Homework - 5

Sriram Karthik Badam

October 18, 2012

1. **Image Mosaicking using RANSAC**

   Before we apply the RANSAC Algorithm, interest points in each image have to be calculated using SIFT /SURF or Harris corner detector. In this Homework, SURF has been used to find the interest points in each image and a distance metric is used on the feature descriptors obtained from SURF, to get the correspondences. These correspondences are not always true but the RANSAC Algorithm takes care of the mismatched interest points.

   (a) Speeded Up Robust Feature(SURF)

   In contrast to others, SURF defines an interest point as the pixel where the determinant of Hessian(H) is maximized at some scale(σ value). SURF works over a pyramid(set) of images gaussian smoothed with different scale(σ) values.

$$H(x, y, \sigma) = \begin{bmatrix} \frac{\partial^2 ff(x,y,\sigma)}{\partial x^2} & \frac{\partial^2 ff(x,y,\sigma)}{\partial x \partial y} \\ \frac{\partial^2 ff(x,y,\sigma)}{\partial x \partial y} & \frac{\partial^2 ff(x,y,\sigma)}{\partial y^2} \end{bmatrix}$$

   where $ff(x, y, \sigma)$ is the gaussian smoothed version of the image. The values of the elements in the Hessian matrix are computed by using discrete approximations of the double derivatives.

   Note:

   i. Inorder to make calculations faster SURF uses Integral images.

$$I(x, y) = \sum_{i=0}^{x} \sum_{j=0}^{x} f(i, j)$$

   where I is the Integral image and f(i,j) is the pixel value of i,j in original image.

   ii. When the double derivatives in the Hessian Matrix are to be computed for every pixel at every scale(σ), the use of Integral image eases up the computation because the number of pixel look ups needed are less.

   The feature descriptor for SURF interest points is a simple 64x1 vector. Once the Feature descriptor is calculated at each interest point we can find the correspondences by computing the Euclidean distance between the feature descriptors (exhaustive search). The interest points with low distance value are considered to be similiar.

   (b) **RANSAC Algorithm**

   The Homography (H) that maps a point on the image1 to image2 is given by (image1, image2 are two input images)

$$x^{image2} = H.x^{image1}$$

where

$$x^{image1} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \& x^{image2} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

Then

$$\begin{bmatrix} x & y & 1 & 0 & 0 & 0 & -xx' & -yx' \\ 0 & 0 & 0 & x & y & 1 & -xy' & -yy' \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

Observe that this is a system of linear equations with 8 variables and theoritically this equality holds for every true correspondence. To compute the solution for this system we need atleast 4 points.

The RANSAC Algorithm works as follows:

i. Typically we have more than four correspondences. So we choose more than four random correspondences (8 correspondences have been used in my implementation) and compute the matrix on the left hand side(Matrix A) and right hand side (Matrix B). Its clear that this system is now an overdetermined system of linear equations of the form:

$$A.\vec{h} = B$$

ii. To find $\vec{h}$, we use the Linear Least Squares method. By multiplying the pseudo inverse of A with B, we get the solution($\vec{h}$). Mind that this is an approximation and an error is only minimized.

$$\vec{h} = (A^T A)^{-1} A^T B$$

iii. The Homography H obtained above, is applied to every correspondence to get the set of inliers and outliers. Inliers are the set of points, which when transformed using H, are within a δ (Inlier_distance_threshold) distance from their correspondence. The value ofδ is set manually.

iv. Using the set of inliers obtained above the number of trials(N) needed to compute the optimal H is calculated. The optimal estimate of H is the one which gives the largest inlier set and has least error variance value.

$$\epsilon = 1 - \frac{number\_of\_inliers}{number\_of\_correspondences}$$

$$N = \frac{log(1-0.99)}{log(1-(1-\epsilon)^n)}; (n = 8)$$

v. The estimate of H obtained from the above step can be refined further using the Levenberg-Marquardt functions or Dogleg Method. The Levenberg-Macquardt method finds the extrema(minima) of an nonlinear objective function over a space of parameters.

Image name	H_{Th}	$SURF_{score}$	$SURF_{ratio}$	$\delta(Inlier)$
car	2000	0.2	0.9	20
home1	400	0.2	0.9	30
EE	1000	0.2	0.9	20
Alcatraz	2000	0.15	0.9	20

Table 1: Shows the parameter values in SURF and RANSAC

Its step size is based on both Gradient Descent and Gauss Newton Methods. By passing the estimate of H from the above step to the LM function we can get the best value of H that minimizes the least square error.

$$Error = \sum_{i=1}^{m} (B_i - A_i\vec{h})^2$$

A_i is the i^{th} row of matrix A. Once you get the Homographies for every pair of images, the next step is to use them to stitch the images together.

(c) **Image Mosaicking**:

i. From the Homographies for every pair of successive images obtained above, the cummulative homographies, to project onto the centre image, are calculated.
For Example:
$$H_{left,centre} = H_{left,i}H_{i,i-1}......H_{i-m,centre}$$
where i stands for images between left image and entre

ii. The boundaries of each image in the centre image plane are computed with help of the cummulative homographies. Note that the origin for the centre image is at its left corner and therefore we get the boundaries with respect to that origin.

iii. Using the boundaries we create a huge canvas for the final image. Now the values of each pixel in the final mosaicked image can be calculated by finding the image the pixel belongs to. Note that origin has to be shifted to the coordinates of the left corner of central image when we try to figure out the value of the pixel.

iv. Using the inverse of the homography, we can compute the actual value of the pixel in its parent image.

(d) **Parameters**

Here is a small description of the parameters. Refer Table 1 for values

i. H_{Th} = Threshold on Hessian value of feature point in SURF

ii. $SURF_{score}$ = Threshold on the SURF distance value between two feature points

iii. $SURF_{ratio}$ = Threshold on the ratio of minimum score and second minimum sore for SURF feature correspondence

iv. δ = Inlier distance threshold to determine whether a correspondence is an inlier or not.

v. M = denotes the minumum number of inliers required to treat a H value as acceptable. This has been set to 0.8 times the correspondences.

2. **Observations**:

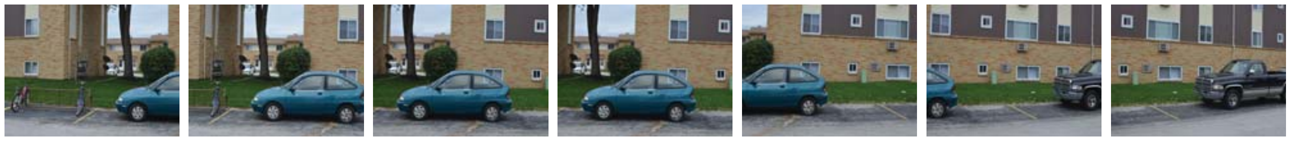
   (a) The presence of 3D features in the image leads to a really bad performance of the Homography Estimation. This can clearly be observed in the input images.

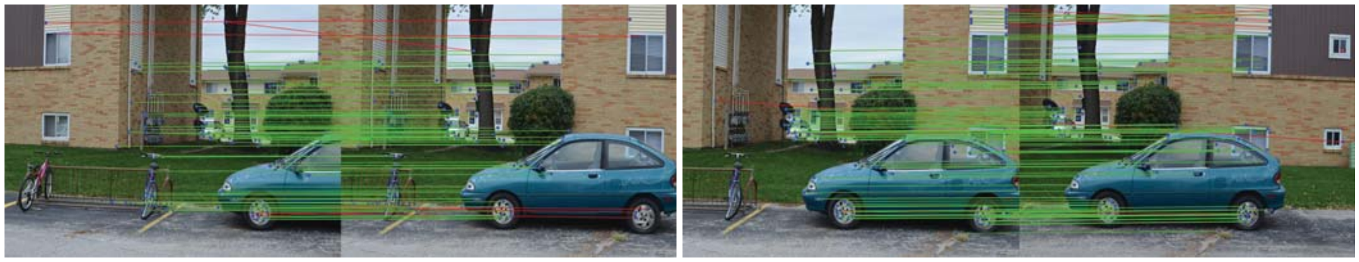
   (b) The LM algorithm doesn't improve the result much in any of my input images. (Sometimes it makes the result worse and its documentation is really sparse to find any bugs in my code)

   (c) Since I used SURF, illumination variations didnt effect the final mosaic.

3. **Results:**

   Input: Car
   Lines in green connect the inliers and the lines in Red connect the outliers. Zoom in for a better view or you can also find the image in zip file attached.





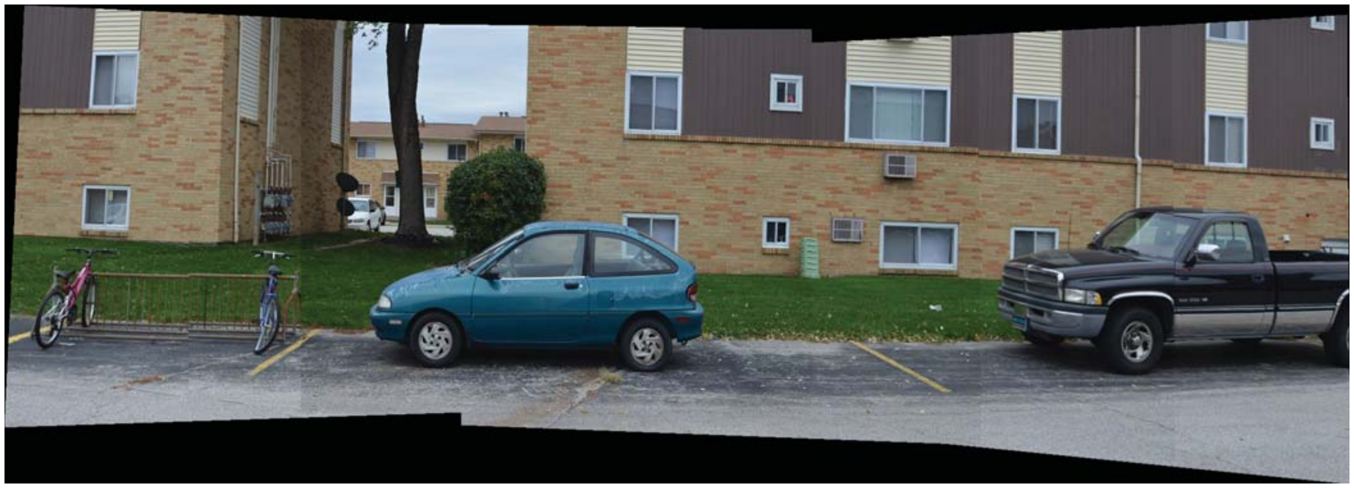
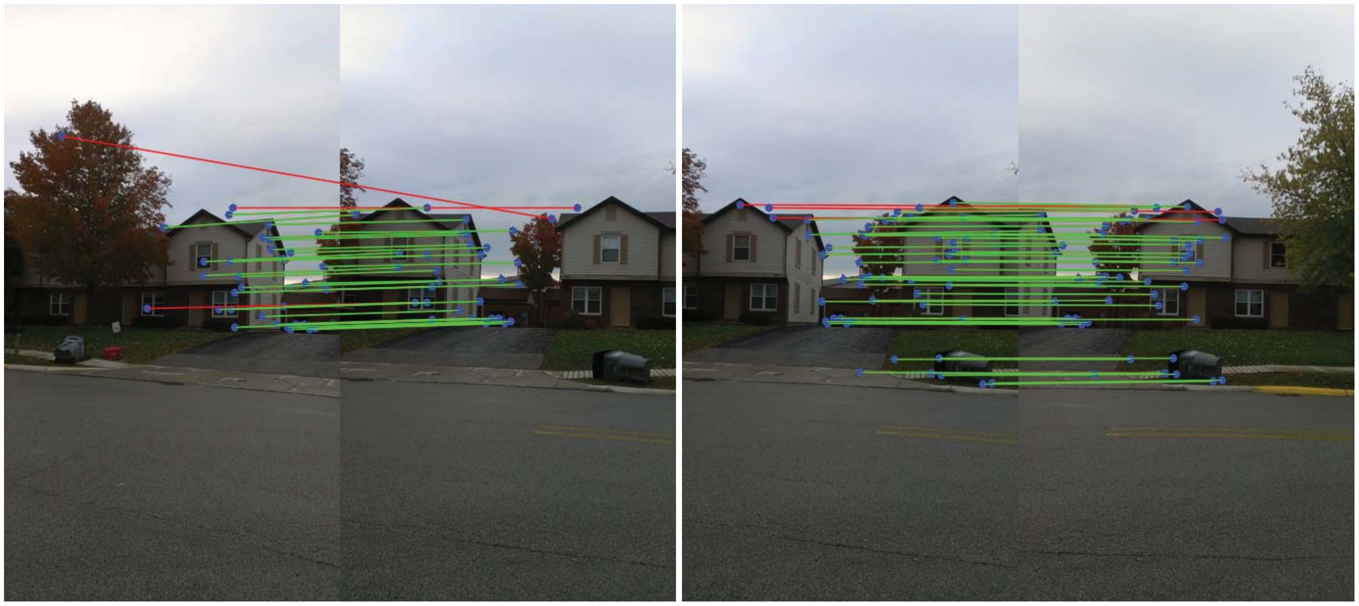


Figure 1: ”After Image mosaicking”

(The above set of input pictures were actually taken by C.J.R.Bharath and I thank him for letting me use them in my homework)

Input: Home1



Observe that the final result is acceptable even though there are illumination variations in the image.

Mosaicking Results in next page....

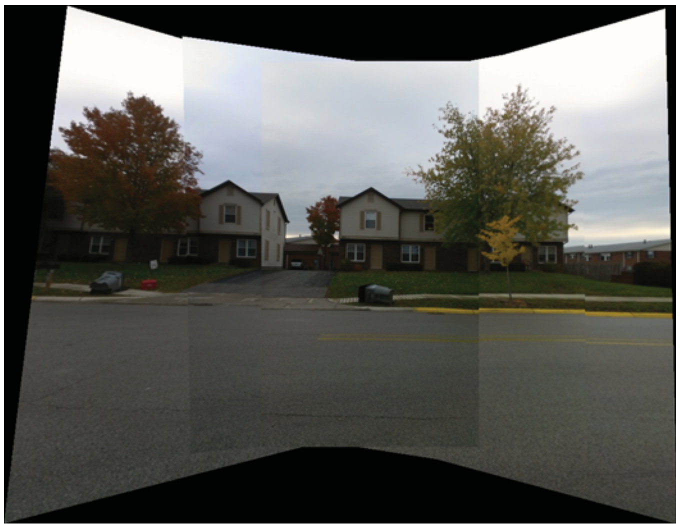


Figure 2: Results of mosaicking 5 images in input2

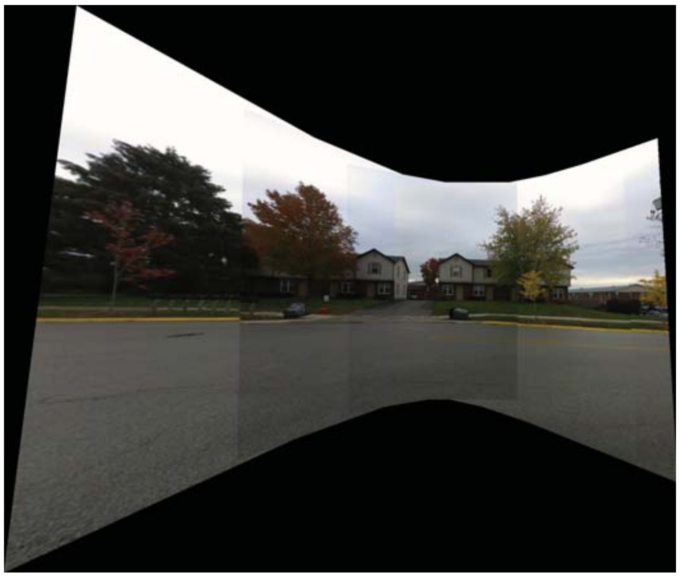
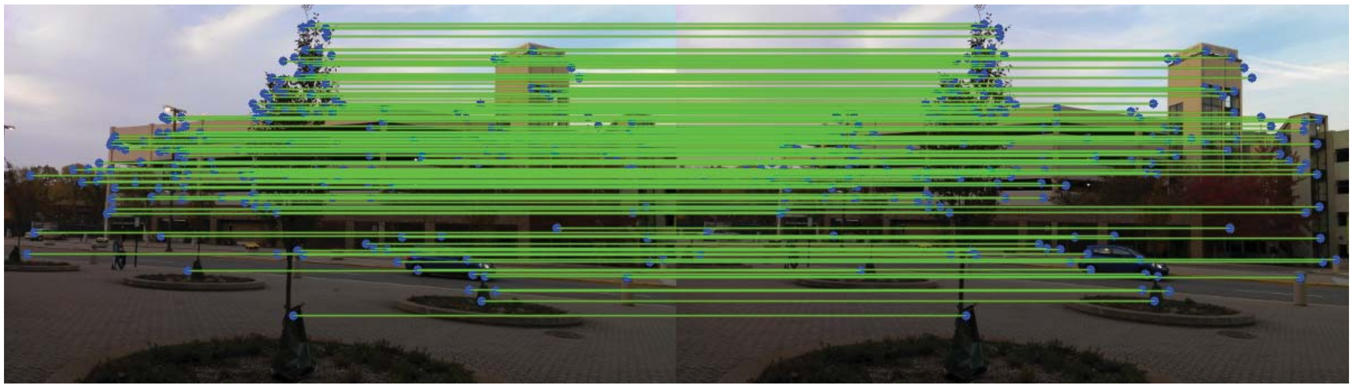
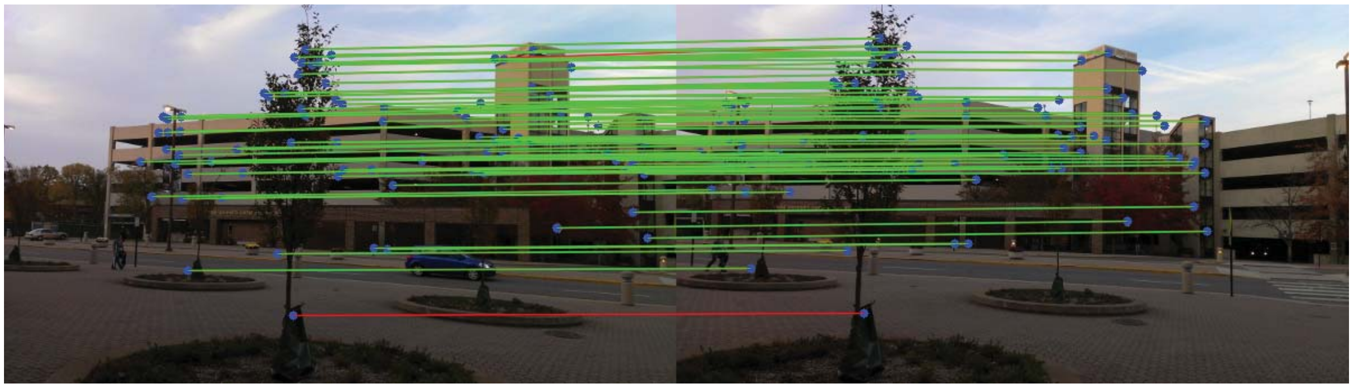


Figure 3: Results of mosaicking all 7 images in input2

Input: Opposite Electrical Building





Results : next Page....
Observe that bicycles become a dominant part in the right images, causing an expansion of the background.

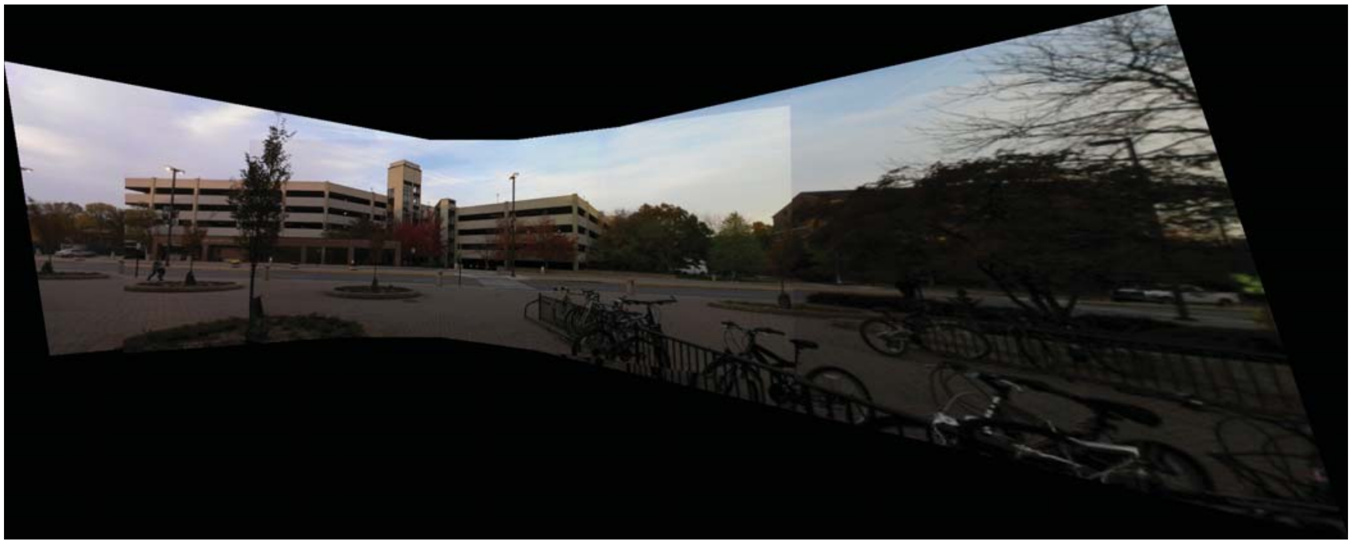
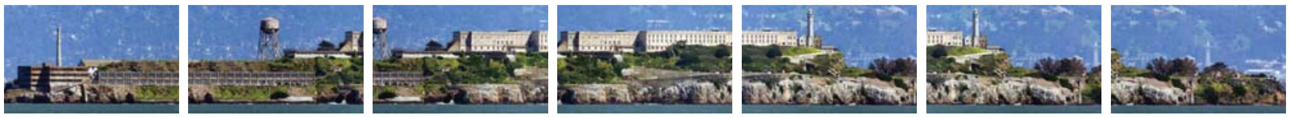
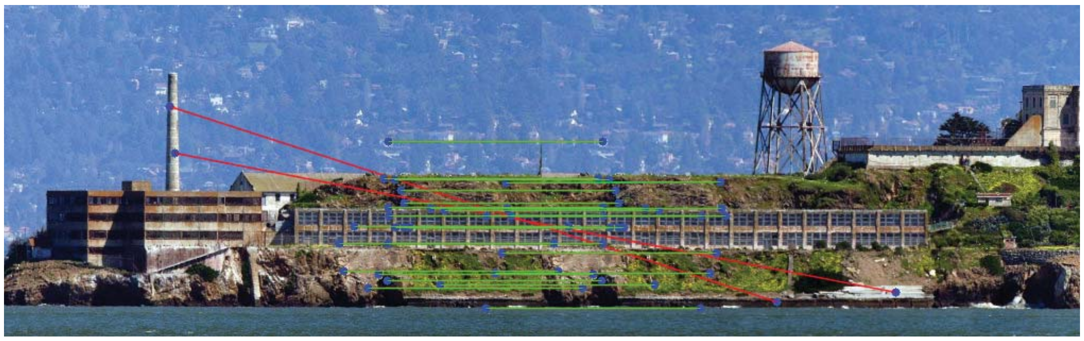


Figure 4: Results of mosaicking

Input: Alcatraz





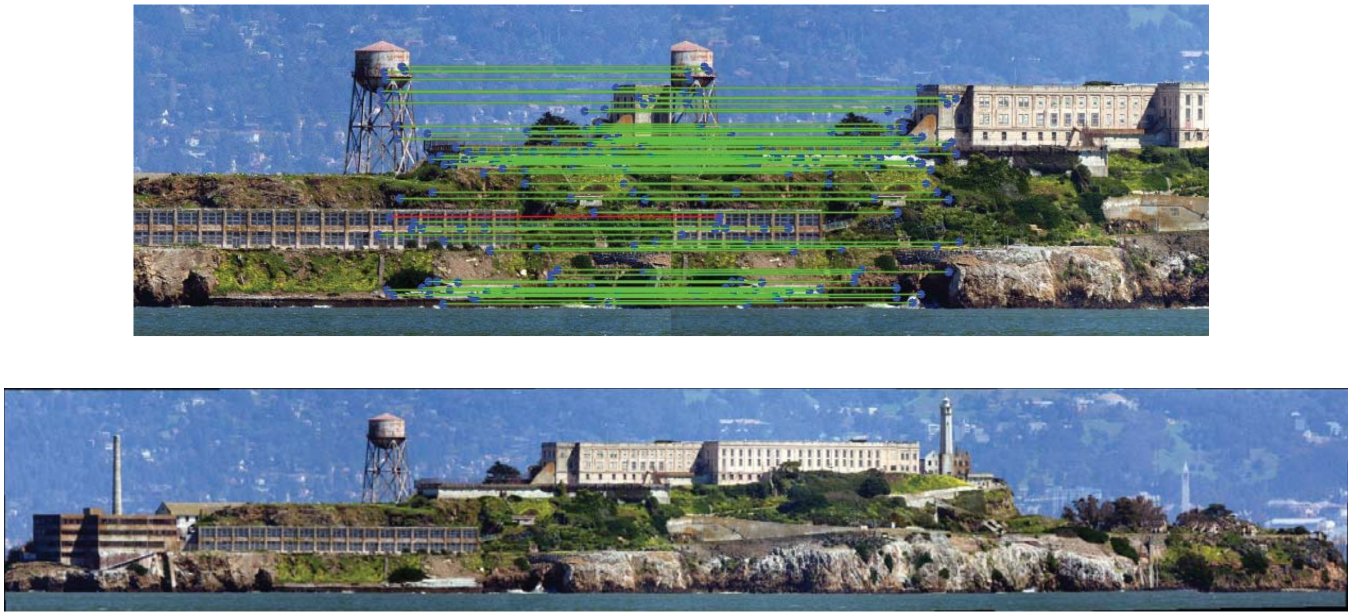


Figure 5: Results of mosaicking

(This set of input pictures were obtained from the internet. The pictures only require translation for reconstructing the final mosiac. I wish I had found images of Alcatraz with more distortion.)

4. **Source Code: ransac.py**

```
#!/usr/bin/python
#
# Author: Sriram Karthik Badam
# Date: Oct 13, 2012
#

import sys, os
import cv, lmfit
import numpy as np

#taken as input from user
NUMBER_OF_IMAGES = 0

#input for lmfit function
A_matrix = 0
b_values = 0

PERFORM_LM = 0

#the threshold M
NUMBER_OF_INLIERS_THRESHOLD = 0.8


#Thresholds for home1
"""
SCORE_THRESHOLD = 0.2
RATIO_THRESHOLD = 0.9
SURF_THRESHOLD = 400
INLIER_DISTANCE_THRESHOLD = 30
"""
#Thresholds for car

SCORE_THRESHOLD = 0.2
RATIO_THRESHOLD = 0.9
SURF_THRESHOLD = 2000
INLIER_DISTANCE_THRESHOLD = 20

"""
#Thresholds for EE
SCORE_THRESHOLD = 0.2
RATIO_THRESHOLD = 0.9
SURF_THRESHOLD = 1000
INLIER_DISTANCE_THRESHOLD = 20
"""

#Thresholds for Alcatraz
"""
SCORE_THRESHOLD = 0.15
RATIO_THRESHOLD = 0.9
SURF_THRESHOLD = 2000
INLIER_DISTANCE_THRESHOLD = 20
"""

"""
Computes the feature points and feature descriptors using the Opencv ExtractSURF
    method.
```

```
"""
def SURFdetector(image):
  global SURF_THRESHOLD #Threshold on the Hessian Value to prune the interest points
  (feature_points, feature_descriptors) = cv.ExtractSURF(image, None, cv.
     CreateMemStorage(), (0, SURF_THRESHOLD, 3, 1))
  return (feature_points, feature_descriptors)

"""
Computes the distance between two feature descriptors using Euclidean Distance measure
"""
def distance(p1, p2):
  p1_sub_p2 = np.subtract(p1, p2)
  p1_p2_square = np.power(p1_sub_p2, 2)
  score = np.sum(p1_p2_square)
  return np.power(score, 0.5)

"""
Error function for LMfit
"""
def error_func(p):
  global A_matrix, b_values
  h = np.zeros((8,1), float)
  h[0][0] = p['h0'].value
  h[1][0] = p['h1'].value
  h[2][0] = p['h2'].value
  h[3][0] = p['h3'].value
  h[4][0] = p['h4'].value
  h[5][0] = p['h5'].value
  h[6][0] = p['h6'].value
  h[7][0] = p['h7'].value
  C = np.dot(A_matrix, h)
  return_value = np.zeros(len(b_values), float)
  for i in range(len(b_values)):
    return_value[i] = b_values[i][0] - C[i][0]
  return return_value

"""
Finds the Correspondences between interest points in two images
"""

def correspondence_finder(feature_points1, feature_points2, feature_descriptors1,
    feature_descriptors2, correspondence):
  global SCORE_THRESHOLD, RATIO_THRESHOLD
  #Calculates the correspondences
  for i in range(len(feature_points1)):
    score = 0
    score_min = 1e10
    score_second_min = 1e10
    for j in range(len(feature_points2)):
      score =  distance(feature_descriptors1[i], feature_descriptors2[j])
      #updates the min, second_min
      if score_min > score and score < SCORE_THRESHOLD:
  score_second_min = score_min
  score_min = score
  cv.mSet(correspondence, i, 0, j)
      #updates second_min
      elif score_min < score and score > score_second_min:
```

```
  score_second_min = score

    if score_second_min >0 and score_min/score_second_min > RATIO_THRESHOLD:
  cv.mSet(correspondence, i, 0, -1)

"""
Applies RANSAC Algorithm on the input to find the best homography
"""

def RANSAC(feature_points1, feature_points2, correspondence, H):

  #feature_point is a huge structure so points1, points2 are used to store the
  #coordinates of interest points
  points1 = np.zeros((len(feature_points1), 2), float)
  points2 = np.zeros((len(feature_points2), 2), float)

  count = 0
  number_of_correspondences = 0

  #Gets Input ready
  for ((x,y), laplacian, size, dir, hessian) in feature_points1:
    points1[count][0] = x
    points1[count][1] = y
    if(cv.mGet(correspondence, count, 0) != -1):
      number_of_correspondences = number_of_correspondences + 1
    count+=1

  print "number of correspondences :", number_of_correspondences
  count = 0
  for ((x,y), laplacian, size, dir, hessian) in feature_points2:
    points2[count][0] = x
    points2[count][1] = y
    count+=1

  correspondence_mapper = cv.CreateMat(number_of_correspondences, 2, cv.CV_64FC1)
  count = 0
  for i in range(len(feature_points1)):
    if(cv.mGet(correspondence, i, 0) != -1):
      cv.mSet(correspondence_mapper, count, 0, i)
      cv.mSet(correspondence_mapper, count, 1, cv.mGet(correspondence, i, 0))
      count+=1


  """
  Allocate space for inlier and best_inlier variables
  """
  inliers1 = np.zeros((number_of_correspondences, 2), float)
  inliers2 = np.zeros((number_of_correspondences, 2), float)
  best_inliers1 = np.zeros((number_of_correspondences, 2), float)
  best_inliers2 = np.zeros((number_of_correspondences, 2), float)
  #the actual Ransac Algorithm
  global INLIER_DISTANCE_THRESHOLD
  global NUMBER_OF_INLIERS_THRESHOLD
  trials = 0
  N = 1e3
  #N is updated at the end of each loop
  while trials < N:
```

```
temp_H = cv.CreateMat(3, 3, cv.CV_64FC1)
A_matrix_as_vector = np.zeros(128, float)
B_vector = np.zeros(16, float)
A = cv.CreateMat(16, 8, cv.CV_64FC1)
B = cv.CreateMat(16, 1, cv.CV_64FC1)
x = cv.CreateMat(8, 1, cv.CV_64FC1)
for i in range(8):#takes 8 random points and uses linear least squares
  index = np.random.randint(number_of_correspondences, size = 1)[0]
  A_matrix_as_vector[16*i + 0] = points1[cv.mGet(correspondence_mapper, index, 0)
      ][0]
  A_matrix_as_vector[16*i + 1] = points1[cv.mGet(correspondence_mapper, index, 0)
      ][1]
  A_matrix_as_vector[16*i + 2] = 1
  A_matrix_as_vector[16*i + 3] = 0
  A_matrix_as_vector[16*i + 4] = 0
  A_matrix_as_vector[16*i + 5] = 0
  A_matrix_as_vector[16*i + 6] = -points1[cv.mGet(correspondence_mapper,index, 0)
      ][0]*points2[cv.mGet(correspondence_mapper, index, 1)][0]
  A_matrix_as_vector[16*i + 7] = -points1[cv.mGet(correspondence_mapper,index, 0)
      ][1]*points2[cv.mGet(correspondence_mapper, index, 1)][0]

  A_matrix_as_vector[8*(2*i+1) + 0] = 0
  A_matrix_as_vector[8*(2*i+1) + 1] = 0
  A_matrix_as_vector[8*(2*i+1) + 2] = 0
  A_matrix_as_vector[8*(2*i+1) + 3] = points1[cv.mGet(correspondence_mapper, index
      , 0)][0]
  A_matrix_as_vector[8*(2*i+1) + 4] = points1[cv.mGet(correspondence_mapper, index
      , 0)][1]
  A_matrix_as_vector[8*(2*i+1) + 5] = 1
  A_matrix_as_vector[8*(2*i+1) + 6] = -points1[cv.mGet(correspondence_mapper,index
      , 0)][0]*points2[cv.mGet(correspondence_mapper, index, 1)][1]
  A_matrix_as_vector[8*(2*i+1) + 7] = -points1[cv.mGet(correspondence_mapper,index
      , 0)][1]*points2[cv.mGet(correspondence_mapper, index, 1)][1]

  B_vector[2*i] = points2[cv.mGet(correspondence_mapper, index, 1)][0]
  B_vector[2*i+1] = points2[cv.mGet(correspondence_mapper, index, 1)][1]

cv.SetData(A, A_matrix_as_vector, cv.CV_AUTOSTEP)
cv.SetData(B, B_vector, cv.CV_AUTOSTEP)
cv.Solve(A, B, x, cv.CV_LU)
A_transpose = cv.CreateMat(8, 16, cv.CV_64FC1)
cv.Transpose(A, A_transpose)
#computes Pseudo Inverse
A_transpose_A = cv.CreateMat(8, 8, cv.CV_64FC1)
A_transpose_A_inverse = cv.CreateMat(8, 8, cv.CV_64FC1)
cv.MatMul(A_transpose, A, A_transpose_A)
cv.Invert(A_transpose_A, A_transpose_A_inverse, cv.CV_SVD)

A_pseudo_inverse = cv.CreateMat(8, 16, cv.CV_64FC1)
cv.MatMul(A_transpose_A_inverse, A_transpose, A_pseudo_inverse)
#calculates H
cv.MatMul(A_pseudo_inverse, B, x)

for j in range(3):
  for k in range(3):
    if (j*3+k)!=8:
cv.mSet(temp_H, j , k, cv.mGet(x, j*3 + k, 0))
```

```
else:
  cv.mSet(temp_H, 2, 2, 1)

  #calculate total error
  number_of_inliers = 0
  error = 0
  error_squared = 0
  for i in range(number_of_correspondences):
    temp_point1 = cv.CreateMat(3, 1, cv.CV_64FC1)
    temp_point2 = cv.CreateMat(3, 1, cv.CV_64FC1)
    cv.mSet(temp_point1, 0, 0, points1[cv.mGet(correspondence_mapper, i, 0)][0])
    cv.mSet(temp_point1, 1, 0, points1[cv.mGet(correspondence_mapper, i, 0)][1])
    cv.mSet(temp_point1, 2, 0, 1)
    cv.MatMul(temp_H, temp_point1, temp_point2)
    dx = cv.mGet(temp_point2, 0, 0)/cv.mGet(temp_point2, 2, 0) - points2[cv.mGet(
        correspondence_mapper, i, 1)][0]
    dy = cv.mGet(temp_point2, 1, 0)/cv.mGet(temp_point2, 2, 0) - points2[cv.mGet(
        correspondence_mapper, i, 1)][1]
    #calculates the euclidean distance between the true coordinates and the
    #calculated coordinates
    temp_error = np.power(dx*dx + dy*dy, 0.5)
    #print temp_error
    if temp_error < INLIER_DISTANCE_THRESHOLD:#threshold
      inliers1[number_of_inliers][0] = points1[cv.mGet(correspondence_mapper, i, 0)
          ][0]
      inliers1[number_of_inliers][1] = points1[cv.mGet(correspondence_mapper, i, 0)
          ][1]
      inliers2[number_of_inliers][0] = points2[cv.mGet(correspondence_mapper, i, 1)
          ][0]
      inliers2[number_of_inliers][1] = points2[cv.mGet(correspondence_mapper, i, 1)
          ][1]
      error = temp_error + error
      error_squared = np.power(temp_error, 2) + error_squared
      number_of_inliers+=1

  #Checks if the current estimate is the best estimate
  max_inliers = 0
  min_variance = 1e10
  if number_of_inliers > max_inliers:
    mean = float(error)/float(number_of_inliers)
    variance = error_squared/float(number_of_inliers) - np.power(mean, 2)
    if variance < min_variance:
      max_inliers = number_of_inliers
      min_variance = variance
      cv.Copy(temp_H, H)
      for i in range(number_of_inliers):
        best_inliers1[i][0] = inliers1[i][0]
        best_inliers1[i][1] = inliers1[i][1]
        best_inliers2[i][0] = inliers2[i][0]
        best_inliers2[i][1] = inliers2[i][1]
  trials+=1

  #updates N
  if number_of_inliers > 0:
    e = 1.0 - float(number_of_inliers)/float(number_of_correspondences)
    e_1 = 1.0 - e
    if e_1 == 1:
      break
    if np.log(1.0-e_1*e_1*e_1*e_1*e_1*e_1*e_1*e_1)!=0:
      N = int(np.log(1.0-0.99)/np.log(1.0-e_1*e_1*e_1*e_1*e_1*e_1*e_1*e_1))
if float(number_of_inliers) / float(number_of_correspondences) <
   NUMBER_OF_INLIERS_THRESHOLD and trials > N:
  trials = 1

print "Matrix H is"
printMatrix(H)
print "Number of Inliers =", number_of_inliers," of", number_of_correspondences
global PERFORM_LM
if PERFORM_LM == 1:
  #Not working for some inputs!
  #refining H using levenberg Marquardt
  #construct inputs to the error function
  global A_matrix
  global b_values
  A_matrix = np.zeros((2*number_of_inliers, 8), float)
  b_values = np.zeros((2*number_of_inliers, 1), float)
  for index in range(number_of_inliers):
    A_matrix[2*i][0] = best_inliers1[i][0]
    A_matrix[2*i][1] = best_inliers1[i][1]
    A_matrix[2*i][2] = 1
    A_matrix[2*i][3] = 0
    A_matrix[2*i][4] = 0
    A_matrix[2*i][5] = 0
    A_matrix[2*i][6] = -best_inliers1[i][0]*best_inliers2[i][0]
    A_matrix[2*i][7] = -best_inliers1[i][1]*best_inliers2[i][0]

    A_matrix[(2*i+1)][0] = 0
    A_matrix[(2*i+1)][1] = 0
    A_matrix[(2*i+1)][2] = 0
    A_matrix[(2*i+1)][3] = best_inliers1[i][0]
    A_matrix[(2*i+1)][4] = best_inliers1[i][1]
    A_matrix[(2*i+1)][5] = 1
    A_matrix[(2*i+1)][6] = -best_inliers1[i][0]*best_inliers2[i][1]
    A_matrix[(2*i+1)][7] = -best_inliers1[i][1]*best_inliers2[i][1]

    b_values[2*i] = best_inliers2[i][0]
    b_values[2*i+1] = best_inliers2[i][1]

  h = np.zeros((8, 1), float)
  p=lmfit.Parameters()
  p.add_many(('h0', cv.mGet(H, 0, 0)), ('h1', cv.mGet(H, 0, 1)), ('h2', cv.mGet(H,
      0, 2)), ('h3', cv.mGet(H, 1, 0)), ('h4', cv.mGet(H, 1, 1)),
  ('h5', cv.mGet(H, 1, 2)), ('h6', cv.mGet(H, 2, 0)), ('h7', cv.mGet(H, 2,1)))

  result=lmfit.minimize(error_func, p)
  #lmfit.printfuncs.report_errors(result.params)
  cv.mSet(H, 0, 0, result.params['h0'].value)
  cv.mSet(H, 0, 1, result.params['h1'].value)
  cv.mSet(H, 0, 2, result.params['h2'].value)
  cv.mSet(H, 1, 0, result.params['h3'].value)
  cv.mSet(H, 1, 1, result.params['h4'].value)
  cv.mSet(H, 1, 2, result.params['h5'].value)
  cv.mSet(H, 2, 0, result.params['h6'].value)
  cv.mSet(H, 2, 1, result.params['h7'].value)
```

```
    print "Refined Matrix H"
    printMatrix(H)
  print "------------------------------------------------------------"
  return (best_inliers1, best_inliers2)


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



"""
Computes the boundaries of a given image in the image plane of another
"""
def findBoundaries(image1, image2, H):
  #image2 is centre_image

  width = image1.width
  height = image1.height

  #computes boundaries
  boundaries_image1 = cv.CreateMat(3, 4, cv.CV_64FC1)
  boundaries_image2 = cv.CreateMat(3, 4, cv.CV_64FC1)
  cv.mSet(boundaries_image1, 0, 0, 0)
  cv.mSet(boundaries_image1, 0, 1, width-1)
  cv.mSet(boundaries_image1, 0, 2, 0)
  cv.mSet(boundaries_image1, 0, 3, width-1)
  cv.mSet(boundaries_image1, 1, 0, 0)
  cv.mSet(boundaries_image1, 1, 1, 0)
  cv.mSet(boundaries_image1, 1, 2, height-1)
  cv.mSet(boundaries_image1, 1, 3, height-1)
  cv.mSet(boundaries_image1, 2, 0, 1)
  cv.mSet(boundaries_image1, 2, 1, 1)
  cv.mSet(boundaries_image1, 2, 2, 1)
  cv.mSet(boundaries_image1, 2, 3, 1)
  cv.MatMul(H, boundaries_image1, boundaries_image2)

  #checks boundaries in image2 coordinates to get the min and max values
  x_min = 1e100
  y_min = 1e100
  x_max = 0
  y_max = 0

  for i in range(4):
    x = cv.mGet(boundaries_image2, 0, i)/cv.mGet(boundaries_image2, 2,i)
    y = cv.mGet(boundaries_image2, 1, i)/cv.mGet(boundaries_image2, 2,i)
    if x < x_min:
      x_min = x
    if x > x_max:
      x_max = x
    if y < y_min:
      y_min = y
    if y > y_max:
      y_max = y

  return (x_min, x_max, y_min, y_max)

"""
main function
"""
def main():
  print "@Author: S.Karthik Badam"

  #initializes variables
  images = []
  grey_scale_images = []
  feature_points = []
  feature_descriptors = []
  filename = "image"

  #Take input from user -> Please enter an odd number!
  global NUMBER_OF_IMAGES
  print "enter number of images = "
  NUMBER_OF_IMAGES=int(raw_input())

  #Computes Feature Points and Feature Descriptors using SURF
  for i in range(NUMBER_OF_IMAGES):
    #loads image -> Please configure the directory  name and path appropriately!!
    #temp_image = cv.LoadImage("car/"+filename+`i+1`+".jpg", cv.
        CV_LOAD_IMAGE_UNCHANGED)
    temp_image = cv.LoadImage("home1/"+filename+`i+1`+".jpg", cv.
        CV_LOAD_IMAGE_UNCHANGED)
    images.append(temp_image)
    if temp_image == 0:
      print "enter a valid Image Path for Image ",i
    grey_temp_image = cv.CreateImage((temp_image.width, temp_image.height),temp_image.
      depth, 1)
    cv.CvtColor(temp_image, grey_temp_image, cv.CV_RGB2GRAY)
    grey_scale_images.append(grey_temp_image)
    (temp_feature_points, temp_feature_descriptors) = SURFdetector(grey_temp_image)
    feature_points.append(temp_feature_points)
    feature_descriptors.append(temp_feature_descriptors)


  correspondences = []
  H = []
  inliers = []

  global PERFORM_LM
  print "Enter '1' to perform LM, otherwise enter '0' "
  PERFORM_LM = int(raw_input())

  #Calculates the correspondences and also applies RANSAC to get the best
  #Estimate for Homography

  #for images on the left of centre images
  for i in range(int(NUMBER_OF_IMAGES/2)):
    H_temp = cv.CreateMat(3, 3, cv.CV_64FC1)
    correspondence_temp = cv.CreateMat(len(feature_points[i]), 1, cv.CV_64FC1)
    correspondence_finder(feature_points[i], feature_points[i+1],
        feature_descriptors[i], feature_descriptors[i+1], correspondence_temp)
    correspondences.append(correspondence_temp)
    (inliers_temp1, inliers_temp2)= RANSAC(feature_points[i], feature_points[i+1],
        correspondence_temp, H_temp)
    H.append(H_temp)
    inliers.append(inliers_temp1)
    #draw the inliers and outliers
    if i == 0 or i==1:

      inlier_image = cv.CreateImage((2*images[i].width, images[i].height), images[i].
          depth, 3)
      cv.SetImageROI(inlier_image, (0, 0, images[i].width, images[i].height))
      cv.Copy(images[i], inlier_image)

      #marks the inliers in the first part
      for j in range(len(inliers_temp1)):
        if inliers_temp1[j][0]!=0:
          cv.Circle(inlier_image, (int(inliers_temp1[j][0]), int(inliers_temp1[j][1]))
              , 0, (255, 0, 0), 4)
      cv.ResetImageROI(inlier_image)

      cv.SetImageROI(inlier_image, (images[i].width, 0, images[i].width, images[i].
          height))
      cv.Copy(images[i+1], inlier_image)

      #marks the inliers in the second part of the image
      for j in range(len(inliers_temp2)):
        if inliers_temp2[j][0]!=0:
          cv.Circle(inlier_image, (int(inliers_temp2[j][0]),
            int(inliers_temp2[j][1])), 0, (255, 0, 0), 4)
      cv.ResetImageROI(inlier_image)

      #Draws all correspondences
      for j in range(len(feature_points[i])):
        value = cv.mGet(correspondence_temp, j,0)
        if value != -1:
          ((x1,y1), laplacian1, size1, dir, hessian1) = feature_points[i][j]
          ((x2,y2), laplacian2, size2, dir, hessian2) = feature_points[i+1][int(value)
              ]
          #Multiple line colors are used to avoid confusion with overlaps
          cv.Circle(inlier_image, (int(x1),int(y1)), 0, (255, 0, 0), 8)
          cv.Circle(inlier_image, (int(x2)+images[0].width,int(y2)), 0, (255, 0, 0),
              8)
          cv.Line(inlier_image, (int(x1),int(y1)),
              (int(x2)+images[0].width,int(y2)), (0, 0, 255), 1, cv.CV_AA, 0)

      #changes the color of inliers
      #shows correspondences by drawing a line between matching Interest Points
      count = 0
      for j in range(len(inliers_temp1)):
        if inliers_temp1[j][0]!=0:
          cv.Line(inlier_image,(int(inliers_temp1[j][0]),int(inliers_temp1[j][1])), (
              images[0].width+int(inliers_temp2[j][0]),int(inliers_temp2[j][1])), (0,
              255, 0), 1, cv.CV_AA, 0)

      cv.SaveImage("inlier"+`i`+".png", inlier_image)
```

```
#Appends a Identity Homography Matrix for the centre image
H.append(cv.CreateMat(3,3,cv.CV_64FC1))

#For images on the right of centre image
i = int(NUMBER_OF_IMAGES/2)+1
while i < NUMBER_OF_IMAGES:
  H_temp = cv.CreateMat(3, 3, cv.CV_64FC1)
  correspondence_temp = cv.CreateMat(len(feature_points[i]), 1, cv.CV_64FC1)
  correspondence_finder(feature_points[i], feature_points[i-1],
      feature_descriptors[i], feature_descriptors[i-1], correspondence_temp)
  correspondences.append(correspondence_temp)
  (inliers_temp1, inliers_temp2) = RANSAC(feature_points[i], feature_points[i-1],
      correspondence_temp, H_temp)
  H.append(H_temp)
  inliers.append(inliers_temp1)
  i+=1


#compute homographies w.r.t centre image
i = int(NUMBER_OF_IMAGES/2) - 2
while i >= 0:
  cv.MatMul(H[i],H[i+1],H[i])
  i-=1

i = int(NUMBER_OF_IMAGES/2) + 2
while i < NUMBER_OF_IMAGES:
  cv.MatMul(H[i], H[i-1], H[i])
  i+=1


x_min = np.zeros(NUMBER_OF_IMAGES-1, float)
x_max = np.zeros(NUMBER_OF_IMAGES-1, float)
y_min = np.zeros(NUMBER_OF_IMAGES-1, float)
y_max = np.zeros(NUMBER_OF_IMAGES-1, float)

#computes the boundaries of each image with respect to centre image
count = 0
#combine each image with centre image
#compute the boundaries of each projection
for i in range(NUMBER_OF_IMAGES):
  if i!=int(NUMBER_OF_IMAGES/2):
    (x_min[count], x_max[count], y_min[count], y_max[count]) = findBoundaries(images
        [i],images[int(NUMBER_OF_IMAGES/2)], H[i])
    count+=1

print "Boundaries are:"
print "x_min ="
print x_min

print "x_max = "
print x_max

print "y_min = "
print y_min

print "y_max = "
print y_max

final_x_min = 1e10
final_y_min = 1e10
final_x_max = 0
final_y_max = 0

#calculates the dimensions of the final mosaic
for i in range(NUMBER_OF_IMAGES-1):
  if final_x_min > x_min[i]:
    final_x_min = x_min[i]
  if final_x_max < x_max[i]:
    final_x_max = x_max[i]
  if final_y_min > y_min[i]:
    final_y_min = y_min[i]
  if final_y_max < y_max[i]:
    final_y_max = y_max[i]

#print final_x_min, final_x_max, final_y_min, final_y_max
#creates the final mosaic
final_image = cv.CreateImage((int(final_x_max - final_x_min), int(final_y_max -
    final_y_min)), images[0].depth, 3)
cv.SetImageROI(final_image, (int(0-final_x_min), int(0-final_y_min), images[int(
    NUMBER_OF_IMAGES/2)].width, images[int(NUMBER_OF_IMAGES/2)].height))
#copies the centre image into its place
cv.Copy(images[int(NUMBER_OF_IMAGES/2)], final_image)
cv.ResetImageROI(final_image)

#calculates the inverse of the homography
H_inverse = []
for i in range(NUMBER_OF_IMAGES):
  if i!= int(NUMBER_OF_IMAGES/2):
    H_inverse_temp = cv.CreateMat(3, 3, cv.CV_64FC1)
    cv.Invert(H[i], H_inverse_temp, cv.CV_LU)
    H_inverse.append(H_inverse_temp)
  else:
    H_inverse.append(cv.CreateMat(3,3,cv.CV_64FC1))

#Fills the mosaic!
#world_coordinates mean the coordinates in the mosaic's plane
world_coordinates = cv.CreateMat(3, 1, cv.CV_64FC1)
image_coordinates = cv.CreateMat(3, 1, cv.CV_64FC1)
cv.mSet(world_coordinates, 2, 0, 1)
for i in range(final_image.width):
  for j in range(final_image.height):
    #stitch left side images
    if i < images[int(NUMBER_OF_IMAGES/2)].width - final_x_min:
      image_number = int(NUMBER_OF_IMAGES/2) - 1
      #finds the image the pixel belongs to and writes the data
      while image_number >= 0:
        temp_i = i + final_x_min
        temp_j = j + final_y_min
        cv.mSet(world_coordinates, 0, 0, temp_i)
        cv.mSet(world_coordinates, 1, 0, temp_j)
        cv.MatMul(H_inverse[image_number], world_coordinates, image_coordinates)
        x = cv.mGet(image_coordinates, 0, 0)/cv.mGet(image_coordinates, 2, 0)
        y = cv.mGet(image_coordinates, 1, 0)/cv.mGet(image_coordinates, 2, 0)
```

```
          if temp_i > x_min[image_number] and temp_i < x_max[image_number] and temp_j
              > y_min[image_number] and temp_j < y_max[image_number]:
            temp = [0,0,0]
            if x > 0 and x < temp_image.width - 1 and  y > 0 and y <temp_image.height
                - 1:
              temp_image = images[image_number]
        for k in range(3):
          temp[k] = 0
    temp[k] =  temp[k]+(x-int(x))*(y-int(y))*(temp_image[int(y+1),int(x+1)][k])
    temp[k] = temp[k]+(1.0-(x-int(x)))*(y-int(y))*(temp_image[int(y+1),int(x)][k])
    temp[k] = temp[k]+(x-int(x))*(1.0-(y-int(y)))*(temp_image[int(y),int(x+1)][k])
    temp[k] = temp[k]+(1.0-(x-int(x)))*(1.0-(y-int(y)))*(temp_image[int(y),int(x)][k])
              if final_image[j,i][0] == 0 and final_image[j, i][1] == 0 and temp
                  !=[0,0,0]:
                #writes only if the pixel is empty
                final_image[j,i] =  temp
              break
          image_number-=1


      #stitching the right side images
      if i > -final_x_min:
        for image_number in range(int(NUMBER_OF_IMAGES/2)+1, NUMBER_OF_IMAGES):
          temp_i = i + final_x_min
          temp_j = j + final_y_min
          cv.mSet(world_coordinates, 0, 0, temp_i)
          cv.mSet(world_coordinates, 1, 0, temp_j)
          cv.MatMul(H_inverse[image_number], world_coordinates, image_coordinates)
          x = cv.mGet(image_coordinates, 0, 0)/cv.mGet(image_coordinates, 2, 0)
          y = cv.mGet(image_coordinates, 1, 0)/cv.mGet(image_coordinates, 2, 0)
          if temp_i > x_min[image_number-1] and temp_i < x_max[image_number-1]  and
              temp_j > y_min[image_number-1] and temp_j < y_max[image_number-1]:
            temp_image = images[image_number]
            temp = [0,0,0]
            if x > 0 and x < temp_image.width - 1 and  y > 0 and y < temp_image.height
                - 1:
        for k in range(3):
          temp[k] = 0
    temp[k] = temp[k]+(x-int(x))*(y-int(y))*(temp_image[int(y+1),int(x+1)][k])
          temp[k] = temp[k]+(1.0-(x-int(x)))*(y-int(y))*(temp_image[int(y+1),int(x)][k
              ])
    temp[k] = temp[k]+(x-int(x))*(1.0-(y-int(y)))*(temp_image[int(y),int(x+1)][k])
    temp[k] = temp[k]+(1.0-(x-int(x)))*(1.0-(y-int(y)))*(temp_image[int(y),int(x)][k])
              if final_image[j,i][0] == 0 and final_image[j, i][1] == 0 and temp
                  !=[0,0,0]:
                #writes only if the pixel is empty
                final_image[j,i] =  temp
              break
  cv.SaveImage("Result.png", final_image)

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