ECE661 - Assignment 5

Caluadewa Dharshaka Tharindu Mathew (mathewc@purdue.edu)

October 13, 2016

1 SIFT Matching

Sift was used to recognize interest points in image pairs. The descriptors was then compared with the Euclidean distance between the descriptor vectors. A threshold, T_{SIFT} and the minimum distance d_{min} was used to supress matches, by choosing only matching points that above $T_{SIFT} \cdot d_{min}$.

2 RANSAC

We need an algorithm that is specialized to removing outliers that were not removed by thresholding the SIFT matches. As the linear least squares estimation, that we use to estimate a homography H, does not tolerate outliers, estimating it while outliers are present will produce an incorrect estimate of H.

1. Parameters needed for RANSAC
   - the decision theshold δ usually set as 3σ
   - the number of trials N
   - the minimum size of the inlier set for it to be acceptable M
   - The percentage of outliers ϵ, roughly estimated to be 0.1
   - The probability that one of the N trials is free of outliers p, chosen to be 0.99
   - The number of matches chosen at random for a trial n, in this case chosen to be 6

     N can be calculated to be $N = \dfrac{ln(1-p)}{ln[1-(1-\epsilon)^n]}$.

     M can be calculated to be $M = (1-\epsilon) \cdot n_{total}$, where n_{total} is the total number of matches. This places an additional constraint on the minimum number of required matches.
2. For each trial in RANSAC, we randomly pick n number of correspondences, and forming a linear least squares matrix to solve for H. The homogeneous equation was formed for n correspondence pairs (X', X_i') as follows:

$$\begin{bmatrix} 0 & 0 & 0 & -w_1'x_1 & -w_1'y_1 & -w_1'w_1 & y_1'x_1 & y_1'y_1 & y_1'w_1 \\ w_1'x_1 & w_1'y_1 & w_1'w & 0 & 0 & 0 & -x_1'x_1 & -x_1'y_1 & -x_1'w_1 \\ & & \vdots & & \vdots & & & & \\ 0 & 0 & 0 & -w_n'x_n & -w_n'y_n & -w_n'w_n & y_n'x_n & y_n'y_n & y_n'w_n \\ w_n'x_n & w_n'y_n & w_n'w_n & 0 & 0 & 0 & -x_n'x_n & -x_n'y_n & -x_n'w_n \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix}$$

In this case $w_i = w_i' = 1$. We solve this by SVD. The last column of V in the decomposition that corresponds to the smallest eigenvalue is the solution to this equation.

3. After solving for H, we then check the rest of the points for outliers, based on the δ threshold chosen above. Those where the calculated point and the actual points norm is greater than δ is considered an outlier. This way we form an inlier set.

4. After N trials, we choose the solution that resulted in the largest inlier set that is greater than M. Then, based on this whole set of inliers, we again refine the estimate of H.

3 DogLeg

Since the homography operation is a non-linear combination of points, the linear least-squares will not give the best solution. The solution given by linear least-squares can be further refined using a non-linear least squares method such as DogLeg.

DogLeg is based on a clever combination of the Gradient Descent and Gauss-Newton methods.

$$\delta_{p,GD} = \frac{||J_f^T \epsilon(p_k)||}{||J_f J_f^T \epsilon(p_k)||} J_f^T \epsilon(p_k), \delta_{p,GN} = \frac{1}{J_f J_f^T + \mu^k I} J_f^T \epsilon(p_k)$$

$\epsilon(p_k) = ||X'_{actual} - F(P_k)||$ is the overall geometric error. $P(k)$ becomes our homography estimation, $\begin{bmatrix} h_{11} & h_{12} & h_{13} & h_{21} & h_{22} & h_{23} & h_{31} & h_{32} & h_{33} \end{bmatrix}$. $F(p_k)$ is the vector of all i points, where $F^(i) = \begin{bmatrix} X'^(i) \\ Y'^(i)] \end{bmatrix} = \begin{bmatrix} \frac{h_{11}x_i + h_{12}y_i + h_{13}}{h_{31}x_i + h_{32}y_i + h_{33}} \\ \frac{h_{21}x_i + h_{22}y_i + h_{23}}{h_{31}x_i + h_{32}y_i + h_{33}} \end{bmatrix}$.

Then, we are able to calculate the Jacobian $\frac{\partial F(p_k)}{\partial p(k)}$.

Based on a defined trust region r_k, every iteration p_k is updated as follows:

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

To obtain β, we solve $||\delta_{p,GD} + B(\delta_{p,GD} - \delta_{p,GD}||^2$. To obtain r_{k+1}, we first calculate:

$$\rho^{DL} = \frac{C(p_k) - C(p_{k+1})}{2\delta_p^T J_f^T \epsilon(p_k) - \delta_p J_f^T J_f \delta_p}$$

Here, $C(p) = \epsilon^T(p)\epsilon(p)$. Then,

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

To obtain $\mu k + 1$, we calculate ρ^{LM}

$$\rho^{DM} = \frac{C(p_k) - C(p_{k+1})}{\delta_p^T J_f^T \epsilon(p_k) - \delta_p \mu_k I \delta_p}$$

Then,

$$\mu_{k+1} = \mu_k \cdot max\frac{1}{3}, 1 - (2\rho^{LM} - 1)^3$$

Initial values for p_k is from the estimate obtained from RANSAC¿. Initial value for $\mu_0 = \tau \cdot maxdiag(J_f^T J_f)$, and $r_0 \approx 0.5$. Here $0 < \tau \leq 1$.

4 Image Mosaicing

4 Homographies are calculated between 5 Image Pairs using the method above described from 1-4. Let the Homography between image i and image j be H_{ij}. Then, we have homographies, H_{12}, H_{23}, H_{34}, H_{45}. Now, to get all images respective to the center image (which is image 3 in this case), we calculate,

$$H_{13} = H_{12}H_{23}, H_{43} = H_{34}^{-1}, H_{53} = H_{45}^{-1}H_{34}^{-1}$$

Then, by applying H_{i3} to image i, where $i \neq 3$, we get all the points in the coordinates of the center frame.

5 Bundle Adjusting

For bundle adjust, the SIFT correspondence was taken from every i^{th} image to every j^{th} image. Let us continue assuming the number of images is 5. The homographies H_{12}, H_{23}, H_{34}, H_{45} were used as the parameters to minimize, i.e. $9 \times 5 = 45$ parameters. The residual error was calculated for each i^{th} image to every j^{th} image. This makes 10 unique cases for 5 images. So, the total number of error components will be the total number of matches (M) across all 10 unique cases. Then, this error was minimized using the Levenberg-Marquardt algorithm (through a downloaded package).

6 Parameters

The parameters defined in the above sections were set to the values shown in the table:

Parameter	Value
T_{SIFT}	10
$nKeyPoints_{SIFT}$	300
ϵ	0.1
p	0.99
n	6
δ	3
M	$0.9n_{total}$
N	6 (calculated)
τ	0.001
r_k	0.5

Here, $nKeyPoints_{SIFT}$ is the maximum keypoints SIFT returns per image.

7 Results

7.1 Image Mosaics

It can be seen specially at the left hand side bottle, the edges of the desk, and the edges of the whiteboard, that they line up better at when DogLeg is used. Since these images taken from a short distance, and apparent rotation between each image is high, the homographies are not able to make matches at the corners. The clock at the upper left corner shows this well.

In the bundle adjusted case some of the edges that seemed broken even with DogLeg, seem completely fixed. Still some of the edges are broken, showing the difficulty of the images chosen.

The 2nd image set of the night scene gave similar results for all 3 cases. This seems to be because of the low correspondence points due to low light and the dynamic scene and lights.

Note: Difficult images were chosen to see the effect of bundle adjusting.

7.1.1 Without DogLeg

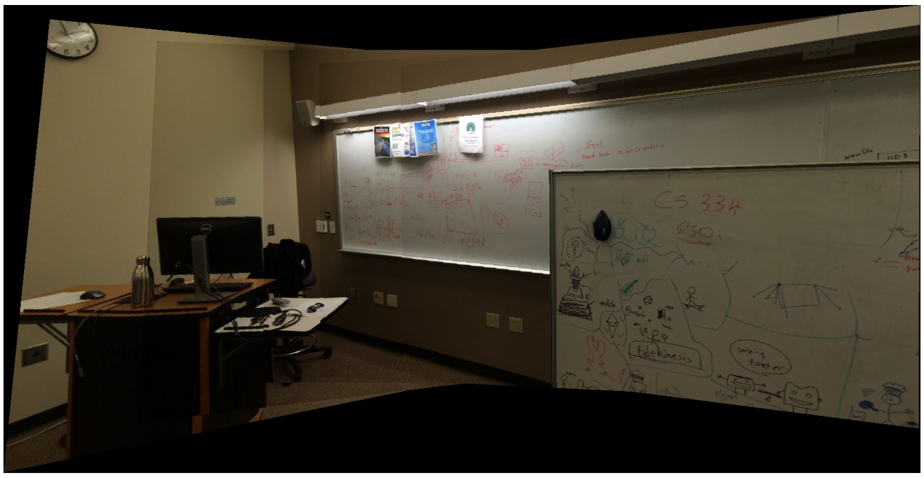

Figure 1: Image Mosaic - Without DogLeg

7.1.2 with DogLeg

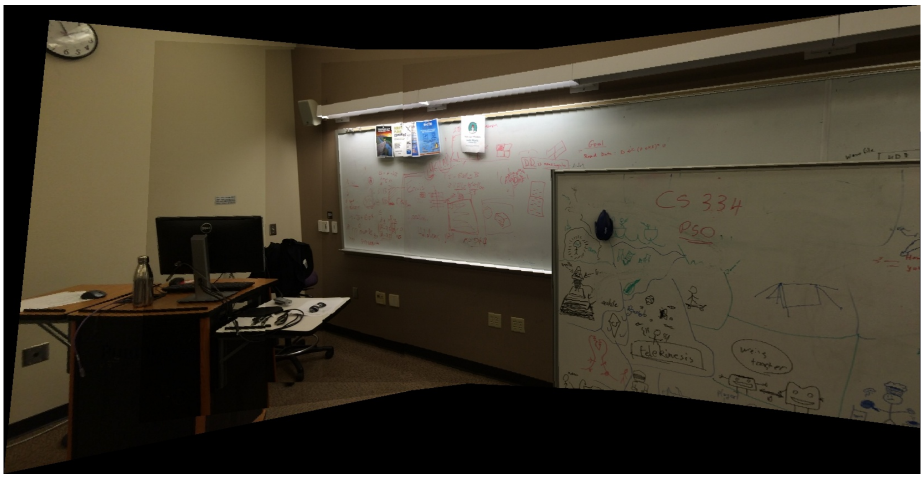

Figure 2: Image Mosaic - With DogLeg

7.1.3 Bundle Adjusted

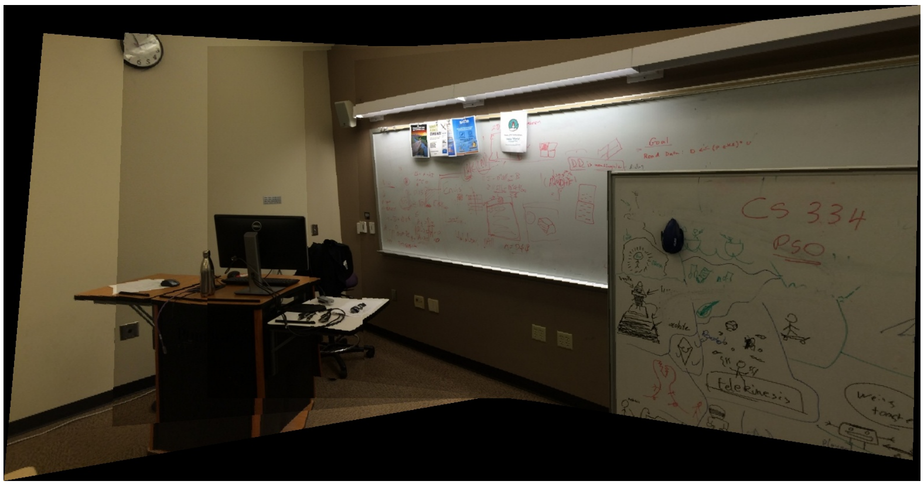

Figure 3: Image Mosaic - Bundle Adjusted

7.1.4 Without DogLeg

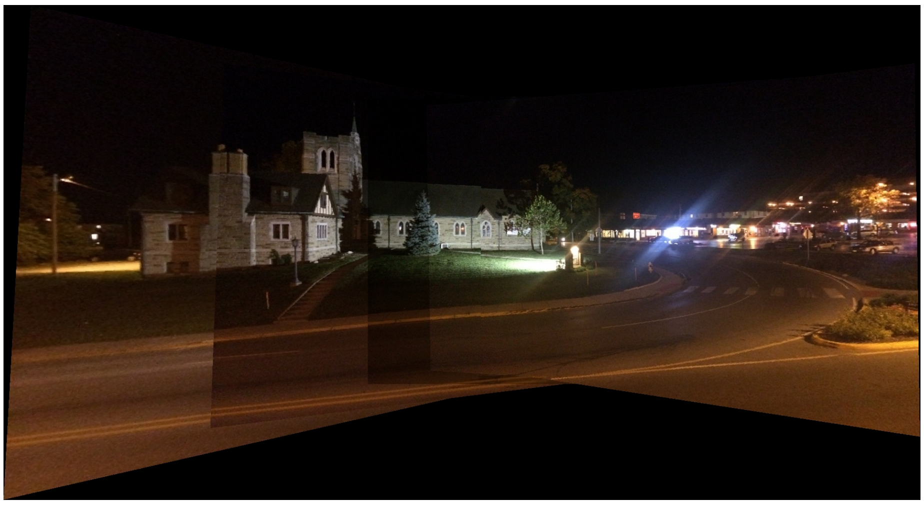

Figure 4: Image Mosaic - Without DogLeg

7.1.5 with DogLeg

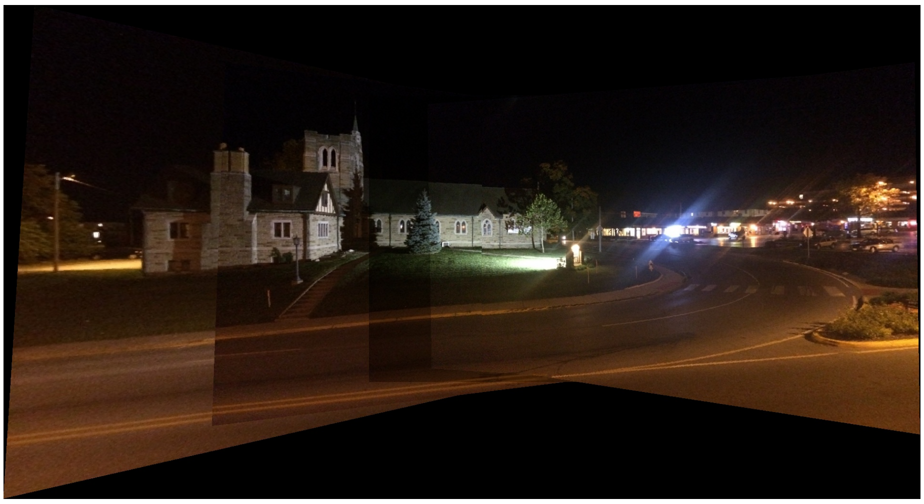

Figure 5: Image Mosaic - With DogLeg

7.1.6 Bundle Adjusted

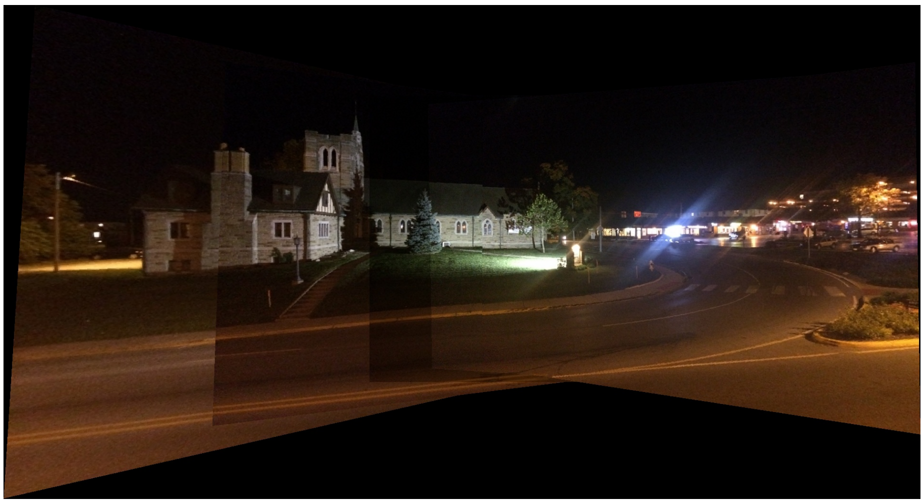

Figure 6: Image Mosaic - Bundle Adjusted

7.2 SIFT Correspondences

The SIFT correspondences after threhsolding are shown with lines. The unchosen keypoints are present without any lines.

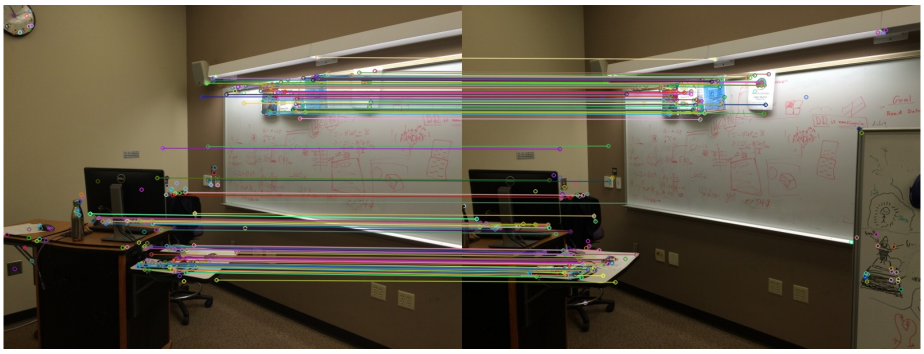

Figure 7: SIFT correspondence - image 1 and 2

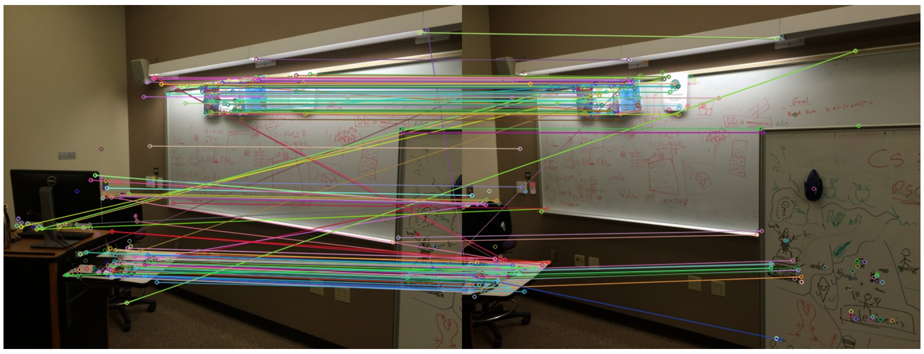

Figure 8: SIFT correspondence - image 2 and 3

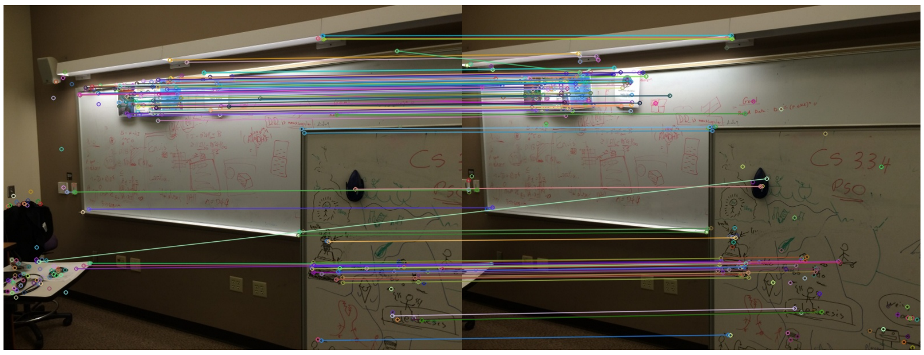

Figure 9: SIFT correspondence - image 3 and 4

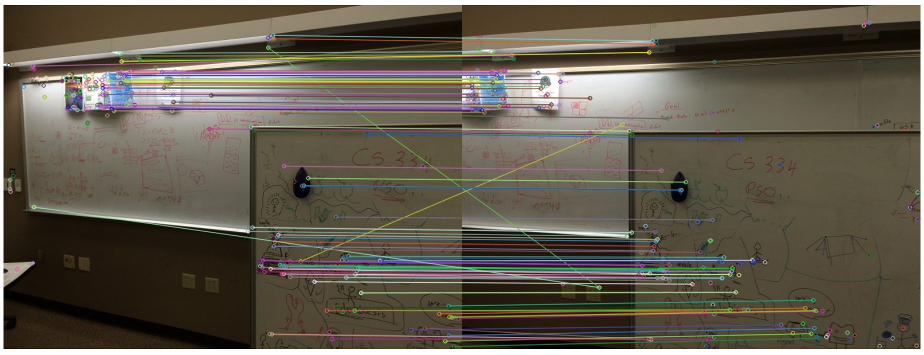

Figure 10: SIFT correspondence - image 4 and 5

7.3 Inliers and Outliers

Inliers from RANSAC are shown with blue lines and points. Red points are outliers.

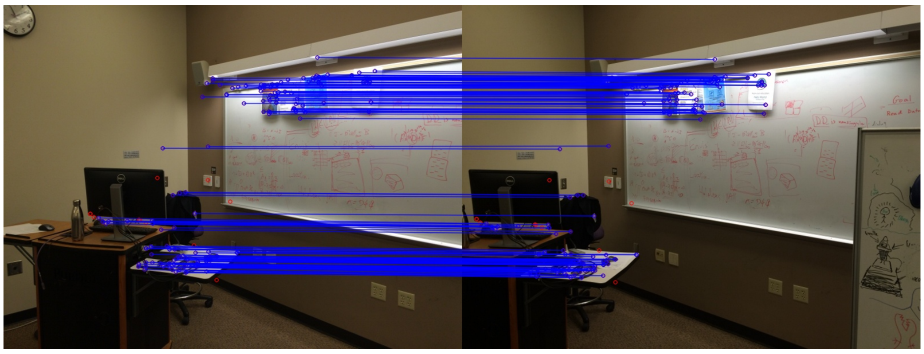
Figure 11: Inliers/Outliers from RANSAC - image 1 and 2

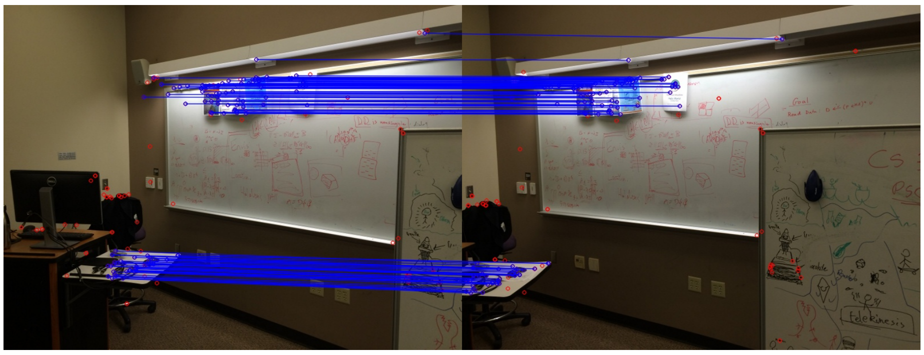
Figure 12: Inliers/Outliers from RANSAC - image 2 and 3

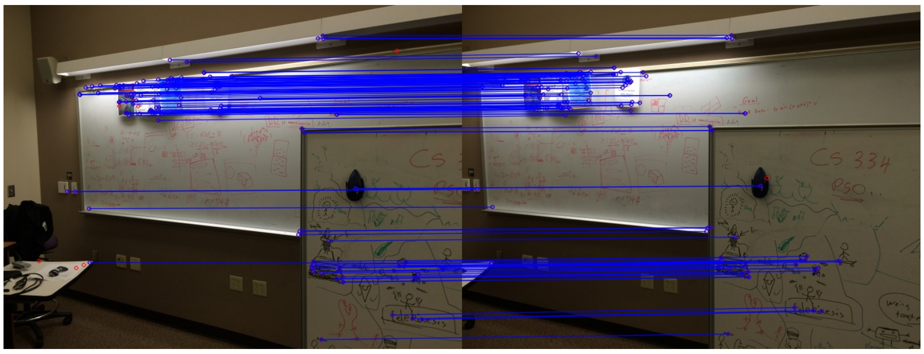
Figure 13: Inliers/Outliers from RANSAC - image 3 and 4

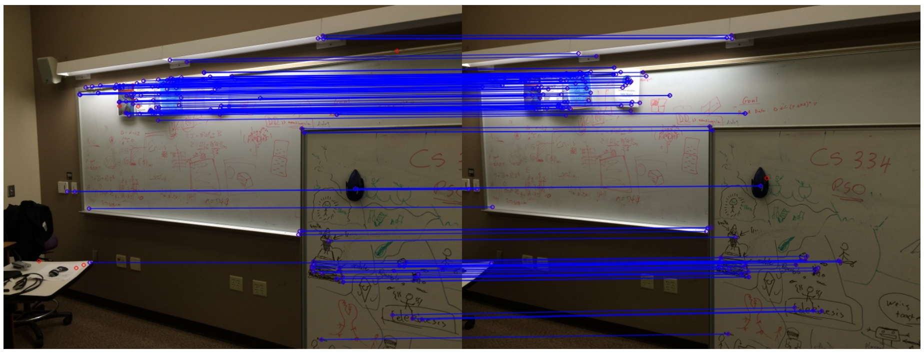
Figure 14: Inliers/Outliers from RANSAC - image 3 and 4

8 Source Code

Panorama Stitching

```
#include "opencv2/core/core.hpp"
#include "opencv2/features2d/features2d.hpp"

struct MatchingFeaturePt {
        double score;
        int index_1;
        int index_2;
};

class EuclideanDistance {
public:
        static void match_by_euclidean_distance(const std::vector<cv::KeyPoint>&
             keypoints_1, const cv::Mat& descriptor_1,
                const std::vector<cv::KeyPoint>& keypoints_2, const cv::Mat&
                     descriptor_2, double threshold, std::vector<cv::DMatch>&
                     matches);

};

class LinearLeastSquares {
public:
        static void linear_least_squares_for_homography(const cv::Mat& A, cv::Mat& H
             );
};

class Ransac {
public:
        static void ransac_for_homography(const double
             epsilon_percentage_of_outliers, const int total_correspondences,
                const std::vector<cv::KeyPoint>& key_points_1, const std::vector<cv
                     ::KeyPoint>& key_points_2,
        const std::vector<cv::DMatch >& matches, cv::Mat& final_H, std::vector<cv::
             Vec3d>& final_img1_pts_vec3d, std::vector<cv::Vec3d>&
             final_img2_pts_vec3d,
        std::vector<cv::KeyPoint>& max_inliner_keypoins_1,
        std::vector<cv::KeyPoint>& max_inliner_keypoins_2,
        std::vector<cv::DMatch>& max_inliner_matches
                );

};

class Homography {
private:
        cv::Mat H;
        void construct_first_row(const cv::Vec3d& img1_pt, const cv::Vec3d& img2_pt,
              int pts_i, cv::Mat& A);
        void construct_second_row(const cv::Vec3d& img1_pt, const cv::Vec3d& img2_pt
             , int pts_i, cv::Mat& A);

public:
        Homography(const std::vector<cv::Vec3d>& img_vec_array_1, const std::vector<
             cv::Vec3d>& img_vec_array_2);
        Homography(const std::vector<cv::KeyPoint>& key_points_1, const std::vector<
             cv::KeyPoint>& key_points_2, const std::vector<cv::DMatch>&
             selected_matches);
        void construct_H(const std::vector<cv::Vec3d>& img_vec_array_1, const std::
             vector<cv::Vec3d>& img_vec_array_2);
        cv::Mat get_homography() const;

};

class DogLeg {

        static void construct_first_row_of_jacobian(const cv::Vec3d& img1_pt, const
             cv::Vec3d& calculated_pt, int pts_i, cv::Mat& J);
        static void construct_second_row_of_jacobian(const cv::Vec3d& img1_pt, const
              cv::Vec3d& calculated_pt, int pts_i, cv::Mat& J);
        static void compute_jacobian(std::vector<cv::Vec3d>& img1_pts, std::vector<
             cv::Vec3d>& calculated_pts, cv::Mat& jacobian);
        static void compute_error_vector(std::vector<cv::Vec3d>& ground_truth, std::
             vector<cv::Vec3d>& calculated_pts, cv::Mat& error_vector);
        static void compute_gradient_descent_increment(const cv::Mat& J, const cv::
             Mat& error_vector, cv::Mat& gradient_descent_increment);
        static void compute_gauss_newton_increment(const cv::Mat& J, const cv::Mat&
             error_vector, const cv::Mat& identity, double mu_k, cv::Mat&
             gauss_newton_increment);
        static double compute_beta(const cv::Mat& gauss_newton_increment, const cv::
             Mat& gradient_descent_increment, double r_k);
        static void convert_to_H_k(const cv::Mat& P_k, cv::Mat& H_k);
        static void convert_to_P_k(const cv::Mat& H_k, cv::Mat& P_k);
        static double initialize_mu(const cv::Mat& J, double tau);
        static void calculate_result_pts(const std::vector<cv::Vec3d>& img1_pts,
             const cv::Mat& H_k, std::vector<cv::Vec3d>& calculated_pts);
public:
        static void dogleg_for_non_linear_least_squares_optimization(const cv::Mat&
             H, std::vector<cv::Vec3d>& img1_pts, std::vector<cv::Vec3d>& img2_pts,
             cv::Mat& output_H);

};

class ImgUtils {
public:
        static cv::Point2d apply_custom_homography(const cv::Mat matrix, const cv::
             Point2d& src_pt);
        static void find_bounding_box(const std::vector<cv::Point2d>& points, double
             & min_x, double& max_x, double& min_y, double& max_y);
        static void apply_distortion_correction(cv::Mat& homography_matrix, const cv
             ::Mat& img, cv::Mat& world_img, const std::string& final_image_name);
        static void combine_transformed_imgs(const cv::Mat& img_1, const cv::Mat&
             img_2, cv::Mat& H, cv::Mat& transformed_img);
        static void get_bounding_box(const std::vector<cv::Mat>& imgs, const std::
             vector<cv::Mat>& Hs, double& minx, double& maxx, double& miny, double&
             maxy);
        static void combine_imgs_to_middle_img(const std::vector<cv::Mat>& imgs,
             const std::vector<cv::Mat>& Hs, cv::Mat& transformed_img);
};
```

```
#include "panorama.h"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/nonfree/features2d.hpp"
#include <iostream>
#include <chrono>
#include <cmath>
#include <random>
#include <set>



void EuclideanDistance::match_by_euclidean_distance(const std::vector<cv::KeyPoint>&
        keypoints_1, const cv::Mat& descriptor_1,
          const std::vector<cv::KeyPoint>& keypoints_2, const cv::Mat& descriptor_2,
               double threshold, std::vector<cv::DMatch>& matches) {

        std::vector<MatchingFeaturePt> matching_pts;
        double max_ssd = -DBL_MAX;
        double min_ssd = DBL_MAX;

        //std::cout << descriptor_1.row(0) << " " << descriptor_1.rows << " " <<
             descriptor_1.cols << "\n";

        for (auto i1 = 0; i1 < keypoints_1.size(); ++i1) {

                cv::Mat keypt_1_descriptor = descriptor_1.row(i1);

                for (auto i2 = 0; i2 < keypoints_2.size(); ++i2) {

                        double ssd = 0.0;

                        cv::Mat keypt_2_descriptor = descriptor_2.row(i2);

                        for (int k = 0; k < keypt_1_descriptor.cols; ++k) {
                                auto des_1 = keypt_1_descriptor.at<float>(0, k);
                                auto des_2 = keypt_2_descriptor.at<float>(0, k);
                                ssd += std::pow((des_1 - des_2), 2);
                        }

                        ssd = std::sqrt(ssd);

                        max_ssd = std::max(max_ssd, ssd);
                        min_ssd = std::min(min_ssd, ssd);

                        MatchingFeaturePt matching_pt;
                        matching_pt.score = ssd;
                        matching_pt.index_1 = i1;
                        matching_pt.index_2 = i2;

                        matching_pts.push_back(matching_pt);
                }
        }


        // sort values according to SSD for each point
        std::sort(matching_pts.begin(), matching_pts.end(), [&] (const
             MatchingFeaturePt& lhs, const MatchingFeaturePt& rhs)
        {
                if (lhs.index_1 == rhs.index_1) {
                        return lhs.score < rhs.score;
                }
                return lhs.index_1 < rhs.index_1;
        });

        std::vector<MatchingFeaturePt> filtered_matching_pts;

        int prev_index = -1;


        std::cout << "max␣ssd␣:␣" << max_ssd << "␣min␣ssd:␣" << min_ssd << "␣
             threshold␣:␣" << threshold << "\n";

        // keep only points that are T * SSD_(max)
        for (auto i = 0; i < matching_pts.size(); ++i) {
                auto matching_pt = matching_pts[i];
                if (prev_index == matching_pt.index_1) {
                } else {
                        if (matching_pt.score < threshold * (min_ssd)) {
                                cv::DMatch match;
                                match.queryIdx = matching_pt.index_1;
                                match.trainIdx = matching_pt.index_2;
                                matches.push_back(match);
                        }
                }
                prev_index = matching_pt.index_1;
        }
}

void LinearLeastSquares::linear_least_squares_for_homography(const cv::Mat& A, cv::
     Mat& H) {

        cv::Mat U, D, Vt;
        cv::SVD::compute(A, D, U, Vt, cv::SVD::FULL_UV);

        //std::cout << Vt << "\n";

        cv::Mat last_col_V = Vt.t().col(Vt.cols - 1);

        H = last_col_V;

        //std::cout << H << "\n";

}

double round(double number)
{
    return number < 0.0 ? ceil(number - 0.5) : floor(number + 0.5);
}

void Ransac::ransac_for_homography(const double epsilon_percentage_of_outliers,
     const int n_total_correspondences,
        const std::vector<cv::KeyPoint>& key_points_1, const std::vector<cv::
             KeyPoint>& key_points_2,
        const std::vector<cv::DMatch >& matches, cv::Mat& final_H, std::vector<cv::
             Vec3d>& final_img1_pts_vec3d, std::vector<cv::Vec3d>&
             final_img2_pts_vec3d,
        std::vector<cv::KeyPoint>& max_inliner_keypoins_1,
        std::vector<cv::KeyPoint>& max_inliner_keypoins_2,
        std::vector<cv::DMatch>& max_inliner_matches
                ) {

        //const double sigma = 5;
        // threshold for deciding whether an inlier
        const double delta = 3;


        // N = no of trials
        const double p = 0.99;
        const int n = 6;
        const int N = std::log(1 - p) / std::log(1 - std::pow((1 -
             epsilon_percentage_of_outliers), n));

        // M = minimum size of the inlier set for it to be considered acceptable
        const int M = round((1 - epsilon_percentage_of_outliers) *
             n_total_correspondences);

        cv::Mat max_h;

        std::random_device rd;
        std::mt19937 eng(rd());
        std::uniform_int_distribution<int> dist(0, n_total_correspondences - 1);

        std::vector<cv::DMatch> selected_matches(n);

        // max
        int max_inliers = 0;
        std::vector<cv::Vec3d> max_img1_pts_vec3d;
        std::vector<cv::Vec3d> max_img2_pts_vec3d;



        for (int i = 0; i < N; ++i) {
                // randomly select n points
                std::set<int> selected_matches_index;
                for (int j = 0; j < n || selected_matches_index.size() < n; ++j) {
                        selected_matches_index.insert(dist(eng));
                }

                std::vector<cv::Vec3d> inlier_img1_pts_vec3d;
                std::vector<cv::Vec3d> inlier_img2_pts_vec3d;


                // construct Homography
                int j = 0;
                for (auto itr = selected_matches_index.begin(); itr !=
                     selected_matches_index.end(); ++itr, ++j) {
                        selected_matches[j] = matches[*itr];
                        cv::KeyPoint img_pt1 = key_points_1[matches[*itr].queryIdx];
                        cv::KeyPoint img_pt2 = key_points_2[matches[*itr].trainIdx];
                        cv::Vec3d img_pt1_vec(img_pt1.pt.x, img_pt1.pt.y, 1.0);
                        cv::Vec3d img_pt2_vec(img_pt2.pt.x, img_pt2.pt.y, 1.0);
                        inlier_img1_pts_vec3d.push_back(img_pt1_vec);
                        inlier_img2_pts_vec3d.push_back(img_pt2_vec);
                }

                Homography homography(key_points_1, key_points_2, selected_matches);
                cv::Mat H = homography.get_homography();
                // calculate outliers and inliers
                std::vector<cv::KeyPoint> tmp_max_inliner_keypoins_1;
                std::vector<cv::KeyPoint> tmp_max_inliner_keypoins_2;
                std::vector<cv::DMatch> tmp_max_inliner_matches;

                for (j = 0; j < matches.size(); ++j) {

                        cv::KeyPoint img_pt1 = key_points_1[matches[j].queryIdx];
                        cv::KeyPoint img_pt2 = key_points_2[matches[j].trainIdx];

                        tmp_max_inliner_keypoins_1.push_back(img_pt1);
                        tmp_max_inliner_keypoins_2.push_back(img_pt2);

                        // make sure it's not a randomly selected point
                        if (selected_matches_index.find(j) != selected_matches_index
                             .end()) {
                                cv::DMatch match;
                                match.queryIdx = tmp_max_inliner_keypoins_1.size() -
                                      1;
                                match.trainIdx = tmp_max_inliner_keypoins_2.size() -
                                      1;
                                tmp_max_inliner_matches.push_back(match);
                                continue;
                        }

                        cv::Vec3d img_pt1_vec(img_pt1.pt.x, img_pt1.pt.y, 1.0);
                        cv::Vec3d img_pt2_vec(img_pt2.pt.x, img_pt2.pt.y, 1.0);

                        cv::Mat calculated_pt_mat = H * cv::Mat(img_pt1_vec);
                        cv::Vec3d calculated_pt(calculated_pt_mat);

                        cv::Vec2d img_pt2_vec2d(img_pt2_vec[0], img_pt2_vec[1]);
                        if (std::abs(calculated_pt[2]) < 1e-6) {
                                continue;
                        }
                        cv::Vec2d calculated_pt_vec2d(calculated_pt[0] /
                             calculated_pt[2], calculated_pt[1] / calculated_pt[2]);
                        double distance = cv::norm(calculated_pt_vec2d -
                             img_pt2_vec2d);
                        if (distance < delta) {
                                inlier_img1_pts_vec3d.push_back(img_pt1_vec);
                                inlier_img2_pts_vec3d.push_back(img_pt2_vec);
                                cv::DMatch match;
                                match.queryIdx = tmp_max_inliner_keypoins_1.size() -
                                      1;
                                match.trainIdx = tmp_max_inliner_keypoins_2.size() -
                                      1;
                                tmp_max_inliner_matches.push_back(match);
                        }

                }

                if (max_inliers < inlier_img1_pts_vec3d.size()) {
                        // new max inliers
                        max_inliers = inlier_img1_pts_vec3d.size();
                        max_img1_pts_vec3d = inlier_img1_pts_vec3d;
                        max_img2_pts_vec3d = inlier_img2_pts_vec3d;

                        max_inliner_keypoins_1 = tmp_max_inliner_keypoins_1;
                        max_inliner_keypoins_2 = tmp_max_inliner_keypoins_2;
                        max_inliner_matches = tmp_max_inliner_matches;

                }
        }
        // refine homography with max inlier set
        Homography max_homography(max_img1_pts_vec3d, max_img2_pts_vec3d);
        final_H = max_homography.get_homography();
        final_img1_pts_vec3d = max_img1_pts_vec3d;
        final_img2_pts_vec3d = max_img2_pts_vec3d;

        if (max_img1_pts_vec3d.size() <= M) {
                std::cout << "#␣of␣inliners␣not␣reached␣:␣needed␣=␣" << M << "␣
                      actual␣=␣" << max_img1_pts_vec3d.size() << "\n";
        }
        std::cout << "#␣of␣inliners␣:␣" << max_img1_pts_vec3d.size() << "␣M:␣" << M
              << "\n";

}

void Homography::construct_first_row(const cv::Vec3d& img1_pt, const cv::Vec3d&
      img2_pt, int pts_i, cv::Mat& A) {
        for (int i = 0; i < 3; ++i) {
                A.at<double>(2 * pts_i, i) = 0.0;
        }

        for (int i = 3; i < 6; ++i) {
                A.at<double>(2 * pts_i, i) = -1.0 * img2_pt[2] * img1_pt(i - 3);
        }

        for (int i = 6; i < 9; ++i) {
                A.at<double>(2 * pts_i, i) = img2_pt[1] * img1_pt(i - 6);
        }
}

void Homography::construct_second_row(const cv::Vec3d& img1_pt, const cv::Vec3d&
      img2_pt, int pts_i, cv::Mat& A) {
        for (int i = 0; i < 3; ++i) {
                A.at<double>(2 * pts_i + 1, i) = img2_pt[2] * img1_pt(i);
        }
        for (int i = 3; i < 6; ++i) {
                A.at<double>(2 * pts_i + 1, i) = 0.0;
        }

        for (int i = 6; i < 9; ++i) {
                A.at<double>(2 * pts_i + 1, i) = -1.0 * img2_pt[0] * img1_pt(i - 6);
        }
}


Homography::Homography(const std::vector<cv::KeyPoint>& key_points_1, const std::
     vector<cv::KeyPoint>& key_points_2, const std::vector<cv::DMatch>&
     selected_matches) {

        cv::Mat A(selected_matches.size() * 2, 9, CV_64F);
        std::vector<cv::Vec3d> img_vec_array_1;
        std::vector<cv::Vec3d> img_vec_array_2;

        for (int i = 0; i < selected_matches.size(); ++i) {
                cv::KeyPoint img_pt1 = key_points_1[selected_matches[i].queryIdx];
                cv::KeyPoint img_pt2 = key_points_2[selected_matches[i].trainIdx];

                cv::Vec3d img_vec_1(img_pt1.pt.x, img_pt1.pt.y, 1.0);
                cv::Vec3d img_vec_2(img_pt2.pt.x, img_pt2.pt.y, 1.0);

                img_vec_array_1.push_back(img_vec_1);
                img_vec_array_2.push_back(img_vec_2);

                //construct_first_row(img_vec_1, img_vec_2, i, A);
                //construct_second_row(img_vec_1, img_vec_2, i, A);
        }

        construct_H(img_vec_array_1, img_vec_array_2);

        //cv::Mat H_vector;
        //LinearLeastSquares::linear_least_squares_for_homography(A, H_vector);

        //H = cv::Mat(3, 3, CV_64F);
        //for (int i = 0; i < 3; ++i) {
        //        for (int j = 0; j < 3; ++j) {
        //                H.at<double>(i, j) = H_vector.at<double>(3 * i + j, 0);
        //        }
        //}
}

void Homography::construct_H(const std::vector<cv::Vec3d>& img_vec_array_1, const
     std::vector<cv::Vec3d>& img_vec_array_2) {
        cv::Mat A(img_vec_array_1.size() * 2, 9, CV_64F);
        for (int i = 0; i < img_vec_array_1.size(); ++i) {
                construct_first_row(img_vec_array_1[i], img_vec_array_2[i], i, A);
                construct_second_row(img_vec_array_1[i], img_vec_array_2[i], i, A);
        }
        cv::Mat H_vector;
        LinearLeastSquares::linear_least_squares_for_homography(A, H_vector);

        H = cv::Mat(3, 3, CV_64F);
        for (int i = 0; i < 3; ++i) {
                for (int j = 0; j < 3; ++j) {
                        H.at<double>(i, j) = H_vector.at<double>(3 * i + j, 0);
                }
        }

}

Homography::Homography(const std::vector<cv::Vec3d>& img_vec_array_1, const std::
     vector<cv::Vec3d>& img_vec_array_2) {
        construct_H(img_vec_array_1, img_vec_array_2);

}

cv::Mat Homography::get_homography() const {
        return H;
}

void DogLeg::construct_first_row_of_jacobian(const cv::Vec3d& img1_pt, const cv::
     Vec3d& calculated_pt, int pts_i, cv::Mat& J) {
        for (int i = 0; i < 3; ++i) {
                J.at<double>(2 * pts_i, i) = img1_pt[i] / calculated_pt[2];
        }

        for (int i = 3; i < 6; ++i) {
                J.at<double>(2 * pts_i, i) = 0.0;
        }

        for (int i = 6; i < 9; ++i) {
                J.at<double>(2 * pts_i, i) = -1.0 * img1_pt[i - 6] * calculated_pt
                     [0] / std::pow(calculated_pt[2], 2);
        }
}

void DogLeg::construct_second_row_of_jacobian(const cv::Vec3d& img1_pt, const cv::
     Vec3d& calculated_pt, int pts_i, cv::Mat& J) {
        for (int i = 0; i < 3; ++i) {
                J.at<double>(2 * pts_i + 1, i) = 0.0;
        }

        for (int i = 3; i < 6; ++i) {
                J.at<double>(2 * pts_i + 1, i) = img1_pt[i - 3] / calculated_pt[2];
        }

        for (int i = 6; i < 9; ++i) {
                J.at<double>(2 * pts_i + 1, i) = -1.0 * img1_pt[i - 6] *
                     calculated_pt[1] / std::pow(calculated_pt[2], 2);
        }
}

void DogLeg::compute_jacobian(std::vector<cv::Vec3d>& img1_pts, std::vector<cv::
     Vec3d>& calculated_pts, cv::Mat& jacobian) {
        for (int i = 0; i < img1_pts.size(); ++i) {
                construct_first_row_of_jacobian(img1_pts[i], calculated_pts[i], i,
                     jacobian);
                construct_second_row_of_jacobian(img1_pts[i], calculated_pts[i], i,
                     jacobian);
        }
}

void DogLeg::compute_error_vector(std::vector<cv::Vec3d>& ground_truth, std::vector<
     cv::Vec3d>& calculated_pts, cv::Mat& error_vector) {
        for (int i = 0; i < ground_truth.size(); ++i) {
                error_vector.at<double>(2 * i, 0) = ground_truth[i][0] - (
                     calculated_pts[i][0] / calculated_pts[i][2]);
                error_vector.at<double>(2 * i + 1, 0) = ground_truth[i][1] - (
                     calculated_pts[i][1] / calculated_pts[i][2]);
        }
}

void DogLeg::compute_gradient_descent_increment(const cv::Mat& J, const cv::Mat&
     error_vector, cv::Mat& gradient_descent_increment) {
        cv::Mat increment = J.t() * error_vector;
        double numerator = cv::norm(increment);
        double denominator = cv::norm(J * increment);

        gradient_descent_increment = (numerator / denominator) * increment;
}

void DogLeg::compute_gauss_newton_increment(const cv::Mat& J, const cv::Mat&
     error_vector, const cv::Mat& identity, double mu_k, cv::Mat&
     gauss_newton_increment) {

        cv::Mat multiplier = J.t() * J + mu_k * identity;
        gauss_newton_increment = multiplier.inv() * J.t() * error_vector;

}

double DogLeg::compute_beta(const cv::Mat& gauss_newton_increment, const cv::Mat&
     gradient_descent_increment, double r_k) {
        cv::Mat difference_increment = gauss_newton_increment -
             gradient_descent_increment;
        double a = std::pow(cv::norm(difference_increment), 2);
        cv::Mat result_b_coeff = gradient_descent_increment.t() *
             difference_increment;
        double b = 2.0 * result_b_coeff.at<double>(0, 0);
        double c = std::pow(cv::norm(gradient_descent_increment), 2) - std::pow(r_k,
              2);

        double determinant = std::pow(b, 2) - 4 * a * c;
        if (determinant  < 0) {
                std::cout << "Undefined␣condition␣occurred.\n";
        }

        double beta = -b + std::sqrt(determinant) / (2 * a);

        return beta;
}

void DogLeg::convert_to_H_k(const cv::Mat& P_k, cv::Mat& H_k) {
        for (int i = 0; i < 3; ++i) {
                for (int j = 0; j < 3; ++j) {
                        H_k.at<double>(i, j) = P_k.at<double>(3 * i + j, 0);
                }
        }
}

void DogLeg::convert_to_P_k(const cv::Mat& H_k, cv::Mat& P_k) {
        for (int i = 0; i < 3; ++i) {
                for (int j = 0; j < 3; ++j) {
                         P_k.at<double>(3 * i + j, 0) = H_k.at<double>(i, j);
                }
        }
}

double DogLeg::initialize_mu(const cv::Mat& J, double tau) {
        cv::Mat J_squared = J.t() * J;
        double max = -DBL_MAX;
        for (int i = 0; i < J_squared.rows; ++i) {
                max = std::max(max, J_squared.at<double>(i, i));
        }
        return max * tau;
}

void DogLeg::calculate_result_pts(const std::vector<cv::Vec3d>& img1_pts, const cv::
     Mat& H_k, std::vector<cv::Vec3d>& calculated_pts) {
        for (int i = 0; i < img1_pts.size(); ++i) {
                cv::Mat img_1_pts_mat = cv::Mat(img1_pts[i]);
                cv::Mat calculated_pts_mat = H_k * img_1_pts_mat;
                calculated_pts[i] = calculated_pts_mat;
        }
```

```cpp
}

void DogLeg::dogleg_for_non_linear_least_squares_optimization(const cv::Mat& H, std
     ::vector<cv::Vec3d>& img1_pts, std::vector<cv::Vec3d>& img2_pts, cv::Mat&
     output_H) {

        cv::Mat P_k(9, 1, CV_64F);

        cv::Mat H_k = H;
        cv::Mat H_k_plus_1 = H;

        convert_to_P_k(H_k, P_k);

        int some_iterations = 10000;
        double error_threshold = 1e-10;


        std::vector<cv::Vec3d> calculated_pts(img1_pts.size());
        std::vector<cv::Vec3d> calculated_k_plus_1_pts(img1_pts.size());
        cv::Mat J(img1_pts.size() * 2, 9, CV_64F);
        cv::Mat error_vector(img1_pts.size() * 2, 1, CV_64F);
        cv::Mat error_vector_k_plus_1(img1_pts.size() * 2, 1, CV_64F);

        cv::Mat gradient_descent_increment(9, 1, CV_64F);
        cv::Mat gauss_newton_increment(9, 1, CV_64F);

        calculate_result_pts(img1_pts, H_k, calculated_pts);
        compute_jacobian(img1_pts, calculated_pts, J);
        compute_error_vector(img2_pts, calculated_pts, error_vector);

        double calculated_error = cv::norm(error_vector);

        double tau = 0.001;
        double mu_k = initialize_mu(J, tau);
        double r_k = 0.5;

        cv::Mat identity(9, 9, CV_64F);
        cv::setIdentity(identity);

        cv::Mat last_increment;

        int n = 0;
        for (; n < some_iterations && calculated_error > error_threshold; ++n) {

                convert_to_H_k(P_k, H_k);
                calculate_result_pts(img1_pts, H_k, calculated_pts);

                compute_jacobian(img1_pts, calculated_pts, J);
                compute_error_vector(img2_pts, calculated_pts, error_vector);

                compute_gradient_descent_increment(J, error_vector,
                     gradient_descent_increment);
                compute_gauss_newton_increment(J, error_vector, identity, mu_k,
                     gauss_newton_increment);

                double gauss_newton_norm = cv::norm(gauss_newton_increment);
                double gradient_descent_norm = cv::norm(gradient_descent_norm);


                cv::Mat increment;
                if (gauss_newton_norm < r_k) {
                        increment = gauss_newton_increment;
                } else if (gradient_descent_norm < r_k && r_k < gauss_newton_norm) {
                        double beta = compute_beta(gauss_newton_increment,
                             gradient_descent_increment, r_k);
                        increment = gradient_descent_increment + beta * (
                             gauss_newton_increment - gradient_descent_increment);
                } else {
                        increment = (r_k / gradient_descent_norm) *
                             gradient_descent_increment;
                }

                cv::Mat temp_P_k_plus_1 = P_k + increment;

                // calculate of C_p(k) and C_p(k+1)
                convert_to_H_k(temp_P_k_plus_1, H_k_plus_1);
                calculate_result_pts(img1_pts, H_k_plus_1, calculated_k_plus_1_pts);
                compute_error_vector(img2_pts, calculated_k_plus_1_pts,
                     error_vector_k_plus_1);

                cv::Mat C_p_mat = error_vector.t() * error_vector;
                cv::Mat C_p_plus_1_mat = error_vector_k_plus_1.t() *
                     error_vector_k_plus_1;

                // calculate row_LM
                double row_lm_numerator = C_p_mat.at<double>(0, 0) - C_p_plus_1_mat.
                     at<double>(0, 0);
                cv::Mat row_lm_denominator_mat = increment.t() * J.t() *
                     error_vector + increment.t() * mu_k * identity * increment;

                double row_LM = row_lm_numerator / row_lm_denominator_mat.at<double
                     >(0, 0);


                // calculate row_DL
                double row_DL_numerator = row_lm_numerator;
                cv::Mat row_DL_denominator_mat = 2.0 * increment.t() * J.t() *
                     error_vector - increment.t() * J.t() * J * increment;
                double row_DL = row_DL_numerator / row_DL_denominator_mat.at<double
                     >(0, 0);

                if (row_DL <= 0.0) {
                        // we've jumped too far, revert
                        r_k = r_k / 2.0;
                        mu_k = 2 * mu_k;
                } else {
                        calculated_error = row_lm_numerator;
                        mu_k =  mu_k * std::max(1.0/3.0, 1 - std::pow(2 * row_LM -
                             1, 3));
                        if (row_DL < 0.25) {
                                r_k = r_k / 4.0;
                        } else if (0.25 <= row_DL && row_DL <= 0.75) {
                                // just for the if
                                r_k = r_k;
                        } else {
                                r_k = r_k * 2.0;
                        }
                        P_k = temp_P_k_plus_1;
                }
        }
        output_H = cv::Mat(3, 3, CV_64F);
        convert_to_H_k(P_k, output_H);

        std::cout << "#␣of␣iterations␣:␣" << n << "␣,␣error␣:␣" << calculated_error
             << "\n";


}

void ImgUtils::combine_transformed_imgs(const cv::Mat& img_1, const cv::Mat& img_2,
     cv::Mat& H, cv::Mat& transformed_img) {

        std::vector<cv::Point2d> corner_pts;
        corner_pts.push_back(cv::Point2d(0, 0));
        corner_pts.push_back(cv::Point2d(img_1.cols, 0));
        corner_pts.push_back(cv::Point2d(0, img_1.rows));
        corner_pts.push_back(cv::Point2d(img_1.cols, img_1.rows));

        std::vector<cv::Point2d> projected_pts;
        for (auto& corner_pt : corner_pts) {
                cv::Point2d projected_pt = ImgUtils::apply_custom_homography(H,
                     corner_pt);
                projected_pts.push_back(projected_pt);
        }

        projected_pts.push_back(cv::Point2d(0, 0));
        projected_pts.push_back(cv::Point2d(img_2.cols, 0));
        projected_pts.push_back(cv::Point2d(0, img_2.rows));
        projected_pts.push_back(cv::Point2d(img_2.cols, img_2.rows));

        double minx, maxx, miny, maxy;
        ImgUtils::find_bounding_box(projected_pts, minx, maxx, miny, maxy);

        const int width = std::ceil(maxx - minx);
        float width_ratio = width / (float)(maxx-minx);
        float aspect_ratio = (float)(maxx-minx) / (float)(maxy-miny);

        //int height = width / aspect_ratio;
        int height = std::ceil(maxy - miny);

        int offset_x = minx;
        int offset_y = miny;

        transformed_img = cv::Mat(height, width, CV_8UC3);

        //cv::Mat inverse_homography = H.inv();

        cv::Point2d offset(offset_x, offset_y);

        // initially put img2 on to the final_img
        //for (int y = 0; y < img_2.rows; ++y) {
        //      for (int x = 0; x < img_2.cols; ++x) {
        //              //int transformed_x = x + width - img_2.cols;
        //              //int transformed_y = y + height - img_2.rows;
        //              cv::Point2d transformed_pt(x, y);
        //              transformed_pt -= offset;

        //              if (transformed_pt.y >= 0 && transformed_pt.y <
             transformed_img.rows
        //                      && transformed_pt.x >= 0 && transformed_pt.x <
             transformed_img.cols) {
        //                              transformed_img.at<cv::Vec3b>(transformed_pt
             .y, transformed_pt.x) = img_2.at<cv::Vec3b>(y, x);
        //              } else {
        //                      std::cout << transformed_pt.x << ", " <<
             transformed_pt.y << "\n";
        //              }
        //      }
        //}

        //for (int y = 0; y < img_1.rows; ++y) {
        //      for (int x = 0; x < img_1.cols; ++x) {

        //              cv::Point2d img_pt(x, y);
        //              cv::Point2d corresponding_image_point = ImgUtils::
             apply_custom_homography(H, img_pt);
        //                      corresponding_image_point  -= offset;
        //                      if (corresponding_image_point.y  >= 0 &&
              corresponding_image_point.y < transformed_img.rows
        //                                 && corresponding_image_point.x  >= 0 &&
              corresponding_image_point.x < transformed_img.cols) {
        //                                         transformed_img.at<cv::Vec3b>(
              corresponding_image_point.y, corresponding_image_point.x) = img_1.at<cv
              ::Vec3b>(y, x);
        //                      }
        //              }
        //}

         //inverse route
        cv::Mat inv_H = H.inv();

        for (int y = 0; y < transformed_img.rows; ++y) {
                for (int x = 0; x < transformed_img.cols; ++x) {

                        cv::Point2d img_pt(x, y);
                        cv::Point2d offset_pt = img_pt + offset;
                        cv::Point2d corresponding_image_point = ImgUtils::
                              apply_custom_homography(inv_H, offset_pt);

                        if (corresponding_image_point.y >= 0 &&
                              corresponding_image_point.y < img_1.rows
                                  && corresponding_image_point.x >= 0 &&
                                        corresponding_image_point.x < img_1.cols) {
                                            transformed_img.at<cv::Vec3b>(y, x) = img_1.
                                                  at<cv::Vec3b>(corresponding_image_point
                                                  .y, corresponding_image_point.x);
                        } else {
                                //cv::Point2d transformed_pt(x, y);
                                //transformed_pt += offset;
                                //if (transformed_pt.y >= 0 && transformed_pt.y <
                                      img_2.rows
                                //      && transformed_pt.x >= 0 && transformed_pt.x
                                       < img_2.cols) {
                                //              transformed_img.at<cv::Vec3b>(y, x)
                                      = img_2.at<cv::Vec3b>(transformed_pt.y,
                                      transformed_pt.x);
                                //}

                        }
                }
        }
}

void ImgUtils::get_bounding_box(const std::vector<cv::Mat>& imgs, const std::vector<
      cv::Mat>& Hs,
        double& minx, double& maxx, double& miny, double& maxy) {

        std::vector<cv::Point2d> projected_pts;

        for (int i = 0; i < imgs.size(); ++i) {
                cv::Mat img = imgs[i];
                std::vector<cv::Point2d> corner_pts;
                corner_pts.push_back(cv::Point2d(0, 0));
                corner_pts.push_back(cv::Point2d(img.cols, 0));
                corner_pts.push_back(cv::Point2d(0, img.rows));
                corner_pts.push_back(cv::Point2d(img.cols, img.rows));

                for (auto& corner_pt : corner_pts) {
                        cv::Point2d projected_pt = ImgUtils::apply_custom_homography
                              (Hs[i], corner_pt);
                        projected_pts.push_back(projected_pt);
                }
        }

        ImgUtils::find_bounding_box(projected_pts, minx, maxx, miny, maxy);

}

void ImgUtils::combine_imgs_to_middle_img(const std::vector<cv::Mat>& imgs, const
      std::vector<cv::Mat>& Hs, cv::Mat& transformed_img) {
        int middle_index = imgs.size() / 2;

        //std::vector<cv::Mat> final_homographies(Hs.size() + 1);
        //for (int i = 0; i < imgs.size(); ++i) {
        //      int multiplications = std::abs(middle_index - i);
        //      cv::Mat final_H = cv::Mat(3, 3, CV_64F);
        //      cv::setIdentity(final_H);

        //      if ((middle_index - i) > 0) {
        //              for (int j = 0; j < multiplications; ++j) {
        //                      final_H = final_H * Hs[j];
        //              }
        //      } else if ((middle_index - i) < 0) {
        //              for (int j = i; j < multiplications; ++j) {
        //                      final_H = final_H * Hs[Hs.size() - 1 - j].inv();
        //              }
        //      }
        //      final_homographies[i] = final_H;
        //      // middle index == 0
        //}

        // for 5
        std::vector<cv::Mat> final_homographies(5);
        cv::Mat identity = cv::Mat(3, 3, CV_64F);
        cv::setIdentity(identity);
        // H_1_3
        final_homographies[0] = Hs[0] * Hs[1];
        // H_2_3
        final_homographies[1] = Hs[1];
        // H_3_3
        final_homographies[middle_index] = identity;
        // H_4_3 = H_3_4.inv()
        final_homographies[3] = Hs[2].inv();
        // H_5_3 = H_4_5.inv() * H_3_4.inv()
        final_homographies[4] = Hs[3].inv() * Hs[2].inv();



        double minx, maxx, miny, maxy;
        ImgUtils::get_bounding_box(imgs, final_homographies, minx, maxx, miny, maxy)
             ;

        const int width = std::ceil(maxx - minx);
        float width_ratio = width / (float)(maxx-minx);
        float aspect_ratio = (float)(maxx-minx) / (float)(maxy-miny);

        //int height = width / aspect_ratio;
        int height = std::ceil(maxy - miny);

        int offset_x = minx;
        int offset_y = miny;

        cv::Point2d offset(offset_x, offset_y);

        transformed_img = cv::Mat(height, width, CV_8UC3);



        //inverse route
        for (int i = 0; i < imgs.size(); ++i) {

                cv::Mat inv_H = final_homographies[i].inv();

                cv::Mat img_1 = imgs[i];

                //const int no_of_rows = height;

#pragma loop(hint_parallel(8))
                for (int y = 0, no_of_rows = height; y < no_of_rows; ++y) {
                        for (int x = 0; x < transformed_img.cols; ++x) {

                                cv::Point2d img_pt(x, y);
                                cv::Point2d offset_pt = img_pt + offset;
                                cv::Point2d corresponding_image_point = ImgUtils::
                                     apply_custom_homography(inv_H, offset_pt);

                                if (corresponding_image_point.y >= 0 &&
                                     corresponding_image_point.y < img_1.rows
                                         && corresponding_image_point.x >= 0 &&
                                             corresponding_image_point.x < img_1.
                                             cols) {
                                                transformed_img.at<cv::Vec3b>(y, x)
                                                     = img_1.at<cv::Vec3b>(
                                                     corresponding_image_point.y,
                                                     corresponding_image_point.x);
                                } else {
                                        //cv::Point2d transformed_pt(x, y);
                                        //transformed_pt += offset;
                                        //if (transformed_pt.y >= 0 &&
                                             transformed_pt.y < img_2.rows
                                        //      && transformed_pt.x >= 0 &&
                                             transformed_pt.x < img_2.cols) {
                                        //              transformed_img.at<cv::Vec3b
                                             >(y, x) = img_2.at<cv::Vec3b>(
                                             transformed_pt.y, transformed_pt.x);
                                        //}

                                }
                        }
                }
        }



}

void ImgUtils::apply_distortion_correction(cv::Mat& homography_matrix, const cv::Mat
     & img, cv::Mat& transformed_img, const std::string& final_image_name) {



        cv::imwrite(final_image_name, transformed_img);
}

cv::Point2d ImgUtils::apply_custom_homography(const cv::Mat matrix, const cv::
     Point2d& src_pt) {
        cv::Vec3d homogenous_pt(src_pt.x, src_pt.y, 1.0);
        cv::Mat mapped_pt = matrix * cv::Mat(homogenous_pt);
        cv::Point2d mapped_point2f(mapped_pt.at<double>(0,0) / mapped_pt.at<double
             >(2, 0),
                mapped_pt.at<double>(1, 0) / mapped_pt.at<double>(2, 0));
        return mapped_point2f;
}
```

```
void ImgUtils::find_bounding_box(const std::vector<cv::Point2d>& points, double&
     min_x, double& max_x, double& min_y, double& max_y) {
        //assert(points.size() == 4);

        min_x = DBL_MAX;
        max_x = -DBL_MAX;
        min_y = DBL_MAX;
        max_y = -DBL_MAX;

        for (int i = 0; i < points.size(); ++i) {
                min_x = std::min(points[i].x, min_x);
                min_y = std::min(points[i].y, min_y);
                max_x = std::max(points[i].x, max_x);
                max_y = std::max(points[i].y, max_y);
        }
}

int main(int argc, char** argv)  {

#ifdef _DEBUG
        std::string dir = "dset2/";
#else
        std::string dir = "dset2/";
#endif

        const int no_of_images = 5;

        std::vector<cv::Mat> imgs;
        for (int i = 0; i < no_of_images; ++i) {
                std::stringstream ss;
                ss << dir;
                ss << (i+2) << ".jpg";
                cv::Mat img = cv::imread(ss.str());
                imgs.push_back(img);
        }

        // SIFT extract features
        std::vector<std::vector<cv::KeyPoint>> key_points_in_imgs;
        cv::SiftFeatureDetector sift_feature_detector(300);
        sift_feature_detector.detect(imgs, key_points_in_imgs);

        std::vector<cv::Mat> descriptors(no_of_images);

        for (int i = 0; i < no_of_images; ++i) {
                cv::SiftDescriptorExtractor sift_descriptor_extractor;
                sift_descriptor_extractor.compute(imgs[i], key_points_in_imgs[i],
                     descriptors[i]);
        }

        // consider img 1 and 2
        double threshold = 10;
        std::vector<std::vector<cv::DMatch>> matches_in_imgs(no_of_images - 1);

        double epsilon_percentage_of_outliers = 0.1;
        cv::Mat output_img = imgs[0];
        std::vector<cv::Mat> opt_homographies(no_of_images - 1);
        for (int i = 0; i < no_of_images - 1; ++i) {
                EuclideanDistance::match_by_euclidean_distance(key_points_in_imgs[i
                     ], descriptors[i], key_points_in_imgs[i+1], descriptors[i+1],
                     threshold, matches_in_imgs[i]);
                cv::Mat match_img;
                cv::drawMatches(imgs[i], key_points_in_imgs[i], imgs[i+1],
                     key_points_in_imgs[i+1], matches_in_imgs[i], match_img);
                cv::imshow("match", match_img);
                cv::imwrite("correspondence_" + std::to_string(i+1) + ".jpg",
                     match_img);
                std::vector<cv::Vec3d> img_1_inliers;
                std::vector<cv::Vec3d> img_2_inliers;
                cv::Mat H;
                std::vector<cv::KeyPoint> max_inliner_keypoins_1;
                std::vector<cv::KeyPoint> max_inliner_keypoins_2;
                std::vector<cv::DMatch> max_inliner_matches;
                Ransac::ransac_for_homography(epsilon_percentage_of_outliers,
                     matches_in_imgs[i].size(), key_points_in_imgs[i],
                         key_points_in_imgs[i+1],  matches_in_imgs[i], H,
                              img_1_inliers, img_2_inliers,
                                  max_inliner_keypoins_1, max_inliner_keypoins_2,
                                         max_inliner_matches);

                cv::Mat inliner_img;
                cv::drawMatches(imgs[i], max_inliner_keypoins_1, imgs[i+1],
                     max_inliner_keypoins_2, max_inliner_matches, inliner_img,
                         cv::Scalar(255.0, 0., 0.), cv::Scalar(0.0, 0., 255.));
                cv::imshow("inliers", inliner_img);
                cv::imwrite("inliers_" + std::to_string(i+1) + ".jpg", inliner_img);

                opt_homographies[i] = H;
                cv::Mat opt_H;
                DogLeg::dogleg_for_non_linear_least_squares_optimization(H,
                     img_1_inliers, img_2_inliers, opt_H);
                //opt_homographies[i] = opt_H;
        }


        cv::Mat transformed_img;
        ImgUtils::combine_imgs_to_middle_img(imgs, opt_homographies, transformed_img
             );

        output_img = transformed_img;
        cv::imshow("final_img", output_img);
        cv::imwrite("final_img.jpg", output_img);

        cv::waitKey(0);
}
```

Bundle Adjusting

```
#pragma once
#include "opencv2/core/core.hpp"
#include "opencv2/features2d/features2d.hpp"


extern
int fit_homographies(std::vector<cv::Mat>& homographies, std::vector<std::vector<cv
     ::KeyPoint>>& key_points_1, std::vector<std::vector<cv::KeyPoint>>&
     key_points_2,
        std::vector<std::vector<cv::DMatch>>& matches);
```

```
#include "hfit.h"
#include <math.h>
#include <malloc.h>
#include "opencv2/features2d/features2d.hpp"


extern "C" int mylmdif_(int (*fcn)(int *, int *, double *, double *, int *), int *m,
       int *n, double *x, double *fvec, double *ftol, double *xtol, double *gtol, int
       *maxfev,
         double *epsfcn, double *diag, int *mode, double *factor, int *nprint, int *
              info, int *nfev, double *fjac, int *ldfjac, int *ipvt,
         double *qtf, double *wa1, double *wa2, double *wa3, double *wa4);
//
//
///*****************************************************************************
//*****************************************************************************/

std::vector<std::vector<cv::KeyPoint>> key_points_1_g;
std::vector<std::vector<cv::KeyPoint>> key_points_2_g;
std::vector<std::vector<cv::DMatch>> matches_g;
int homography_size;

static double calc_error(int match_i, cv::DMatch match, const cv::Mat& homography) {
                cv::KeyPoint img_pt1 = key_points_1_g[match_i][match.queryIdx];
                cv::KeyPoint img_pt2 = key_points_2_g[match_i][match.trainIdx];

                cv::Vec3d img_pt1_vec3d(img_pt1.pt.x, img_pt1.pt.y, 1.0);

                cv::Vec2d img_pt2_vec2d(img_pt2.pt.x, img_pt2.pt.y);

                cv::Mat calc_pt = homography * cv::Mat(img_pt1_vec3d);
                cv::Vec3d calc_pt_vec3d(calc_pt);

                cv::Vec2d calc_pt_vec2d(calc_pt_vec3d[0] / calc_pt_vec3d[2],
                     calc_pt_vec3d[1] / calc_pt_vec3d[2]);
                double error = cv::norm(calc_pt_vec2d - img_pt2_vec2d);
                return error;
}

static int
lmdifError_(int *m_ptr, int *n_ptr, double *params, double *error, int *)
{
        int nparms = *n_ptr;
        int nerrors = *m_ptr;

        std::vector<cv::Mat> homographies(homography_size);


        for (int k = 0; k < homographies.size(); ++k) {
                homographies[k] = cv::Mat(3, 3, CV_64F);
                for (int i = 0; i < 3; ++i) {
                        for (int j = 0; j < 3; ++j) {
                                homographies[k].at<double>(i, j) = params[k * 9 + i
                                      * 3 + j];
                        }
                }
        }

        // H_{01}, H_{12}, H_{23}, H_{34}

        // calc error

        // every match has from 0->1, 0->2, 0->3, 0->4
        // then from 1->2, 1->3, 1->4
        // then from 2->3, 2->4
        // then from 3->4
        // assume everything has 10 points or less

        // cover the 9 cases
        int k = 0;
        // 0->1
        int index = 0;
        for (int i = 0; i < matches_g[0].size(); ++i) {
                auto match = matches_g[0][i];
                cv::Mat homography = homographies[0];
                error[index + i] = calc_error(0, match, homography);
        }
        index += matches_g[0].size();

        // 0->2
        for (int i = 0; i < matches_g[1].size(); ++i) {
                auto match = matches_g[1][i];
                cv::Mat homography = homographies[1] * homographies[0];
                error[index + i] = calc_error(1, match, homography);
        }
        index += matches_g[1].size();

        // 0->3
        for (int i = 0; i < matches_g[2].size(); ++i) {
                auto match = matches_g[2][i];
                cv::Mat homography = homographies[2] * homographies[1] *
                     homographies[0];
                error[index + i] = calc_error(2, match, homography);
        }
        index += matches_g[2].size();

        // 0->4
        for (int i = 0; i < matches_g[3].size(); ++i) {
                auto match = matches_g[3][i];
                cv::Mat homography = homographies[3] * homographies[2] *
                     homographies[1] * homographies[0];
                error[index + i] = calc_error(3, match, homography);
        }
        index += matches_g[3].size();

        // 1->2
        k = 4;
        for (int i = 0; i < matches_g[k].size(); ++i) {
                auto match = matches_g[k][i];
                cv::Mat homography =  homographies[1];
                error[index + i] = calc_error(k, match, homography);
        }
        index += matches_g[k].size();

        // 1->3
        k = 5;
        for (int i = 0; i < matches_g[k].size(); ++i) {
                auto match = matches_g[k][i];
                cv::Mat homography = homographies[2] * homographies[1];
                error[index + i] = calc_error(k, match, homography);
        }
        index += matches_g[k].size();

        // 1->4
        k = 6;
        for (int i = 0; i < matches_g[k].size(); ++i) {
                auto match = matches_g[k][i];
                cv::Mat homography = homographies[3] * homographies[2] *
                     homographies[1];
                error[index + i] = calc_error(k, match, homography);
        }
        index += matches_g[k].size();


        // H_{01}, H_{12}, H_{23}, H_{34}
        // 2->3
        k = 7;
        for (int i = 0; i < matches_g[k].size(); ++i) {
                auto match = matches_g[k][i];
                cv::Mat homography = homographies[2];
                error[index + i] = calc_error(k, match, homography);
        }
        index += matches_g[k].size();

        // 2->4
        k = 8;
        for (int i = 0; i < matches_g[k].size(); ++i) {
                auto match = matches_g[k][i];
                cv::Mat homography = homographies[3] * homographies[2];
                error[index + i] = calc_error(k, match, homography);
        }
        index += matches_g[k].size();

        // 3->4
        k = 9;
        for (int i = 0; i < matches_g[k].size(); ++i) {
                auto match = matches_g[k][i];
                cv::Mat homography = homographies[3];
                error[index + i] = calc_error(k, match, homography);
        }
        index += matches_g[k].size();

        return 1;
}


/****************************************************************************
****************************************************************************/
/* Parameters controlling MINPACK's lmdif() optimization routine. */
/* See the file lmdif.f for definitions of each parameter.         */
#define REL_SENSOR_TOLERANCE_ftol    1.0E-6      /* [pix] */
#define REL_PARAM_TOLERANCE_xtol     1.0E-7
#define ORTHO_TOLERANCE_gtol         0.0
#define MAXFEV                       (1000*n)
#define EPSFCN                       1.0E-10 /* was E-16 Do not set to 0! */
```

```
#define MODE                        2       /* variables scaled internally */
#define FACTOR                      100.0


int fit_homographies(std::vector<cv::Mat>& homographies, std::vector<std::vector<cv
    ::KeyPoint>>& key_points_1, std::vector<std::vector<cv::KeyPoint>>&
    key_points_2,
        std::vector<std::vector<cv::DMatch>>& matches)
{
    /* Parameters needed by MINPACK's lmdif() */
        int     n = 9 * homographies.size();

        int no_of_errors = 0;
        for (int i = 0; i < matches.size(); ++i) {
                no_of_errors += matches[i].size();
        }

        int     m = no_of_errors;
    double *x;
    double *fvec;
    double  ftol = REL_SENSOR_TOLERANCE_ftol;
    double  xtol = REL_PARAM_TOLERANCE_xtol;
    double  gtol = ORTHO_TOLERANCE_gtol;
    int     maxfev = MAXFEV;
    double  epsfcn = EPSFCN;
    double *diag;
    int     mode = MODE;
    double  factor = FACTOR;
    int     ldfjac = m;
    int     nprint = 0;
    int     info;
    int     nfev;
    double *fjac;
    int    *ipvt;
    double *qtf;
    double *wa1;
    double *wa2;
    double *wa3;
    double *wa4;


         /* copy to globals */
         key_points_1_g = key_points_1;
         key_points_2_g = key_points_2;
         matches_g = matches;
         homography_size = homographies.size();

    /* allocate stuff dependent on n */
    x    = (double *)calloc(n, sizeof(double));
    diag = (double *)calloc(n, sizeof(double));
    qtf  = (double *)calloc(n, sizeof(double));
    wa1  = (double *)calloc(n, sizeof(double));
    wa2  = (double *)calloc(n, sizeof(double));
    wa3  = (double *)calloc(n, sizeof(double));
    ipvt = (int    *)calloc(n, sizeof(int));

    /* allocate some workspace */
    if (( fvec = (double *) calloc ((unsigned int) m,
                                    (unsigned int) sizeof(double))) == NULL ) {
       fprintf(stderr,"calloc:␣Cannot␣allocate␣workspace␣fvec\n");
       exit(-1);
    }

    if (( fjac = (double *) calloc ((unsigned int) m*n,
                                    (unsigned int) sizeof(double))) == NULL ) {
       fprintf(stderr,"calloc:␣Cannot␣allocate␣workspace␣fjac\n");
       exit(-1);
    }

    if (( wa4 = (double *) calloc ((unsigned int) m,
                                   (unsigned int) sizeof(double))) == NULL ) {
       fprintf(stderr,"calloc:␣Cannot␣allocate␣workspace␣wa4\n");
       exit(-1);
    }


    /* copy parameters in as initial values */
        for (int k = 0; k < homographies.size(); ++k) {
                for (int i = 0; i < 3; ++i) {
                        for (int j = 0; j < 3; ++j) {
                                x[k * 9 + i * 3 + j] = homographies[k].at<double>(i,
                                    j);
                        }
                }
        }

    /* define optional scale factors for the parameters */
    if ( mode == 2 ) {
                int offset = 0;
                for (int offset = 0; offset<n; offset++) {
                        diag[offset] = 1.0;
                }
    }

    /* perform the optimization */
    //printf("Starting optimization step...\n");
    mylmdif_ (lmdifError_,
            &m, &n, x, fvec, &ftol, &xtol, &gtol, &maxfev, &epsfcn,
            diag, &mode, &factor, &nprint, &info, &nfev, fjac, &ldfjac,
                ipvt, qtf, wa1, wa2, wa3, wa4);
    double totalerror = 0;
    for (int i=0; i<m; i++) {
       totalerror += fvec[i];
         }
    //printf("\tnum function calls = %i\n", nfev);
    //printf("\tremaining total error value = %f\n", totalerror);
    //printf("\tor %1.2f per point\n", std::sqrt(totalerror) / m);
    //printf("...ended optimization step.\n");

    /* copy result back to parameters array */
        for (int k = 0; k < homographies.size(); ++k) {
                for (int i = 0; i < 3; ++i) {
                        for (int j = 0; j < 3; ++j) {
                                homographies[k].at<double>(i, j) = x[k * 9 + i * 3
                                     + j];
                        }
                }
        }



    /* release allocated workspace */
    free (fvec);
    free (fjac);
    free (wa4);
    free (ipvt);
    free (wa1);
    free (wa2);
    free (wa3);
    free (qtf);
    free (diag);
    free (x);

         return (1);
}
```