ECE 661
Computer Vision
Fall 2016
Homework 5

1 SIFT correspondence matching

This part is the same from Homework 4, the following is just the quick summary of steps.

1. SIFT keypoints and descriptors for each image are computed using OpenCV built-in function.
2. SSD is used for descriptor matching between subsequent image pairs.
3. The ratio between minimum SSD and second minimum SSD is used to find correspondences, by applying a threshold of 0.6

2 Random Sample Consensus (RANSAC)

After finding initial correspondences, the output may be riddled with false correspondences. We can get rid of them by applying RANSAC algorithm. Let n_t be the total number of correspondences found in SIFT matching stage.

1. Parameters setting for RANSAC
   - **p** : The probability that at least one of the N trials will be free of outliers. In this implementation it is set as $p = 0.99$
   - **n** : set of correspondences for calculating homography matrix H in each trial. In this implementation it is set as $n = 8$
   - ϵ : rough estimation of false correspondences from the total set. In this implementation it is set as $\epsilon = 0.1$
   - δ : decision threshold to construct the inlier set i.e. if the distance between the data point and the estimate is less than δ we place it in the inlier set. In this implementation it is set as $\delta = 40$ (pixels)
   - **N** : Number of trials, which can be given as $N = \frac{ln(1-p)}{ln(1-(1-\epsilon)^n)} = 8$
   - **M** : A minimum value for the size of the inlier set for it to be acceptable, which can be given as $M = n_t * (1 - \epsilon)$
2. Randomly select n correspondences from the initial set of SIFT matchings.

3. Calculate Homography matrix H using the selected n correspondences. H can be computed using the following $2n$ homogeneous equations

$$\begin{bmatrix} 0 & 0 & 0 & -w'_1x_1 & -w'_1y_1 & -w'_1w_1 & y'_1x_1 & y'_1y_1 & y'_1w_1 \\ w'_1x_1 & w'_1y_1 & w'_1w_1 & 0 & 0 & 0 & -x'_1x_1 & -x'_1y_1 & -x'_1w_1 \\ 0 & 0 & 0 & -w'_2x_2 & -w'_2y_2 & -w'_2w_2 & y'_2x_2 & y'_2y_2 & y'_2w_2 \\ w'_2x_2 & w'_2y_2 & w'_2w_2 & 0 & 0 & 0 & -x'_2x_2 & -x'_2y_2 & -x'_2w_2 \\ & & \vdots & & \vdots & & \vdots & & \\ & & \vdots & & \vdots & & \vdots & & \\ 0 & 0 & 0 & -w'_nx_n & -w'_ny_n & -w'_nw_n & y'_nx_n & y'_ny_n & y'_nw_n \\ w'_nx_n & w'_ny_n & w'_nw_n & 0 & 0 & 0 & -x'_nx_n & -x'_ny_n & -x'_nw_n \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = 0$$

We solve $min\ ||A\overrightarrow{h}||$ subject to $||\overrightarrow{h}|| = 1$. The solution is the eigenvector of A^TA corresponding to the smallest singular value in matrix D which we get from Singular Value Decomposition of matrix A.

4. Compute the distance between the mapped location Hx and the correspondence location x' for all correspondences found in SIFT matching stage. Whenever the distance is $< \delta$ add that correspondence in inlier set.

5. Repeat the above steps for N times and accept the inlier set with maximum matching , ideally its size should be $> M$.

3 Homography refinement using DogLeg

DogLeg (DL) combines the best of Gradient-Descent (GD) and Gauss-Newton (GN) method. It's based on the notion of a trust region of radius r_k around the point $\overrightarrow{p_k}$. The trust region decides the the step size we want to take in the next iteration of optimization.

$$\overrightarrow{\delta}_{p,GD} = \frac{||J_{\overrightarrow{f}}^T \overrightarrow{\epsilon}(\overrightarrow{p_k})||}{||J_{\overrightarrow{f}} J_{\overrightarrow{f}}^T \overrightarrow{\epsilon}(\overrightarrow{p_k})||} J_{\overrightarrow{f}}^T \overrightarrow{\epsilon}(\overrightarrow{p_k}) \tag{1}$$

$$\overrightarrow{\delta}_{p,GN} = \frac{1}{J_{\overrightarrow{f}}^T J_{\overrightarrow{f}} + \mu_k I} J_{\overrightarrow{f}}^T \overrightarrow{\epsilon}(\overrightarrow{p_k}) \tag{2}$$

where $\overrightarrow{f}$ represents mapped point in physical coordinates and $J_{\overrightarrow{f}}$ is Jacobian matrix w.r.t. $\overrightarrow{p_k}$. $\overrightarrow{\epsilon}(\overrightarrow{p}) = \overrightarrow{X} - \overrightarrow{f}(\overrightarrow{p})$; $\overrightarrow{p_k} = \begin{pmatrix} h_{11} & h_{12} & h_{13} & h_{21} & h_{22} & h_{23} & h_{31} & h_{32} & h_{33} \end{pmatrix}^T$

$$\overrightarrow{f}(\overrightarrow{p_k}) = \begin{pmatrix} f_1(\overrightarrow{p}_k) \\ f_2(\overrightarrow{p}_k) \\ \vdots \\ f_{2N}(\overrightarrow{p}_k) \end{pmatrix} = \begin{pmatrix} \frac{h_{11}x_1+h_{12}y_1+h_{13}}{h_{31}x_1+h_{32}y_1+h_{33}} \\ \frac{h_{21}x_1+h_{22}y_1+h_{23}}{h_{31}x_1+h_{32}y_1+h_{33}} \\ \vdots \\ \frac{h_{11}x_N+h_{12}y_N+h_{13}}{h_{31}x_N+h_{32}y_N+h_{33}} \\ \frac{h_{21}x_N+h_{22}y_N+h_{23}}{h_{31}x_N+h_{32}y_N+h_{33}} \end{pmatrix}$$

$\overrightarrow{p}_k$ is updated to $\overrightarrow{p}_{k+1}$ using the following logic

$$\overrightarrow{p}_{k+1} = \overrightarrow{p}_k + \begin{cases} \overrightarrow{\delta}_{p,GN} & \text{if } ||\overrightarrow{\delta}_{p,GN}|| < r_k \\ \overrightarrow{\delta}_{p,GD} + \beta(\overrightarrow{\delta}_{p,GN} - \overrightarrow{\delta}_{p,GD}) & \text{if } ||\overrightarrow{\delta}_{p,GD}|| < r_k < ||\overrightarrow{\delta}_{p,GN}|| \\ \frac{r_k}{||\overrightarrow{\delta}_{p,GD}||} \cdot \overrightarrow{\delta}_{p,GD} & otherwise \end{cases}$$

We can obtain β by solving $||\overrightarrow{\delta}_{p,GD} + \beta(\overrightarrow{\delta}_{p,GN} - \overrightarrow{\delta}_{p,GD})||^2 = r_k^2$. Finally we update r_k based on ρ^{DL} which is defined as

$$\rho^{DL} = \frac{C(\overrightarrow{p}_k) - C(\overrightarrow{p}_{k+1})}{2\overrightarrow{\delta}_p J_{\overrightarrow{f}}^T \overrightarrow{\epsilon}(\overrightarrow{p}_k) - \overrightarrow{\delta}_p^T J_{\overrightarrow{f}}^T J_{\overrightarrow{f}} \overrightarrow{\delta}_p}$$

where $C(\overrightarrow{p}) = \overrightarrow{\epsilon}^T(\overrightarrow{p})\overrightarrow{\epsilon}(\overrightarrow{p})$. r_k is updated as

$$r_{k+1} = \begin{cases} r_k/4 & \text{if } \rho^{DL} < 1/4 \\ r_k & \text{if } 1/4 < \rho^{DL} \leq 3/4 \\ 2r_k & otherwise \end{cases}$$

We repeat the above procedure until $\rho^{DL} \leq 0$.

4 Image mosaicing to the center image

Given 5 input images first we compute Homography matrices H_{ij} for mapping points from image i to image j. For 5 images we compute 4 matrices: H_{01}, H_{12}, H_{32}, H_{43}, either using only RANSAC or followed by refinement using DogLeg or bundle adjustment. As we want to map all the images to the center image 2, we compute

$$H_{02} = H_{01} * H_{12}$$

$$H_{42} = H_{43} * H_{32}$$

Then using H_{02},H_{12}, H_{32} and H_{42} we map the 4 images onto center image and for the center image itself simple 3×3 identity matrix is used , in order to keep the same function for generating output mosaic. Scaling logic is same as earlier homeworks, in order to avoid very large output image.

5 Refine Homographies using Bundle adjustment (Extra Credit)

Here we want to refine homographies between all image pairs together and we minimize the cumulative error which is defined as

$$e = \sum_{i=1}^{n} \sum_{j \in I(i)} \sum_{k \in F(i,j)} h(r_{ij}^k)$$

where $r_{ij}^k = u_i^k - p_{ij}^k$ for u_i^k is kth feature of image i and p_{ij}^k is projection from image j to image i. h represents the L_2 norm. Levmar package is used for minimizing this error, for Windows c++ version, CLAPACK is used as an external dependency. `dlevmar_dif` function call from Levmar can be used which does not need a function pointer for computing Jacobian. It takes a function pointer as a first input , it's defined as

```
void(*err)(double *p, double *hx, int m, int n, void *adata);
```

where the first value can be passed as all parameters from homography matrices stacked together and passed as an array. Second parameter is X' values , as we want to minimize distance with respect to these points. m and n are their respective sizes. A global array of X values is also used in the error function definition.

6 Results

The input images (only) are taken from [1], they are just a subset of the images available on the link. However, no part of the code is used in this homework implementation.

[1] https://github.com/ppwwyyxx/panorama/releases/tag/0.1

6.1 Image set 1

Figure 1: Input image 0

Figure 2: Input image 1

Figure 3: Input image 2

Figure 4: Input image 3

Figure 5: Input image 4

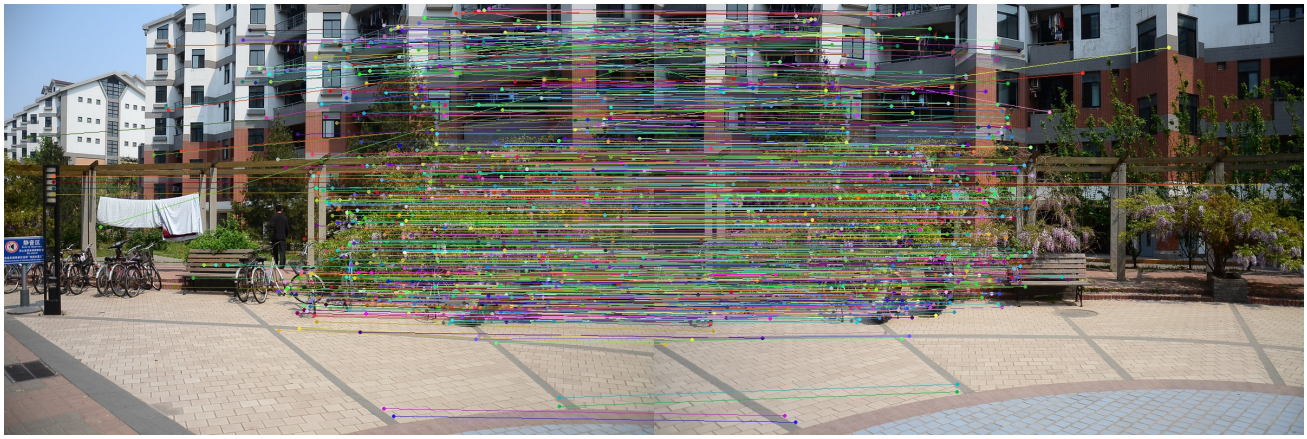

Figure 6: SIFT correspondences between images 0 and 1

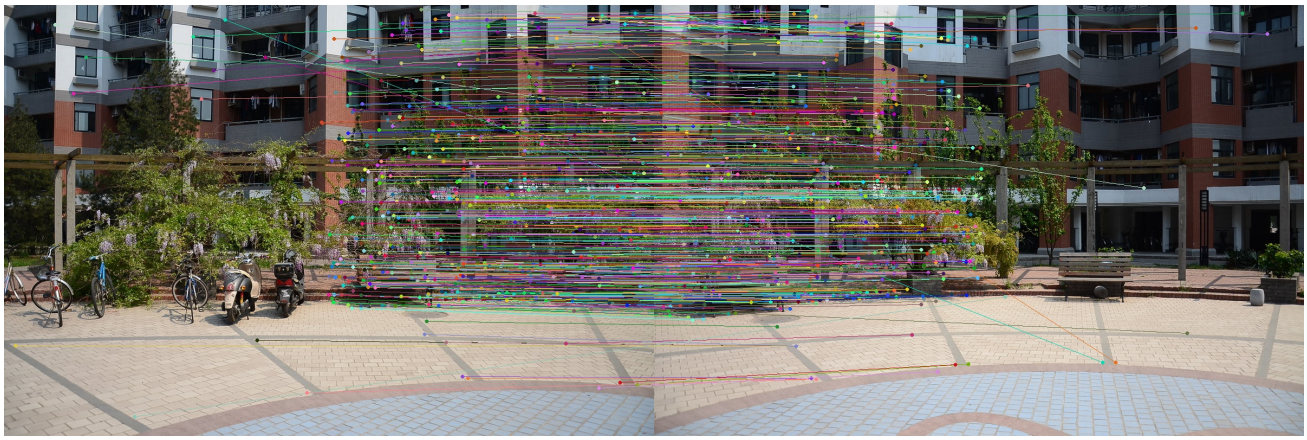

Figure 7: SIFT correspondences between images 1 and 2

Figure 8: SIFT correspondences between images 3 and 2

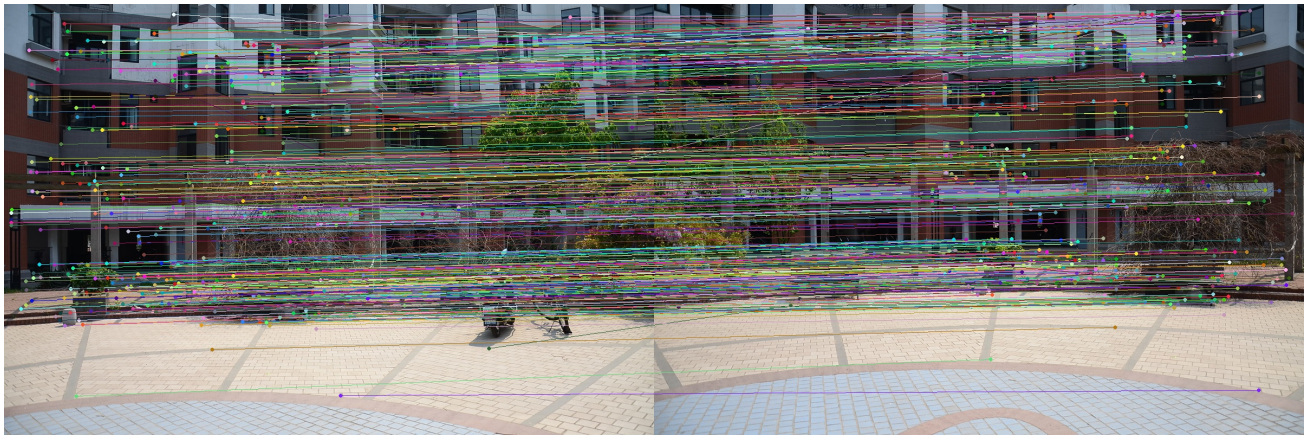

Figure 9: SIFT correspondences between images 4 and 3

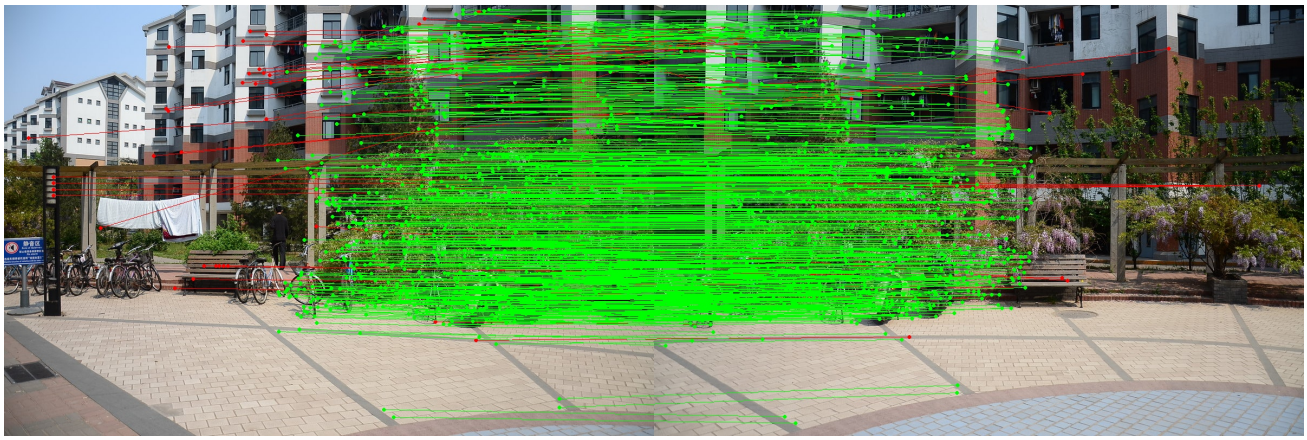

Figure 10: Inliers(green) and outliers(red) between images 0 and 1

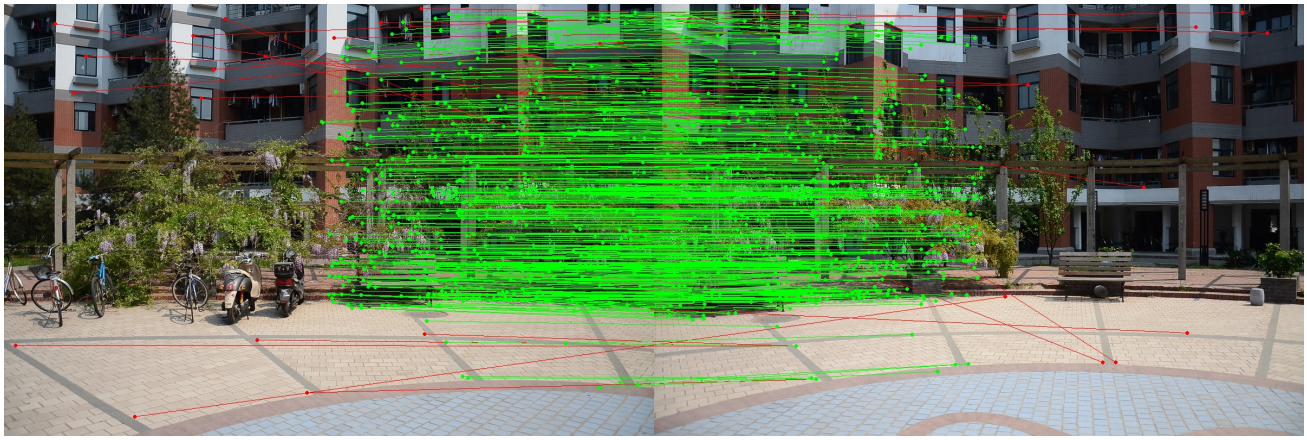

Figure 11: Inliers(green) and outliers(red) between images 1 and 2

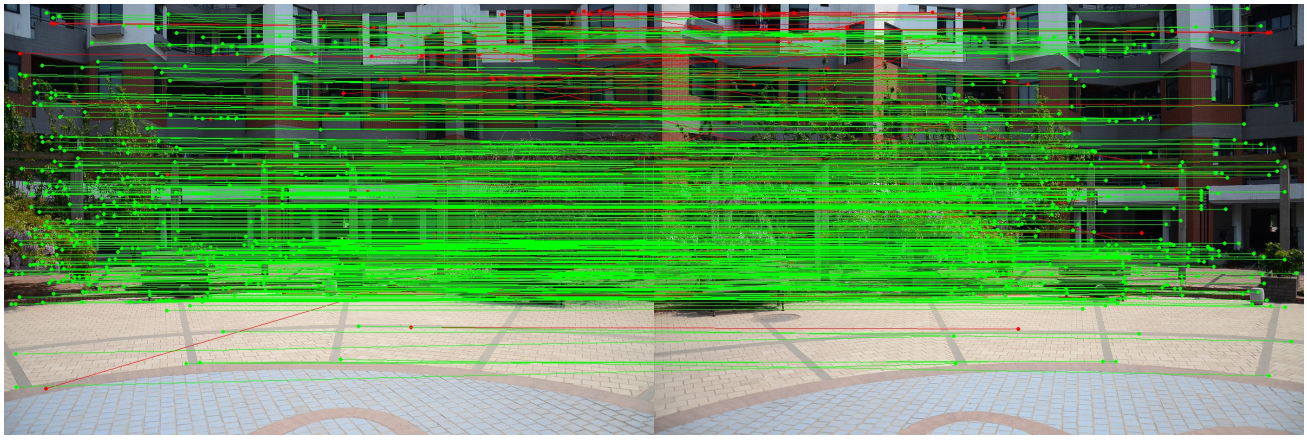

Figure 12: Inliers(green) and outliers(red) between images 3 and 2

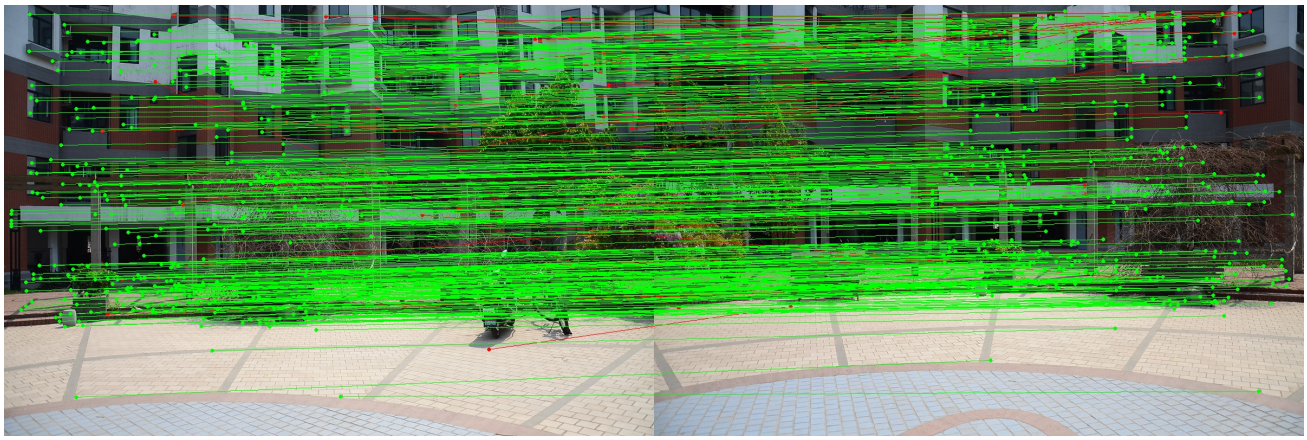

Figure 13: Inliers(green) and outliers(red) between images 4 and 3

Figure 14: Mosaic output without homography refinement

Figure 15: Mosaic output with DogLeg homography refinement

Figure 16: Mosaic output with homography refinement using bundle adjustment

6.2 Image Set 2

Figure 17: Input image 0

Figure 18: Input image 1

Figure 19: Input image 2

Figure 20: Input image 3

Figure 21: Input image 4

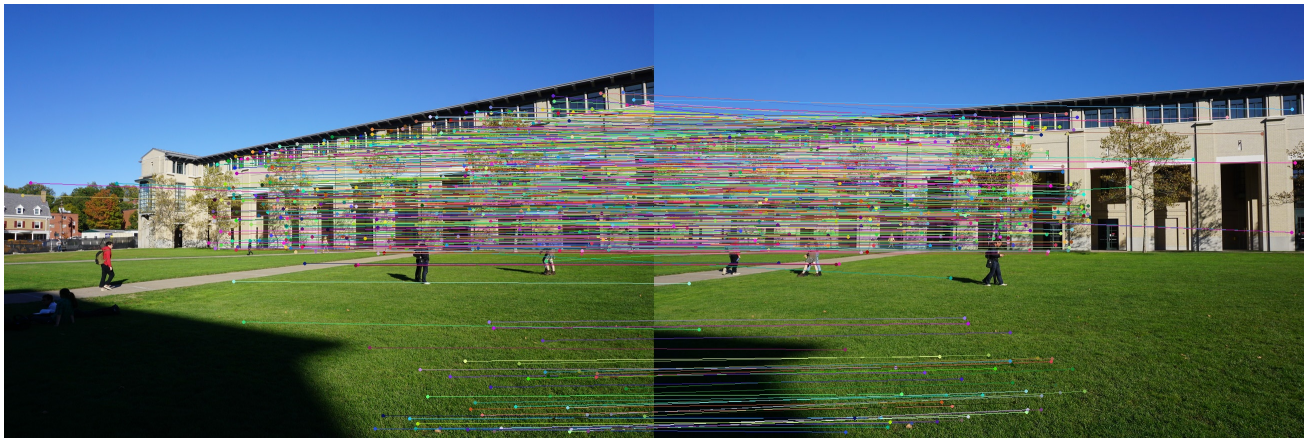

Figure 22: SIFT correspondences between images 0 and 1

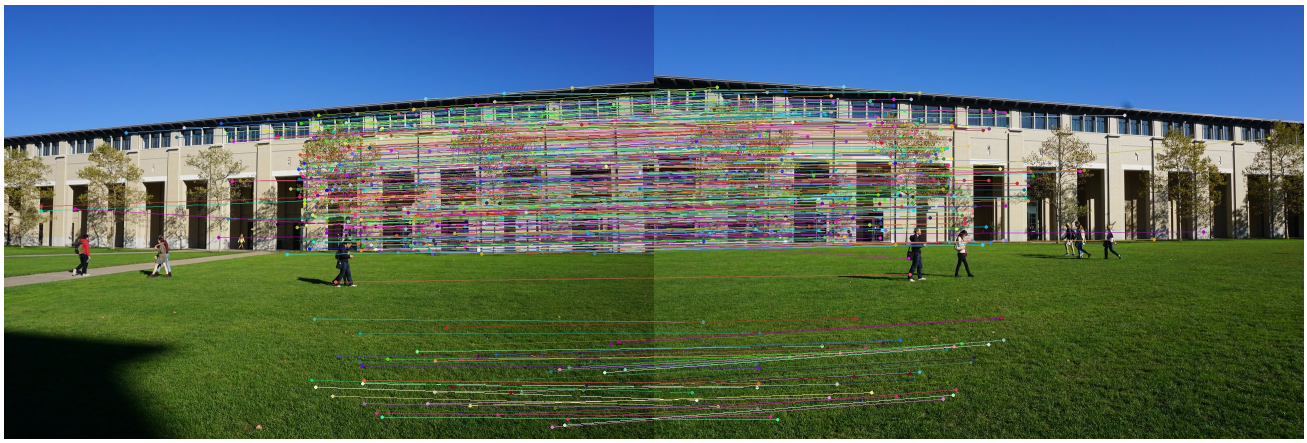

Figure 23: SIFT correspondences between images 1 and 2

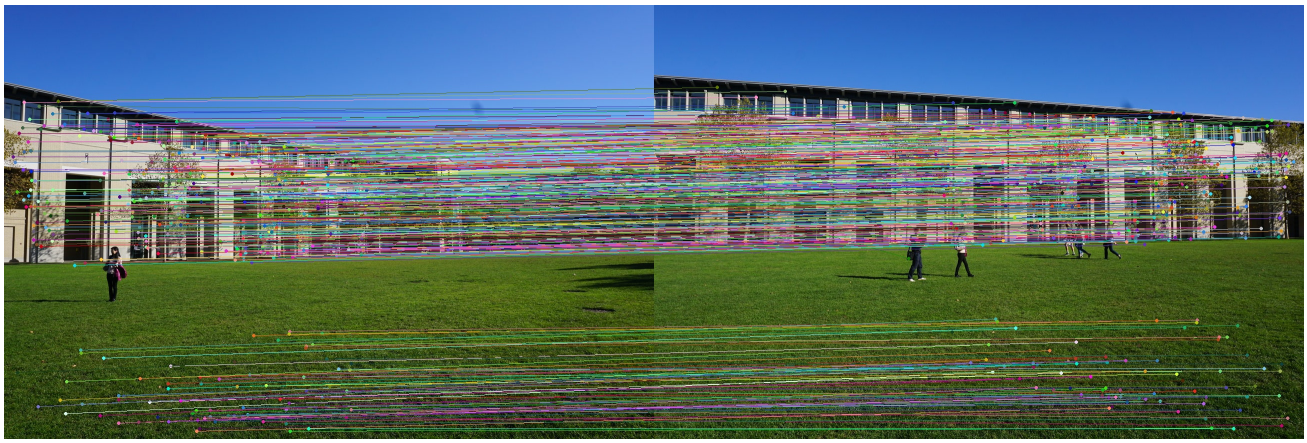

Figure 24: SIFT correspondences between images 3 and 2

Figure 25: SIFT correspondences between images 4 and 3

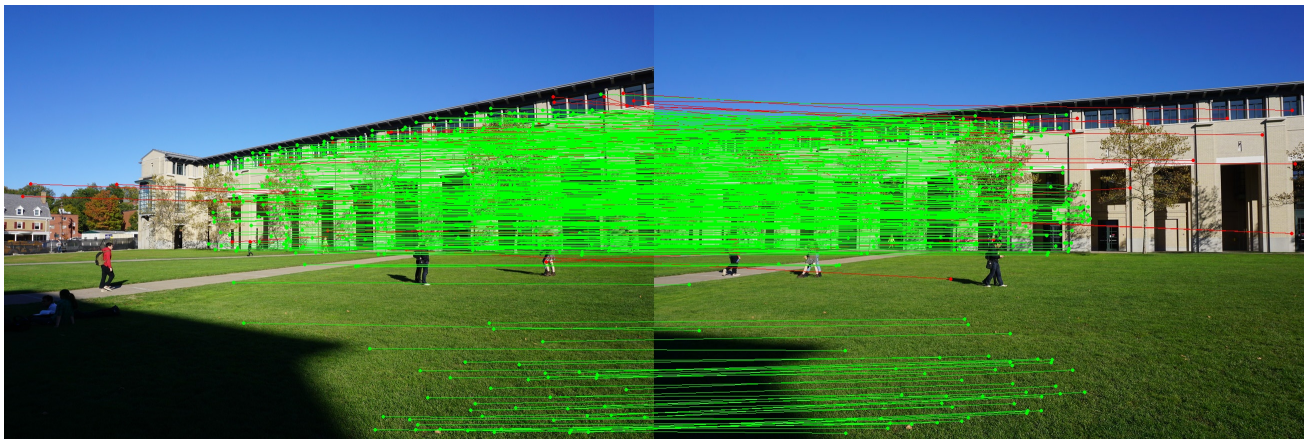

Figure 26: Inliers(green) and outliers(red) between images 0 and 1

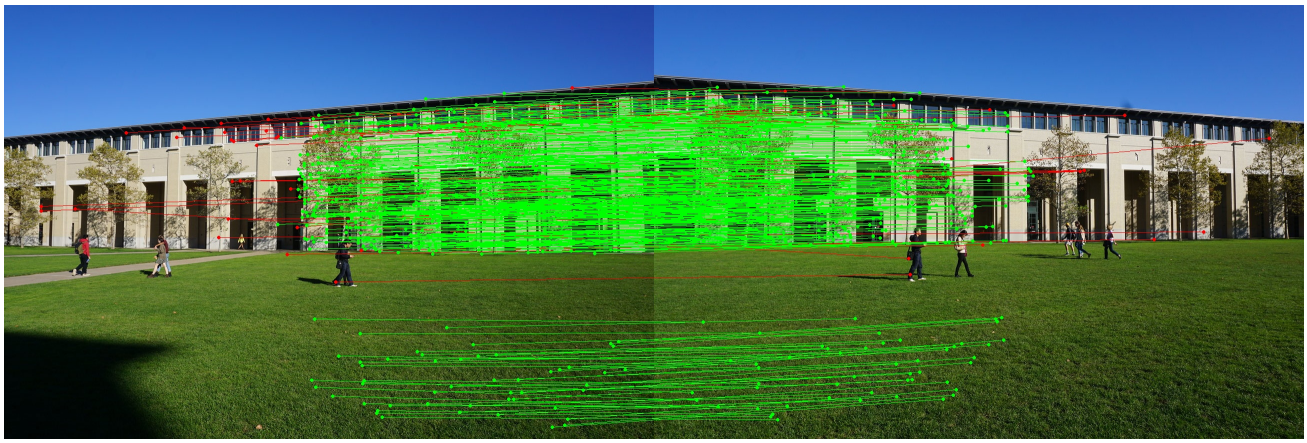

Figure 27: Inliers(green) and outliers(red) between images 1 and 2

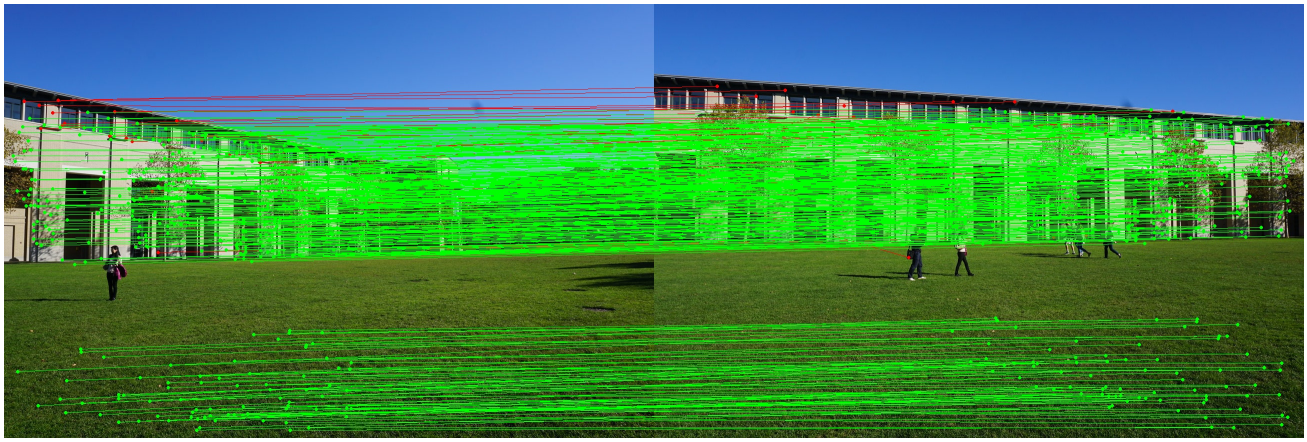

Figure 28: Inliers(green) and outliers(red) between images 3 and 2

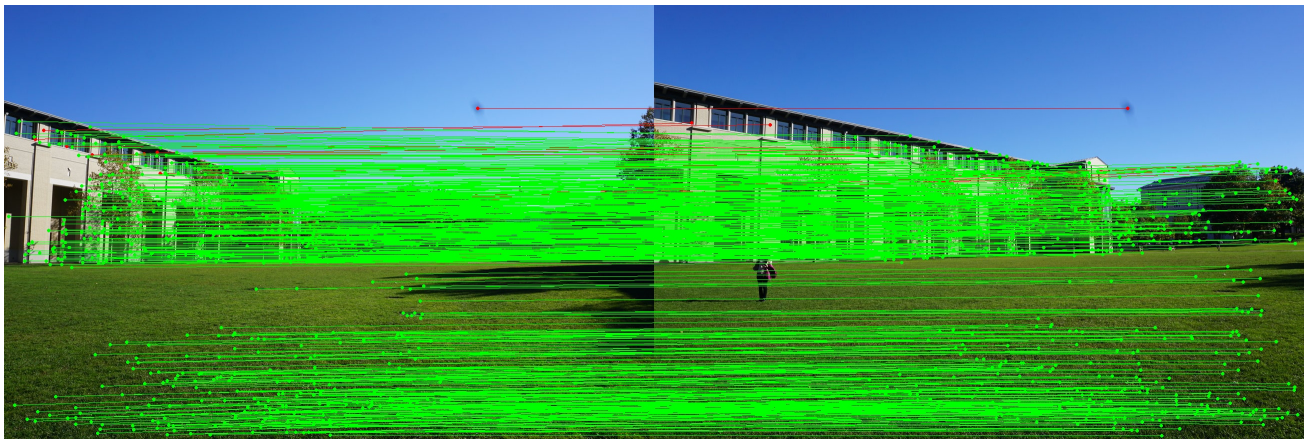

Figure 29: Inliers(green) and outliers(red) between images 4 and 3

Figure 30: Mosaic output without homography refinement

Figure 31: Mosaic output with DogLeg homography refinement

Figure 32: Mosaic output with homography refinement using bundle adjustment

7 Source Code

```
#include <opencv2/core/core.hpp>
#include <opencv2/opencv.hpp>
#include <opencv2/nonfree/nonfree.hpp>
#include <opencv2/highgui/highgui.hpp>
//#include <opencv2/core/eigen.hpp>

#include <iostream>
#include <Eigen/Dense>
#include <vector>
#include <iostream>
#include <math.h>
#include <limits.h>
#include <ctime>
#include <random>
#include "levmar.h"
using namespace Eigen;

using namespace cv;
using namespace std;

struct Parameters
{
  double scale;
  double k;
  double p;
  double wNMS, wSSD, wNCC;
};

//RANSAC parameters
//parameters
double eps = 0.1;
double p = 0.99;
int n = 8;
int delta =40;

//computes homography matrix from a set of corresponding points
Matrix3d ComputeHomography(vector<Point2d>& worldPts, vector<Point2d>& imgPts)
{
  MatrixXd A(2 * worldPts.size(), 9);


  //initialize matrix A
  int count = 0;
  for (auto i = 0; i < worldPts.size(); ++i)
  {

    A.row(count) << 0, 0, 0, -worldPts[i].x, -worldPts[i].y, -1, imgPts[i].y*worldPts[i].x, imgPts[i].y*
        worldPts[i].y, imgPts[i].y;
    ++count;
    A.row(count) << worldPts[i].x, worldPts[i].y, 1, 0, 0, 0, -imgPts[i].x*worldPts[i].x, -imgPts[i].x*
        worldPts[i].y, -imgPts[i].x;
    ++count;
  }


  JacobiSVD<MatrixXd> svd = A.jacobiSvd(ComputeFullU | ComputeFullV);

  VectorXd h = svd.matrixV().col(svd.matrixV().cols()-1);
  MatrixX3d H = Matrix3d::Identity();


  H(0, 0) = h(0);
  H(0, 1) = h(1);
  H(0, 2) = h(2);
  H(1, 0) = h(3);
  H(1, 1) = h(4);
  H(1, 2) = h(5);
  H(2, 0) = h(6);
  H(2, 1) = h(7);
  H(2, 2) = h(8);
```

```
  return H;
}
//normalize HC or convert into physical coordinates
Vector3d normalizeHC(Vector3d& HCin)
{
  Vector3d HCout;
  HCout = HCin / HCin(2);
  return HCout;
}

//dogleg correction
Matrix3d dogleg(Matrix3d& H, vector<Point2d>& imgCorresPts1, vector<Point2d>& imgCorresPts2)
{
  int N = imgCorresPts1.size();
  double uk = 1;
  double rk = 5;
  double rhodl = 1;
  MatrixXd Jacobi = MatrixXd::Zero(2 * N, 9);
  VectorXd Err = VectorXd::Zero(2 * N);
  VectorXd hk = VectorXd::Zero(9);
  hk(0) = H(0, 0);
  hk(1) = H(0, 1);
  hk(2) = H(0, 2);
  hk(3) = H(1, 0);
  hk(4) = H(1, 1);
  hk(5) = H(1, 2);
  hk(6) = H(2, 0);
  hk(7) = H(2, 1);
  hk(8) = H(2, 2);

  Matrix3d Hk = H;
  while (rhodl > 0)
  {
    for (int i = 0; i < N; ++i)
    {
      /*Jacobi =

[ x/(h33 + h31*x + h32*y), y/(h33 + h31*x + h32*y), 1/(h33 + h31*x + h32*y),
0,                         0,                         0,
-(x*(h13 + h11*x + h12*y))/(h33 + h31*x + h32*y)^2,
-(y*(h13 + h11*x + h12*y))/(h33 + h31*x + h32*y)^2, -(h13 + h11*x + h12*y)/(h33 + h31*x + h32*y)^2]
[                          0,                         0,                         0,
x/(h33 + h31*x + h32*y), y/(h33 + h31*x + h32*y), 1/(h33 + h31*x + h32*y),
-(x*(h23 + h21*x + h22*y))/(h33 + h31*x + h32*y)^2, -(y*(h23 + h21*x + h22*y))/(h33 + h31*x + h32*y)^2,
-(h23 + h21*x + h22*y)/(h33 + h31*x + h32*y)^2]

err =

xhat - (h13 + h11*x + h12*y)/(h33 + h31*x + h32*y)
yhat - (h23 + h21*x + h22*y)/(h33 + h31*x + h32*y)

*/
      double denom = (Hk(2, 0)*imgCorresPts1[i].x + Hk(2, 1)*imgCorresPts1[i].y + Hk(2, 2));
      int idx = 2 * i;
      Jacobi(idx, 0) = imgCorresPts1[i].x / denom;
      Jacobi(idx, 1) = imgCorresPts1[i].y / denom;
      Jacobi(idx, 2) = 1 / denom;
      Jacobi(idx, 3) = 0;
      Jacobi(idx, 4) = 0;
      Jacobi(idx, 5) = 0;
      double numCommon = (Hk(0, 2) + Hk(0, 0)*imgCorresPts1[i].x + Hk(0, 1)*imgCorresPts1[i].y);
      Jacobi(idx, 6) = -(imgCorresPts1[i].x * numCommon) / (denom*denom);
      Jacobi(idx, 7) = -(imgCorresPts1[i].y * numCommon) / (denom*denom);
      Jacobi(idx, 8) = -(numCommon) / (denom*denom);

      Err(idx) = imgCorresPts2[i].x - (numCommon / denom);

      ++idx;
      Jacobi(idx, 0) = 0;
      Jacobi(idx, 1) = 0;
      Jacobi(idx, 2) = 0;

      Jacobi(idx, 3) = imgCorresPts1[i].x / denom;
      Jacobi(idx, 4) = imgCorresPts1[i].y / denom;
```

```
    Jacobi(idx, 5) = 1 / denom;

    numCommon = (Hk(1, 2) + Hk(1, 0)*imgCorresPts1[i].x + Hk(1, 1)*imgCorresPts1[i].y);
    Jacobi(idx, 6) = -(imgCorresPts1[i].x * numCommon) / (denom*denom);
    Jacobi(idx, 7) = -(imgCorresPts1[i].y * numCommon) / (denom*denom);
    Jacobi(idx, 8) = -(numCommon) / (denom*denom);

    Err(idx) = imgCorresPts2[i].y - (numCommon / denom);

  }
  MatrixXd JacobiTrans = Jacobi.transpose();
  VectorXd delta_gd = ((JacobiTrans *Err).norm() / (Jacobi*JacobiTrans*Err).norm())*JacobiTrans*Err;
  VectorXd delta_gn = (JacobiTrans*Jacobi + uk * MatrixXd::Identity(9, 9)).inverse() * JacobiTrans*Err;

  VectorXd delta;
  //first case: Gauss-Newton
  if (delta_gn.norm() < rk)
  {
    delta = delta_gn;
  }
  // Second case Gradient descent(GD) within trust region
  //but Gauss-Newton (GN) jump is outside the trust region
  else if ((delta_gd.norm() < rk) && (rk < delta_gn.norm()))
  {
    double aa = (delta_gn - delta_gd).transpose() * (delta_gn - delta_gd);
    double bb = delta_gd.transpose()*(delta_gn - delta_gd);
    double cc = delta_gd.transpose()*delta_gd - (rk*rk);
    double beta = (-bb + sqrt(bb*bb - aa*cc)) / aa;
    delta = delta_gd + beta*(delta_gn - delta_gd);
  }
  //Third case: Both GN and GD outside trust region, fall back to GD until the perimeter
  else
  {
    delta = rk*delta_gd / delta_gd.norm();
  }

  //cost
  double Cpk = Err.transpose()*Err;
  //update hk
  hk = hk + delta;

  VectorXd Err1 = VectorXd::Zero(2 * N);


  Hk(0, 0) = hk(0);
  Hk(0, 1) = hk(1);
  Hk(0, 2) = hk(2);
  Hk(1, 0) = hk(3);
  Hk(1, 1) = hk(4);
  Hk(1, 2) = hk(5);
  Hk(2, 0) = hk(6);
  Hk(2, 1) = hk(7);
  Hk(2, 2) = hk(8);


  for (int i = 0; i < N; ++i)
  {

    int idx = 2 * i;
    Vector3d X = Vector3d(imgCorresPts1[i].x, imgCorresPts1[i].y, 1);
    Vector3d Xp = Hk * X;

    Xp = normalizeHC(Xp);

    Err1(idx) = (X-Xp).x();

    ++idx;

    Err1(idx) = (X - Xp).y();

  }
  //new cost
  double Cpk1 = Err1.transpose()*Err1;
```

```
    double denom = (2.0* delta.transpose() * JacobiTrans*Err); //;
    denom -= ((Jacobi*delta).transpose() * (Jacobi*delta) );
    //calculate rho_DL
      rhodl = (Cpk - Cpk1) / denom;
    //update rk
    if (rhodl < 0.25)
    {
      rk /= 4;
    }
    else if (rhodl <= 0.75)
    {
      rk = rk;
    }
    else
    {
      rk = 2.0 * rk;
    }

  }  //end while

  return Hk;
}

//Mosaicing each image 'img' is written to 'IMG'
void mosaic(Mat& IMG, Mat& img, Matrix3d& H, Vector2d& minCoord, Vector2d& maxCoord)
{
  //scaling along width
  double scaleW = img.cols / (maxCoord.x() - minCoord.x());

  //scaling along height
  double scaleH = img.rows / (maxCoord.y() - minCoord.y());


  //steps along rows or columns
  double stepX = 1.0 / scaleW;
  double stepY = 1.0 / scaleH;

  for (int col = 0; col < IMG.cols; ++col)
  {
    Vector3d world_coord;
    // Set x coordinate
    world_coord(0) = (double)col * stepX + minCoord.x();
    for (int row = 0; row < IMG.rows; ++row)
    {
      // Set y coordinate
      world_coord(1) = (double)row * stepY + minCoord.y();
      world_coord(2) = 1;
      Vector3d img_coord = H.inverse()*world_coord;
      img_coord = normalizeHC(img_coord);

      //if outside the extents ignore the pixel altogether
      if ((int)img_coord.x() < 0 || (int)img_coord.x() > (img.cols - 1) ||
        (int)img_coord.y() < 0 || (int)img_coord.y() > (img.rows - 1))
      {
        continue;
      }

      double dx = img_coord.x() - (int)img_coord.x();
      double dy = img_coord.y() - (int)img_coord.y();


      vector<Vec3b> rgbVec;
      //retrieve color corner img(x,y)
      rgbVec.push_back(img.at<Vec3b>((int)img_coord.y(), (int)img_coord.x()));
      //bilinear if necessary
      if (dx != 0.0 || dy != 0.0)
      {
        //rgb at img(x+1,y)
        if (int(img_coord.x() + 1) < img.cols)
        {
          rgbVec.push_back(img.at<Vec3b>(int(img_coord.y()), int(img_coord.x() + 1)));
        }
        else
        {
```

```
          rgbVec.push_back(Vec3b(0, 0, 0));
        }
        //rgb at img(x,y+1)
        if (int(img_coord.y() + 1) < img.rows)
          rgbVec.push_back(img.at<Vec3b>(int(img_coord.y() + 1), int(img_coord.x())));
        else
          rgbVec.push_back(Vec3b(0, 0, 0));

        //rgb at img(x+1,y+1)
        if ((int(img_coord.y() + 1) < img.rows) && (int(img_coord.x() + 1) < img.cols))
          rgbVec.push_back(img.at<Vec3b>(int(img_coord.y() + 1), int(img_coord.x() + 1)));
        else
          rgbVec.push_back(Vec3b(0, 0, 0));

        IMG.at<Vec3b>(row, col) = rgbVec[0] * (1 - dx)
          * (1 - dy) + rgbVec[1] * dx * (1 - dy) + rgbVec[2] * (1 - dx) * dy
          + rgbVec[3] * dx * dy;
      }
      else
      {
        IMG.at<Vec3b>(row, col) = rgbVec[0];
      }


    }
  }

}


/*
img1 - image 1
img2 - image 2
inId1 - input pt IDs of image 1
inId2 - input pt Ids of image 2
param - struture parameters
outId1 - output ids of image 1
outId2 - corresponding ids of image 2

*/
/*Same as Harris NCC , only instead of neighborhood the decriptors are used*/
void SIFT_NCC(vector<KeyPoint> &keypoints1, Mat &descriptors1, vector<KeyPoint> &keypoints2, Mat &
    descriptors2,
  vector<Point2d> &outPts1, vector<Point2d> &outPts2, Parameters &params)
{
  MatrixXd NCC(keypoints2.size(), keypoints1.size());
  for (int i = 0; i < keypoints2.size(); ++i)
  {
    for (int j = 0; j < keypoints1.size(); ++j)
    {
      Mat feat2 = descriptors2.row(i);
      Mat feat1 = descriptors1.row(j);
      double mean1 = mean(feat1)[0];
      double mean2 = mean(feat2)[0];
      Mat devMat1 = feat1 - mean1;
      Mat devMat2 = feat2 - mean2;
      double numr = sum(devMat1.mul(devMat2))[0];
      double denom = sqrt(sum(devMat1.mul(devMat1))[0] * sum(devMat2.mul(devMat2))[0]);
      NCC(i, j) = numr / denom;
    }
  }
  VectorXd value(NCC.cols());
  VectorXi idx(NCC.cols());
  for (int i = 0; i < NCC.cols(); ++i)
  {
    VectorXd vec = (NCC.col(i).array() - 1).abs();
    value(i) = vec.minCoeff(&(idx[i]));

  }
  double tNCC = params.p * value.mean();

  for (int i = 0; i < keypoints1.size(); ++i)
  {
    if (value[i] <= tNCC)
```

```
    {
      outPts1.push_back(keypoints1[i].pt);
      outPts2.push_back(keypoints2[idx[i]].pt);
    }

  }
}
/*
img1 - image 1
img2 - image 2
inId1 - input pt IDs of image 1
inId2 - input pt Ids of image 2
param - struture parameters
outId1 - output ids of image 1
outId2 - corresponding ids of image 2

*/
/*Same as Harris SSD , only instead of neighborhood the decriptors are used*/
void SIFT_SSD(vector<KeyPoint> &keypoints1, Mat &descriptors1, vector<KeyPoint> &keypoints2, Mat &
    descriptors2,
  vector<Point2d> &outPts1, vector<Point2d> &outPts2)
{
  MatrixXd SSD(keypoints2.size(), keypoints1.size());
  for (int i = 0; i < keypoints2.size(); ++i)
  {
    for (int j = 0; j < keypoints1.size(); ++j)
    {
      Mat feat2 = descriptors2.row(i);
      Mat feat1 = descriptors1.row(j);
      Mat diff = feat1 - feat2;
      Mat Prod = diff.mul(diff);
      SSD(i, j) = sum(Prod)[0];

    }

  }
  double rSSD = 0.6;

  for (int i = 0; i < SSD.cols(); ++i)
  {
    int id;
    double local_minima;
    local_minima = SSD.col(i).minCoeff(&(id));
    SSD(id, i) = SSD.col(i).maxCoeff();

    if ((local_minima / SSD.col(i).minCoeff()) < rSSD)
    {
      outPts1.push_back(keypoints1[i].pt);
      outPts2.push_back(keypoints2[id].pt);
    }


  }
}

//returns keypoints and descriptors for the input image
void SIFTKeyPoints(Mat &img, Parameters &params, vector<KeyPoint> &keypoints, Mat &descriptors)
{
  // initialize detector and extractor
  FeatureDetector* detector;

  /*****************************************************************************/
  /*Using this constructor we can also control the parameters such as sigma for SIFT*/
  /*****************************************************************************/
  detector = new SiftFeatureDetector(
    0, // nFeatures
    4, // nOctaveLayers
    params.k, // contrastThreshold
    10, //edgeThreshold
    params.scale //sigma
    );

  /*default constructor */
  //detector = new SiftFeatureDetector();
```

```
  DescriptorExtractor* extractor;
  extractor = new SiftDescriptorExtractor();
  Mat img_gray;
  cvtColor(img, img_gray, cv::COLOR_BGR2GRAY);


  detector->detect(img_gray, keypoints);
  extractor->compute(img_gray, keypoints, descriptors);

}

//parent method for SIFT detection which takes input images , parameters and returns output image
Mat SIFTDetection(Mat &img1, Mat &img2, Parameters &params, bool bSSD, vector<Point2d>& outPts1, vector<
    Point2d>& outPts2)
{
  //create output image
  Size sz1 = img1.size();
  Size sz2 = img2.size();
  Size sz(sz1.width + sz2.width, sz1.height);
#define PADDING 0
  Mat outImg = Mat::zeros(max(sz1.height, sz2.height), sz1.width + sz2.width + PADDING, CV_8UC3);

#define GAP 0

  Mat left(outImg, Rect(0, 0, sz1.width, sz1.height));
  img1.copyTo(left);
  Mat right(outImg, Rect(sz1.width + GAP, 0, sz2.width, sz2.height));
  img2.copyTo(right);

  //detect SIFT keypoints
  vector<KeyPoint> keypoints1, keypoints2; Mat descriptors1, descriptors2;
  SIFTKeyPoints(img1, params, keypoints1, descriptors1);
  SIFTKeyPoints(img2, params, keypoints2, descriptors2);

  //descriptor matching
  if (bSSD)
  {
    SIFT_SSD(keypoints1, descriptors1, keypoints2, descriptors2, outPts1, outPts2);
  }
  else
  {
    SIFT_NCC(keypoints1, descriptors1, keypoints2, descriptors2, outPts1, outPts2, params);
  }


  //draw
  RNG rng(12345);
  for (int i = 0; i < outPts1.size(); ++i)
  {
    Scalar color = Scalar(rng.uniform(0, 255), rng.uniform(0, 255), rng.uniform(0, 255));
    Point2d from(outPts1[i].x, outPts1[i].y);
    circle(outImg, from, 2, color, 2, 8, 0);
    Point2d to(outPts2[i].x + sz1.width + GAP, outPts2[i].y);
    circle(outImg, to, 2, color, 2, 8, 0);
    line(outImg, from, to, color, 1);
  }

  return outImg;

}

//Extract inlier set
Matrix3d Inliers(vector<Point2d>& inPts1, vector<Point2d>& inPts2, vector<Point2d>& MSet1, vector<Point2d
    >& MSet2,
  vector<Point2d>& inliers1, vector<Point2d>& inliers2)
{
  Matrix3d H = Matrix3d::Identity();
  H = ComputeHomography(MSet1, MSet2);
  for (int i = 0; i < inPts1.size(); ++i)
  {
    Vector3d x(inPts1[i].x, inPts1[i].y, 1);
    Vector3d xp(inPts2[i].x, inPts2[i].y, 1);
    Vector3d Hx = H*x;
    Hx = normalizeHC(Hx);
```

```
    double dist = (Hx - xp).norm();
    if (dist < delta)
    {
      inliers1.push_back(inPts1[i]);
      inliers2.push_back(inPts2[i]);
    }

  }
  return H;
}

//draw inliers and outliers on pair of images
Mat drawRANSACOut(vector<Point2d>& totalPts1, vector<Point2d>& totalPts2,
  vector<Point2d>& inlierPts1, vector<Point2d>& inlierPts2, Mat& img1, Mat& img2 )
{
  Size sz1 = img1.size();
  Size sz2 = img2.size();
  Size sz(sz1.width + sz2.width, sz1.height);
#define PADDING 0
  Mat outImg = Mat::zeros(max(sz1.height, sz2.height), sz1.width + sz2.width + PADDING, CV_8UC3);

#define GAP 0

  Mat left(outImg, Rect(0, 0, sz1.width, sz1.height));
  img1.copyTo(left);
  Mat right(outImg, Rect(sz1.width + GAP, 0, sz2.width, sz2.height));
  img2.copyTo(right);

  for (int i = 0; i < totalPts1.size(); ++i)
  {
    Scalar color = Scalar(0, 0, 255);
    Point2d from(totalPts1[i].x, totalPts1[i].y);
    circle(outImg, from, 2, color, 2, 8, 0);
    Point2d to(totalPts2[i].x + sz1.width + GAP, totalPts2[i].y);
    circle(outImg, to, 2, color, 2, 8, 0);
    line(outImg, from, to, color, 1);

  }

  for (int i = 0; i < inlierPts1.size(); ++i)
  {
    Scalar color = Scalar(0, 255, 0);
    Point2d from(inlierPts1[i].x, inlierPts1[i].y);
    circle(outImg, from, 2, color, 2, 8, 0);
    Point2d to(inlierPts2[i].x + sz1.width + GAP, inlierPts2[i].y);
    circle(outImg, to, 2, color, 2, 8, 0);
    line(outImg, from, to, color, 1);
  }
  return outImg;
}
/*RANSAC main routine*/
Matrix3d runRANSAC(vector<Point2d>& inPts1, vector<Point2d>& inPts2, vector<Point2d>& outPts1, vector<
    Point2d>& outPts2)
{
  cout << "RANSAC " << endl;
  Matrix3d Hout = Matrix3d::Identity();
  int M = inPts1.size()*(1 - eps);

  cout << "M = " << M << endl;

  int N = log(1 - p) / (log(1 - (pow((1 - eps), n))));
  cout << "N = " << N << endl;
  for (int i = 0; i < N; ++i)
  {
    vector<Point2d> Mset1, Mset2;

    std::random_device rd;     // only used once to initialise (seed) engine

    vector<Point2d> t_inPts1 = inPts1;
    vector<Point2d> t_inPts2 = inPts2;
    for (int j = 0; j < n; ++j)
    {

      std::mt19937 rng(rd());    // random-number engine used (Mersenne-Twister in this case)
```

```
      std::uniform_int_distribution<int> uni(0, t_inPts1.size()); // guaranteed unbiased

      int idx = uni(rng) % t_inPts1.size();

      Mset1.push_back(t_inPts1[idx]);
      Mset2.push_back(t_inPts2[idx]);
      //remove the selected points from further consideration
      t_inPts1.erase(t_inPts1.begin() + idx);
      t_inPts2.erase(t_inPts2.begin() + idx);


    }
    vector<Point2d> inliers1, inliers2;

    Matrix3d H = Inliers(inPts1, inPts2, Mset1, Mset2, inliers1, inliers2);

    if (inliers1.size() > M || inliers1.size() > outPts1.size())
    {
      outPts1.clear();
      outPts2.clear();
      outPts1 = inliers1;
      outPts2 = inliers2;
      Hout = H;
    }


  }
  return Hout;
}

/*parent function for doing mosaicing with or without DogLeg refinement
1) SIFT matching
2) RANSAC
3) refinement using DogLeg
4) mosaicing and saving the result

pass true or false as a second argument to set or reset DogLeg method call
*/


void runMosaic(vector<Mat>& images, bool bDogleg)
{
  vector<Point2d> SIFTPts1, SIFTPts2;
  Parameters params;
  params.k = 0.04; //sensitivity for corner response , also used for creating SIFT detector
  params.p = 0.4; //controls thresholding on NCC
  params.scale = 3; //input scale

  params.wNCC = 20; //windows size for NCC NOT used here from harris corner
  params.wNMS = 20; //windows size fo non-maximum suppression NOT used here from harris corner
  params.wSSD = 20; //window size for SSD NOT used here from harris corner

  //pair 0 and 1

  Mat SIFTout = SIFTDetection(images[0], images[1], params, true, SIFTPts1, SIFTPts2);

  //imwrite("./img1/SIFT01.jpg", SIFTout);

  vector<Point2d> inliers1, inliers2;
  Matrix3d H01 = runRANSAC(SIFTPts1, SIFTPts2, inliers1, inliers2);

  cout << ".....done" << endl;
  cout << "# inliers = " << inliers1.size() << endl;

  Mat ransacOut = drawRANSACOut(SIFTPts1, SIFTPts2, inliers1, inliers2, images[0], images[1]);
  //imwrite("./img1/RANSAC01.jpg", ransacOut);

  if (bDogleg)
  {
    H01 = dogleg(H01, inliers1, inliers2);
  }

  //pair 1 and 2
  SIFTPts1.clear();
```

```
SIFTPts2.clear();
SIFTout = SIFTDetection(images[1], images[2], params, true, SIFTPts1, SIFTPts2);

//imwrite("./img1/SIFT12.jpg", SIFTout);

inliers1.clear();
inliers2.clear();
Matrix3d H12 = runRANSAC(SIFTPts1, SIFTPts2, inliers1, inliers2);
cout << "....done" << endl;
cout << "# inliers = " << inliers1.size() << endl;

ransacOut = drawRANSACOut(SIFTPts1, SIFTPts2, inliers1, inliers2, images[1], images[2]);
//imwrite("./img1/RANSAC12.jpg", ransacOut);

if (bDogleg)
{
  H12 = dogleg(H12, inliers1, inliers2);
}
//pair 3 and 2
SIFTPts1.clear();
SIFTPts2.clear();
SIFTout = SIFTDetection(images[3], images[2], params, true, SIFTPts1, SIFTPts2);

//imwrite("./img1/SIFT32.jpg", SIFTout);

inliers1.clear();
inliers2.clear();
Matrix3d H32 = runRANSAC(SIFTPts1, SIFTPts2, inliers1, inliers2);
cout << "....done" << endl;
cout << "# inliers = " << inliers1.size() << endl;

ransacOut = drawRANSACOut(SIFTPts1, SIFTPts2, inliers1, inliers2, images[3], images[2]);
//imwrite("./img1/RANSAC32.jpg", ransacOut);

if (bDogleg)
{
  H32 = dogleg(H32, inliers1, inliers2);
}

//pair 4 and 3
SIFTPts1.clear();
SIFTPts2.clear();
SIFTout = SIFTDetection(images[4], images[3], params, true, SIFTPts1, SIFTPts2);

//imwrite("./img1/SIFT43.jpg", SIFTout);

inliers1.clear();
inliers2.clear();
Matrix3d H43 = runRANSAC(SIFTPts1, SIFTPts2, inliers1, inliers2);
cout << "....done" << endl;
cout << "# inliers = " << inliers1.size() << endl;

ransacOut = drawRANSACOut(SIFTPts1, SIFTPts2, inliers1, inliers2, images[4], images[3]);
//imwrite("./img1/RANSAC43.jpg", ransacOut);

if (bDogleg)
{
  H43 = dogleg(H43, inliers1, inliers2);
}


Matrix3d H02 = H01*H12;
Matrix3d H22 = Matrix3d::Identity();
Matrix3d H42 = H43*H32;

vector<Vector3d> Extents;
Extents.push_back(Vector3d(0, 0, 1));
Extents.push_back(Vector3d(0, images[0].rows - 1, 1));
Extents.push_back(Vector3d(images[0].cols - 1, 0, 1));
Extents.push_back(Vector3d(images[0].cols - 1, images[0].rows, 1));

//compute extents for mosaic output image
Vector2d minCoords(0, 0), maxCoords(images[0].cols, images[0].rows);
for (int i = 0; i < 4; ++i)
```

```
  {
    Vector3d pt = H02*Extents[i];
    pt = normalizeHC(pt);
    minCoords.x() = min(minCoords.x(), pt.x());
    minCoords.y() = min(minCoords.y(), pt.y());
    maxCoords.x() = max(maxCoords.x(), pt.x());
    maxCoords.y() = max(maxCoords.y(), pt.y());


    pt = H12*Extents[i];
    pt = normalizeHC(pt);
    minCoords.x() = min(minCoords.x(), pt.x());
    minCoords.y() = min(minCoords.y(), pt.y());
    maxCoords.x() = max(maxCoords.x(), pt.x());
    maxCoords.y() = max(maxCoords.y(), pt.y());

    pt = H32*Extents[i];
    pt = normalizeHC(pt);
    minCoords.x() = min(minCoords.x(), pt.x());
    minCoords.y() = min(minCoords.y(), pt.y());
    maxCoords.x() = max(maxCoords.x(), pt.x());
    maxCoords.y() = max(maxCoords.y(), pt.y());

    pt = H42*Extents[i];
    pt = normalizeHC(pt);
    minCoords.x() = min(minCoords.x(), pt.x());
    minCoords.y() = min(minCoords.y(), pt.y());
    maxCoords.x() = max(maxCoords.x(), pt.x());
    maxCoords.y() = max(maxCoords.y(), pt.y());
  }
  double width = maxCoords.x() - minCoords.x();
  double height = maxCoords.y() - minCoords.y();

  Mat outImg(images[0].rows, images[0].cols, CV_8UC3, Scalar(0, 0, 0));

  mosaic(outImg, images[0], H02, minCoords, maxCoords);
  mosaic(outImg, images[1], H12, minCoords, maxCoords);
  mosaic(outImg, images[2], H22, minCoords, maxCoords);
  mosaic(outImg, images[3], H32, minCoords, maxCoords);
  mosaic(outImg, images[4], H42, minCoords, maxCoords);

  imwrite("./img1/mosaic.jpg", outImg);

}

/*****************************/
/*BUNDLE ADJUSTMENT */
/*****************************/
/*global variables*/
vector<vector<Point2d> > Pts1;

//h is full array of homography for all image pairs
//idx : index in newX
//h_id : index of homography matrix to be used
void DistFunc(Point2d& X1, double* newX, double *h, int idx, int h_id)
{
  Matrix3d H;

  H(0, 0) = h[h_id+0];
  H(0, 1) = h[h_id + 1];
  H(0, 2) = h[h_id + 2];
  H(1, 0) = h[h_id + 3];
  H(1, 1) = h[h_id + 4];
  H(1, 2) = h[h_id + 5];
  H(2, 0) = h[h_id + 6];
  H(2, 1) = h[h_id + 7];
  H(2, 2) = h[h_id + 8];

  Vector3d ptX(X1.x, X1.y, 1);

  Vector3d ptXp = H*ptX;
  ptXp = normalizeHC(ptXp);

  newX[idx] = ptXp.x();
```

```
  newX[idx + 1] = ptXp.y();



}
/*will be called by dlevmar_dif*/
static void BundleAdjustment(double *h, double *X, int m, int n,void *adata)
{
  int bundleSize = Pts1.size();
  //cout << "bundle size = " << bundleSize << endl;
  int c = 0;
  for (int i = 0; i < bundleSize; ++i)
  {
    int idx = i * 9;

    for (int j = 0; j < Pts1[i].size(); ++j)
    {
      Point2d pt1 = Pts1[i][j];

      DistFunc(pt1,X, h, c,idx);
      c += 2;
    }
  }
}

/*parent function for doing all bundle adjustment operations
1) SIFT matching
2) RANSAC
3) refinement using bundle adjustment
4) mosaicing and saving the result
*/
void runBundle(vector<Mat>& images)
{
  //inputs for bundle adjustments
  //vector < vector<Point2d> > inlierSet1, inlierSet2;
  vector<double> inliersSet;
  unsigned int totalInliers =0;

  vector<Point2d> SIFTPts1, SIFTPts2;
  Parameters params;
  params.k = 0.04; //sensitivity for corner response , also used for creating SIFT detector
  params.p = 0.4; //controls thresholding on NCC
  params.scale = 3; //input scale

  params.wNCC = 20; //windows size for NCC
  params.wNMS = 20; //windows size fo non-maximum suppression
  params.wSSD = 20; //window size for SSD

  //pair 0 and 1
  Mat SIFTout = SIFTDetection(images[0], images[1], params, true, SIFTPts1, SIFTPts2);

  vector<Point2d> inliers1, inliers2;
  Matrix3d H01 = runRANSAC(SIFTPts1, SIFTPts2, inliers1, inliers2);

  cout << ".....done" << endl;
  cout << "# inliers = " << inliers1.size() << endl;

  Mat ransacOut = drawRANSACOut(SIFTPts1, SIFTPts2, inliers1, inliers2, images[0], images[1]);


  Pts1.push_back(inliers1);

  for (int i = 0; i < inliers2.size(); ++i)
  {
    inliersSet.push_back(inliers2[i].x);
    inliersSet.push_back(inliers2[i].y);

  }
  totalInliers += inliers1.size();

  //pair 1 and 2
  SIFTPts1.clear();
  SIFTPts2.clear();
  SIFTout = SIFTDetection(images[1], images[2], params, true, SIFTPts1, SIFTPts2);
```

```
inliers1.clear();
inliers2.clear();
Matrix3d H12 = runRANSAC(SIFTPts1, SIFTPts2, inliers1, inliers2);
cout << ".....done" << endl;
cout << "# inliers = " << inliers1.size() << endl;

ransacOut = drawRANSACOut(SIFTPts1, SIFTPts2, inliers1, inliers2, images[1], images[2]);


Pts1.push_back(inliers1);

for (int i = 0; i < inliers2.size(); ++i)
{
  inliersSet.push_back(inliers2[i].x);
  inliersSet.push_back(inliers2[i].y);

}
totalInliers += inliers1.size();

//pair 3 and 2
SIFTPts1.clear();
SIFTPts2.clear();
SIFTout = SIFTDetection(images[3], images[2], params, true, SIFTPts1, SIFTPts2);

inliers1.clear();
inliers2.clear();
Matrix3d H32 = runRANSAC(SIFTPts1, SIFTPts2, inliers1, inliers2);
cout << ".....done" << endl;
cout << "# inliers = " << inliers1.size() << endl;


ransacOut = drawRANSACOut(SIFTPts1, SIFTPts2, inliers1, inliers2, images[3], images[2]);

Pts1.push_back(inliers1);


for (int i = 0; i < inliers2.size(); ++i)
{
  inliersSet.push_back(inliers2[i].x);
  inliersSet.push_back(inliers2[i].y);

}
totalInliers += inliers1.size();

//pair 4 and 3
SIFTPts1.clear();
SIFTPts2.clear();
SIFTout = SIFTDetection(images[4], images[3], params, true, SIFTPts1, SIFTPts2);

inliers1.clear();
inliers2.clear();
Matrix3d H43 = runRANSAC(SIFTPts1, SIFTPts2, inliers1, inliers2);
cout << ".....done" << endl;
cout << "# inliers = " << inliers1.size() << endl;

ransacOut = drawRANSACOut(SIFTPts1, SIFTPts2, inliers1, inliers2, images[4], images[3]);


Pts1.push_back(inliers1);

for (int i = 0; i < inliers2.size(); ++i)
{
  inliersSet.push_back(inliers2[i].x);
  inliersSet.push_back(inliers2[i].y);

}
totalInliers += inliers1.size();

vector<double> h (9 * (Pts1.size()));
h[0] = H01(0, 0);
h[1] = H01(0, 1);
h[2] = H01(0, 2);
```

```
h[3] = H01(1, 0);
h[4] = H01(1, 1);
h[5] = H01(1, 2);
h[6] = H01(2, 0);
h[7] = H01(2, 1);
h[8] = H01(2, 2);

h[9] = H12(0, 0);
h[10] = H12(0, 1);
h[11] = H12(0, 2);
h[12] = H12(1, 0);
h[13] = H12(1, 1);
h[14] = H12(1, 2);
h[15] = H12(2, 0);
h[16] = H12(2, 1);
h[17] = H12(2, 2);

h[18] = H32(0, 0);
h[19] = H32(0, 1);
h[20] = H32(0, 2);
h[21] = H32(1, 0);
h[22] = H32(1, 1);
h[23] = H32(1, 2);
h[24] = H32(2, 0);
h[25] = H32(2, 1);
h[26] = H32(2, 2);

h[27] = H43(0, 0);
h[28] = H43(0, 1);
h[29] = H43(0, 2);
h[30] = H43(1, 0);
h[31] = H43(1, 1);
h[32] = H43(1, 2);
h[33] = H43(2, 0);
h[34] = H43(2, 1);
h[35] = H43(2, 2);

void(*err)(double *p, double *hx, int m, int n, void *adata);

err = BundleAdjustment;
double opts[LM_OPTS_SZ], info[LM_INFO_SZ];
opts[0] = LM_INIT_MU; opts[1] = 1E-20; opts[2] = 1E-20; opts[3] = 1E-25;
opts[4] = LM_DIFF_DELTA; // relevant only if the finite difference Jacobian version is used
int ret = dlevmar_dif(err, &(h[0]), &(inliersSet[0]), 36, 2* totalInliers, 1000, opts, info, NULL, NULL,
     NULL); // no Jacobian

printf("Levenberg-Marquardt returned %d in %g iter, reason %g\nSolution: ", ret, info[5], info[6]);
  printf("\n\nMinimization info:\n");
for (int i = 0; i<LM_INFO_SZ; ++i)
  printf("%g ", info[i]);
printf("\n");


//modify matrices
cout << "H01 BEFORE" << endl;
cout << H01 << endl;

H01(0, 0) = h[0];
H01(0, 1) = h[1];
H01(0, 2) = h[2];
H01(1, 0) = h[3];
H01(1, 1) = h[4];
H01(1, 2) = h[5];
H01(2, 0) = h[6];
H01(2, 1) = h[7];
H01(2, 2) = h[8];
cout << "H01 AFTER" << endl;
cout << H01 << endl;

cout << "H12 BEFORE" << endl;
cout << H12 << endl;
H12(0, 0) = h[9];
H12(0, 1) = h[10];
H12(0, 2) = h[11];
```

```
H12(1, 0) = h[12];
H12(1, 1) = h[13];
H12(1, 2) = h[14];
H12(2, 0) = h[15];
H12(2, 1) = h[16];
H12(2, 2) = h[17];
cout << "H12 AFTER" << endl;
cout << H12 << endl;

cout << "H32 BEFORE" << endl;
cout << H32 << endl;
H32(0, 0) = h[18];
H32(0, 1) = h[19];
H32(0, 2) = h[20];
H32(1, 0) = h[21];
H32(1, 1) = h[22];
H32(1, 2) = h[23];
H32(2, 0) = h[24];
H32(2, 1) = h[25];
H32(2, 2) = h[26];
cout << "H32 AFTER" << endl;
cout << H32 << endl;

cout << "H43 BEFORE" << endl;
cout << H43 << endl;
H43(0, 0) = h[27];
H43(0, 1) = h[28];
H43(0, 2) = h[29];
H43(1, 0) = h[30];
H43(1, 1) = h[31];
H43(1, 2) = h[32];
H43(2, 0) = h[33];
H43(2, 1) = h[34];
H43(2, 2) = h[35];

cout << "H43 AFTER" << endl;
cout << H43 << endl;

Matrix3d H02 = H01*H12;
Matrix3d H22 = Matrix3d::Identity();
Matrix3d H42 = H43*H32;

vector<Vector3d> Extents;
Extents.push_back(Vector3d(0, 0, 1));
Extents.push_back(Vector3d(0, images[0].rows - 1, 1));
Extents.push_back(Vector3d(images[0].cols - 1, 0, 1));
Extents.push_back(Vector3d(images[0].cols - 1, images[0].rows, 1));

//compute extents for mosaic output image
Vector2d minCoords(0, 0), maxCoords(images[0].cols, images[0].rows);
for (int i = 0; i < 4; ++i)
{
  Vector3d pt = H02*Extents[i];
  pt = normalizeHC(pt);
  minCoords.x() = min(minCoords.x(), pt.x());
  minCoords.y() = min(minCoords.y(), pt.y());
  maxCoords.x() = max(maxCoords.x(), pt.x());
  maxCoords.y() = max(maxCoords.y(), pt.y());


  pt = H12*Extents[i];
  pt = normalizeHC(pt);
  minCoords.x() = min(minCoords.x(), pt.x());
  minCoords.y() = min(minCoords.y(), pt.y());
  maxCoords.x() = max(maxCoords.x(), pt.x());
  maxCoords.y() = max(maxCoords.y(), pt.y());


  pt = H32*Extents[i];
  pt = normalizeHC(pt);
  minCoords.x() = min(minCoords.x(), pt.x());
  minCoords.y() = min(minCoords.y(), pt.y());
  maxCoords.x() = max(maxCoords.x(), pt.x());
  maxCoords.y() = max(maxCoords.y(), pt.y());
```

```
    pt = H42*Extents[i];
    pt = normalizeHC(pt);
    minCoords.x() = min(minCoords.x(), pt.x());
    minCoords.y() = min(minCoords.y(), pt.y());
    maxCoords.x() = max(maxCoords.x(), pt.x());
    maxCoords.y() = max(maxCoords.y(), pt.y());
  }
  double width = maxCoords.x() - minCoords.x();
  double height = maxCoords.y() - minCoords.y();

  Mat outImg(images[0].rows, images[0].cols, CV_8UC3, Scalar(0, 0, 0));

  mosaic(outImg, images[0], H02, minCoords, maxCoords);
  mosaic(outImg, images[1], H12, minCoords, maxCoords);
  mosaic(outImg, images[2], H22, minCoords, maxCoords);
  mosaic(outImg, images[3], H32, minCoords, maxCoords);
  mosaic(outImg, images[4], H42, minCoords, maxCoords);

  imwrite("./img1/mosaicLM.jpg", outImg);

}
/*******************************/
int main()
{
  Mat img;

  vector<Mat> images;
  //read images
  img = imread("./img1/1.jpg", CV_LOAD_IMAGE_COLOR);   // Read the file


  if (!img.data)                              // Check for invalid input
  {
    cout << "Could not open or find the image" << std::endl;
    return -1;
  }

  images.push_back(img);
  //read images
  img = imread("./img1/2.jpg", CV_LOAD_IMAGE_COLOR);   // Read the file


  if (!img.data)                              // Check for invalid input
  {
    cout << "Could not open or find the image" << std::endl;
    return -1;
  }

  images.push_back(img);

  img = imread("./img1/3.jpg", CV_LOAD_IMAGE_COLOR);   // Read the file


  if (!img.data)                              // Check for invalid input
  {
    cout << "Could not open or find the image" << std::endl;
    return -1;
  }

  images.push_back(img);

  img = imread("./img1/4.jpg", CV_LOAD_IMAGE_COLOR);   // Read the file


  if (!img.data)                              // Check for invalid input
  {
    cout << "Could not open or find the image" << std::endl;
    return -1;
  }

  images.push_back(img);

  img = imread("./img1/5.jpg", CV_LOAD_IMAGE_COLOR);   // Read the file
```

```
  if (!img.data)                              // Check for invalid input
  {
    cout << "Could not open or find the image" << std::endl;
    return -1;
  }

  images.push_back(img);

  runMosaic(images, false);

  runBundle(images);
  return 0;

}
```