 2, 3)/cv.mGet(V, 3, 3))

  return world_points

"""
Draws the 3D points as a scatter plot with matplotlib
"""
def draw3D(world_points, size):
  fig = plt.figure()
  ax = fig.add_subplot(111, projection='3d')
  x = []
  y = []
  z = []

  count = 0
  for i in range(size):
    if np.abs(cv.mGet(world_points, i, 2)) < 20:# get rid of outliers!
      x.append((cv.mGet(world_points, i, 0)))
      y.append((cv.mGet(world_points, i, 1)))
      z.append((cv.mGet(world_points, i, 2)))
      #print x[count],",", y[count],",", z[count]
      count+=1

  ax.scatter(x, y, z, zdir='z', s=1)
  ax.set_xlabel('X Label')
  ax.set_ylabel('Y Label')
  ax.set_zlabel('Z Label')

  plt.show()

"""
main function
"""
def main():
  global left_image
  global right_image
  global pointCount
```

```
#loads image
if len(sys.argv) > 2:
  filename1 = sys.argv[1]
  filename2 = sys.argv[2]
  left_image = cv.LoadImage(filename1, cv.CV_LOAD_IMAGE_UNCHANGED)
  if left_image == 0:
    print "enter a valid Image Path"
  right_image = cv.LoadImage(filename2, cv.CV_LOAD_IMAGE_UNCHANGED)
  if right_image == 0:
    print "enter a valid Image Path"
else:
  print "Enter image file path as argument"

temp_image1 = cv.CloneImage(left_image)
temp_image2 = cv.CloneImage(right_image)

global TOTAL_POINTS
#initializes input variables and takes input from user
points1 = cv.CreateMat(TOTAL_POINTS, 3, cv.CV_64FC1)
points2 = cv.CreateMat(TOTAL_POINTS, 3, cv.CV_64FC1)
points_values1 = (( 162/2, 162/2, 1,), ( 366/2, 65/2,   1,), ( 44/2,
  77/2, 1,), ( 298/2, 198/2,   1,), ( 393/2, 109/2,   1,), ( 51/2,
    132/2,   1,) , ( 325/2, 106/2,   1,), ( 391/2, 67/2,   1,), ( 205/2,
      285/2,   1,), ( 366/2, 162/2,   1,), ( 118/2, 157/2,   1,), (
        187/2, 217/2,   1,))
points_values2 = ((54, 77, 1), (178, 31, 1), (10, 31, 1), (136, 106, 1),
    (192, 57, 1), (15, 61, 1), (153, 51, 1), (199, 33, 1), (73, 146, 1),
    (180, 85, 1), (35, 75, 1), (61, 107, 1))

for i in range(TOTAL_POINTS):
  for j in range(3):
    cv.mSet(points1, i, j, points_values1[i][j])
    cv.mSet(points2, i, j, points_values2[i][j])


#Input points from the user or use the hard coded values
params1 = (left_image, points1)
params2 = (right_image, points2)
cv.NamedWindow('Input Image', cv.CV_WINDOW_AUTOSIZE)
cv.SetMouseCallback('Input Image', on_mouse, params1)
cv.ShowImage('Input Image', left_image)
cv.WaitKey()
cv.DestroyWindow('Input Image')
pointCount = 0
cv.NamedWindow('Input Image', cv.CV_WINDOW_AUTOSIZE)
cv.SetMouseCallback('Input Image', on_mouse, params2)
cv.ShowImage('Input Image', right_image)
cv.WaitKey()
cv.DestroyWindow('Input Image')


for i in range(TOTAL_POINTS):
```

```
  cv.Circle(left_image, (int(cv.mGet(points1, i, 0)), int(cv.mGet(points1, i,
    1))), 0, (255, 255, 0), 2)
  cv.Circle(right_image, (int(cv.mGet(points2, i, 0)), int(cv.mGet(points2,
    i, 1))), 0, (255, 255, 0), 2)

cv.SaveImage("image1_selected_p.png", left_image)
cv.SaveImage("image2_selected_p.png", right_image)

#computes the fundamental Matrix
printMatrix(points1)
printMatrix(points2)
F = computeF(points1, points2)
#epipoles and camera matrices
(e1, e2, P1, P2) = findEpipoles_CameraMatrix(F)


left_image = temp_image1
right_image = temp_image2

#perform LM to update the parameters such as P1, P2, F
world_points = find3DPoints(points1, points2, P1, P2, TOTAL_POINTS)
LMOptimize(points1, points2, F, P1, P2, world_points, e1, e2, TOTAL_POINTS)

#rectification homographies
H1 = cv.CreateMat(3, 3, cv.CV_64FC1)
H2 = cv.CreateMat(3, 3, cv.CV_64FC1)

#rectifies images
(left_image, right_image) = imageRectify(left_image, right_image, points1, points2, e1,
    e2, P1, P2, H1, H2, F)

"""
#check rectification
result = cv.CreateMat(3, 1, cv.CV_64FC1)
cv.MatMul(H2, e2, result)
print "Transformed e2"
printMatrix(result)
"""

#edge detection
gray_scale_image1 = cv.CreateImage((left_image.width, left_image.height),
    left_image.depth, 1)
gray_scale_image2 = cv.CreateImage((right_image.width, right_image.height),
    right_image.depth, 1)
cv.CvtColor(left_image, gray_scale_image1, cv.CV_RGB2GRAY)
cv.CvtColor(right_image, gray_scale_image2, cv.CV_RGB2GRAY)

features1 = []
features2 = []

#finds the interest points -> pixels of edges!
getFeatures(gray_scale_image1, features1, 1)
```

```
getFeatures(gray_scale_image2, features2, 2)

global NCC_WINDOW_SIZE, TOTAL_FEATURES

TOTAL_FEATURES = len(features2)
if len(features1) > len(features2):
  TOTAL_FEATURES = len(features1)

#finds neighbors of each feature for ncc score
for i in range(TOTAL_FEATURES):
  f1 = cv.CreateMat(NCC_WINDOW_SIZE, NCC_WINDOW_SIZE, cv.CV_64FC1)
  f2 = cv.CreateMat(NCC_WINDOW_SIZE, NCC_WINDOW_SIZE, cv.CV_64FC1)

  if i < len(features1):
    x1 = features1[i].x
    y1 = features1[i].y
    getNeighbor(x1, y1, gray_scale_image1, f1)
    features1[i].neighbor = f1

  if i < len(features2):
    x2 = features2[i].x
    y2 = features2[i].y
    getNeighbor(x2, y2, gray_scale_image2, f2)
    features2[i].neighbor = f2

#computes the correspondences!
findCorrespondences(features1, features2, F)

#show correspondences
#reconstruct the final image
final_image = cv.CreateImage((2*left_image.width, left_image.height), left_image.depth,
    3)

cv.SetImageROI(final_image, (0, 0, left_image.width, left_image.height))
cv.Copy(left_image, final_image)
cv.ResetImageROI(final_image)

cv.SetImageROI(final_image, (right_image.width, 0, right_image.width,
    right_image.height))
cv.Copy(right_image, final_image)
cv.ResetImageROI(final_image)

count = 0

for i in range(len(features1)):
  if features1[i].match != -1:
    count = count + 1
    #different color for each point to differentiate between them.
    cv.Circle(final_image, (features1[i].x, features1[i].y), 0,
        (255*(count%4), 255*((count+1)%4), 255*((count+2)%4)), 3)
    cv.Circle(final_image, (features2[features1[i].match].x +
      left_image.width, features2[features1[i].match].y), 0, (255*(count%3),
```

```
        255*((count+1)%3), 255*((count+2)%3)), 3)

    #show some correspondence pair to ensure they are correct
    #if count % 40 == 0:
      #cv.Line(final_image, (features1[i].x, features1[i].y),
          (features2[features1[i].match].x + left_image.width,
          features2[features1[i].match].y), (255*(count%4), 255*((count+1)%4),
          255*((count+2)%4)), 1, cv.CV_AA, 0)


correspondence_points1 = cv.CreateMat(count, 2, cv.CV_64FC1)
correspondence_points2 = cv.CreateMat(count, 2, cv.CV_64FC1)

count = 0

#unrectifies the points to get the 3D values!!!
H1_inv = cv.CreateMat(3, 3, cv.CV_64FC1)
H2_inv = cv.CreateMat(3, 3, cv.CV_64FC1)
cv.Invert(H1, H1_inv)
cv.Invert(H2, H2_inv)
point1 = cv.CreateMat(3, 1, cv.CV_64FC1)
cv.mSet(point1, 2, 0, 1)
point2 = cv.CreateMat(3, 1, cv.CV_64FC1)
for i in range(len(features1)):
  if features1[i].match != -1:
    cv.mSet(point1, 0, 0, features1[i].x)
    cv.mSet(point1, 1, 0, features1[i].y)
    cv.MatMul(H1_inv, point1, point2)
    cv.mSet(correspondence_points1, count, 0, cv.mGet(point2, 0, 0)/cv.mGet(point2, 2,
        0))
    cv.mSet(correspondence_points1, count, 1, cv.mGet(point2, 1, 0)/cv.mGet(point2, 2,
        0))
    cv.mSet(point1, 0, 0, features2[features1[i].match].x)
    cv.mSet(point1, 1, 0, features2[features1[i].match].y)
    cv.MatMul(H2_inv, point1, point2)
    cv.mSet(correspondence_points2, count, 0, cv.mGet(point2, 0, 0)/cv.mGet(point2, 2,
        0))
    cv.mSet(correspondence_points2, count, 1, cv.mGet(point2, 1, 0)/cv.mGet(point2, 2,
        0))
    count+=1


print "Number of Correspondences = ", count
cv.SaveImage("Result.png", final_image)#result stored in Result.png
sortPoints(correspondence_points1, correspondence_points2, count)
world_points = find3DPoints(correspondence_points1, correspondence_points2, P1, P2,
    count)
#optimizes the values
LMOptimize(correspondence_points1, correspondence_points2, F, P1, P2,
    world_points, e1, e2, count)

#plots the world_points
```

```
  draw3D(world_points, count)

if __name__=="__main__":
  main()
```