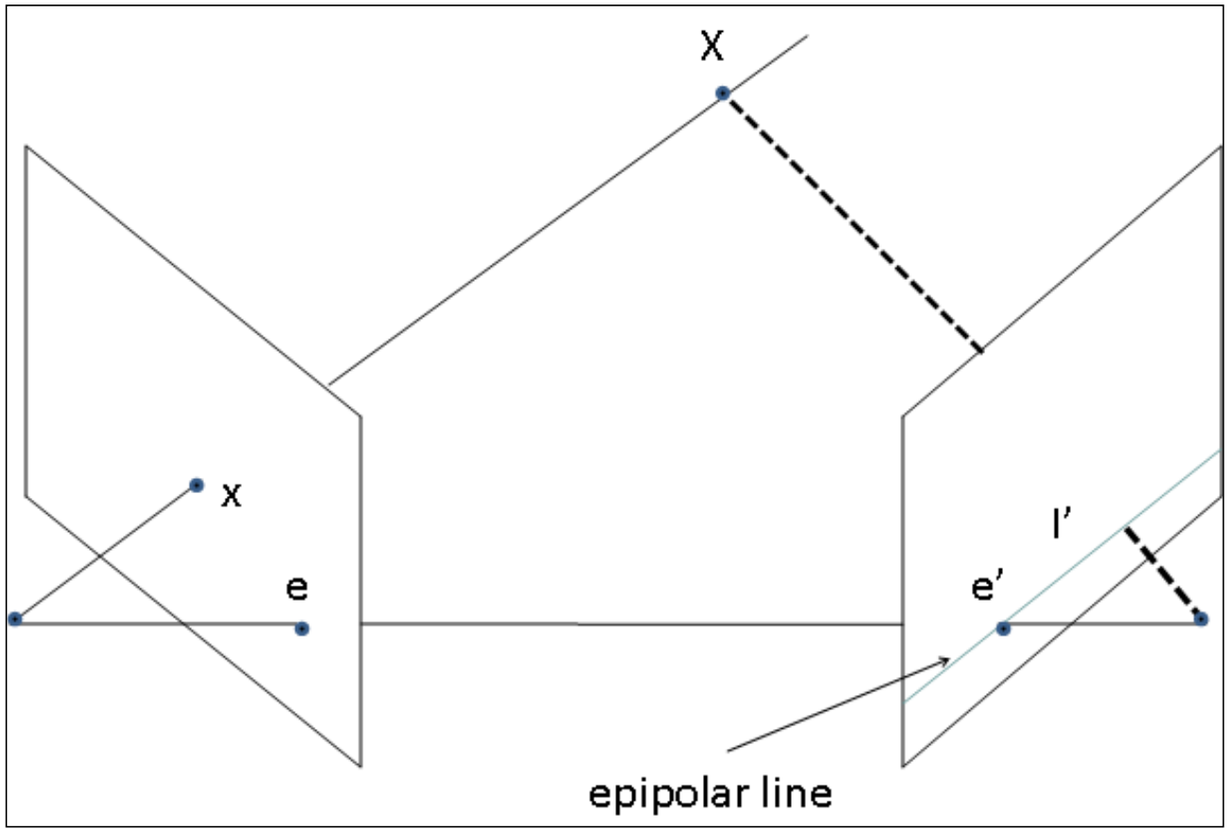
9

Estimating Projective Relations in Images

In this chapter, we will cover:

- Calibrating a camera
- Computing the fundamental matrix of an image pair
- Matching images using random sample consensus
- Computing a homography between two images

Computing the fundamental matrix of an image pair



$$p'^T F p = 0$$

The book by *R. Hartley and A. Zisserman, Multiple View Geometry in Computer Vision, Cambridge University Press, 2004* is the most complete reference on projective geometry in computer vision.

Fundamental Matrix from point correspondences

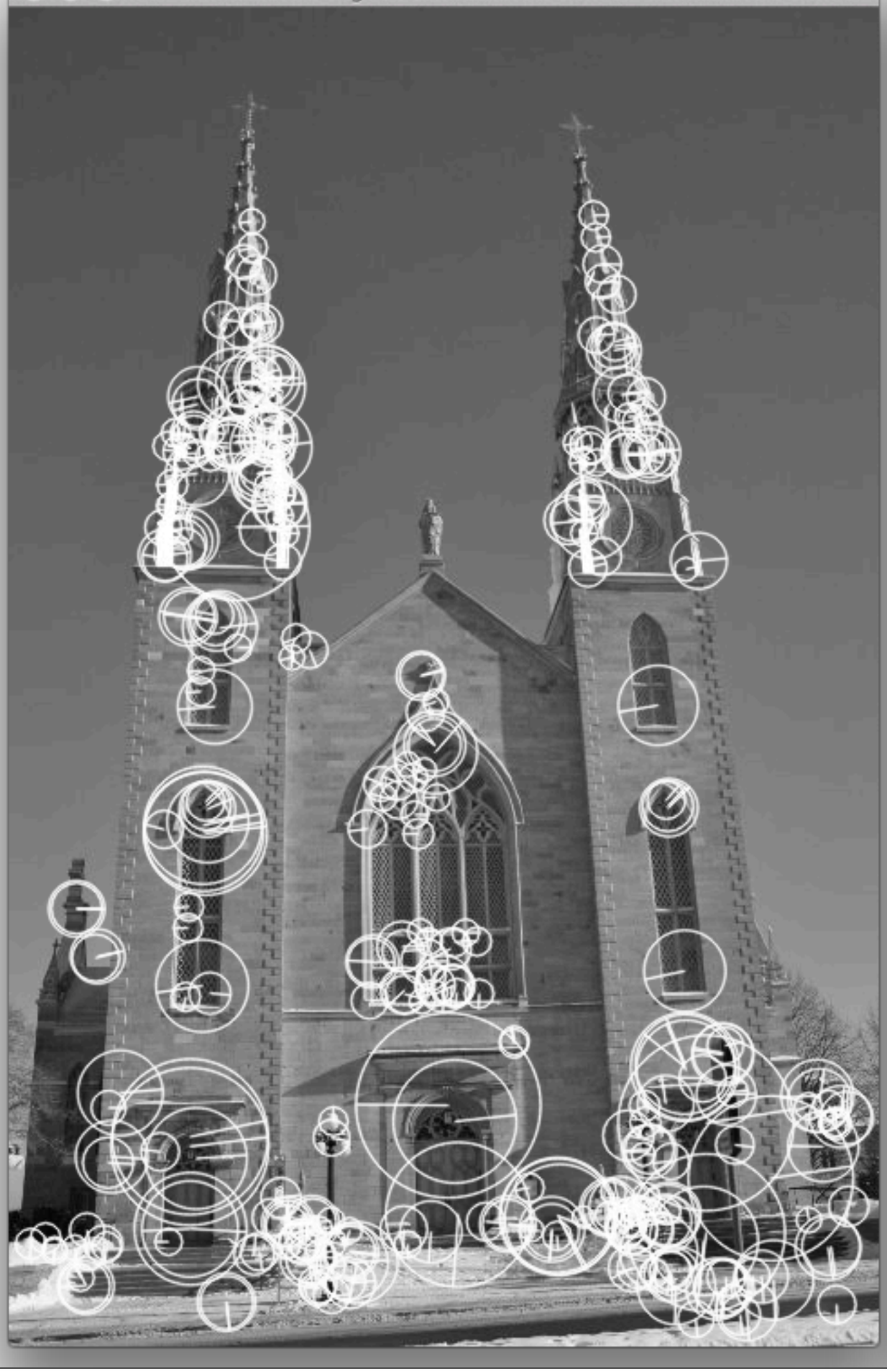
```
std::vector<cv::Point2f> selPoints1, selPoints2;

// Compute F matrix from 7 matches
cv::Mat fundemental = cv::findFundamentalMat(
                                    cv::Mat(selPoints1), // points in first image
                                    cv::Mat(selPoints2), // points in second image
                                    CV_FM_7POINT);       // 7-point method
```

At least, 7 point correspondences are required.

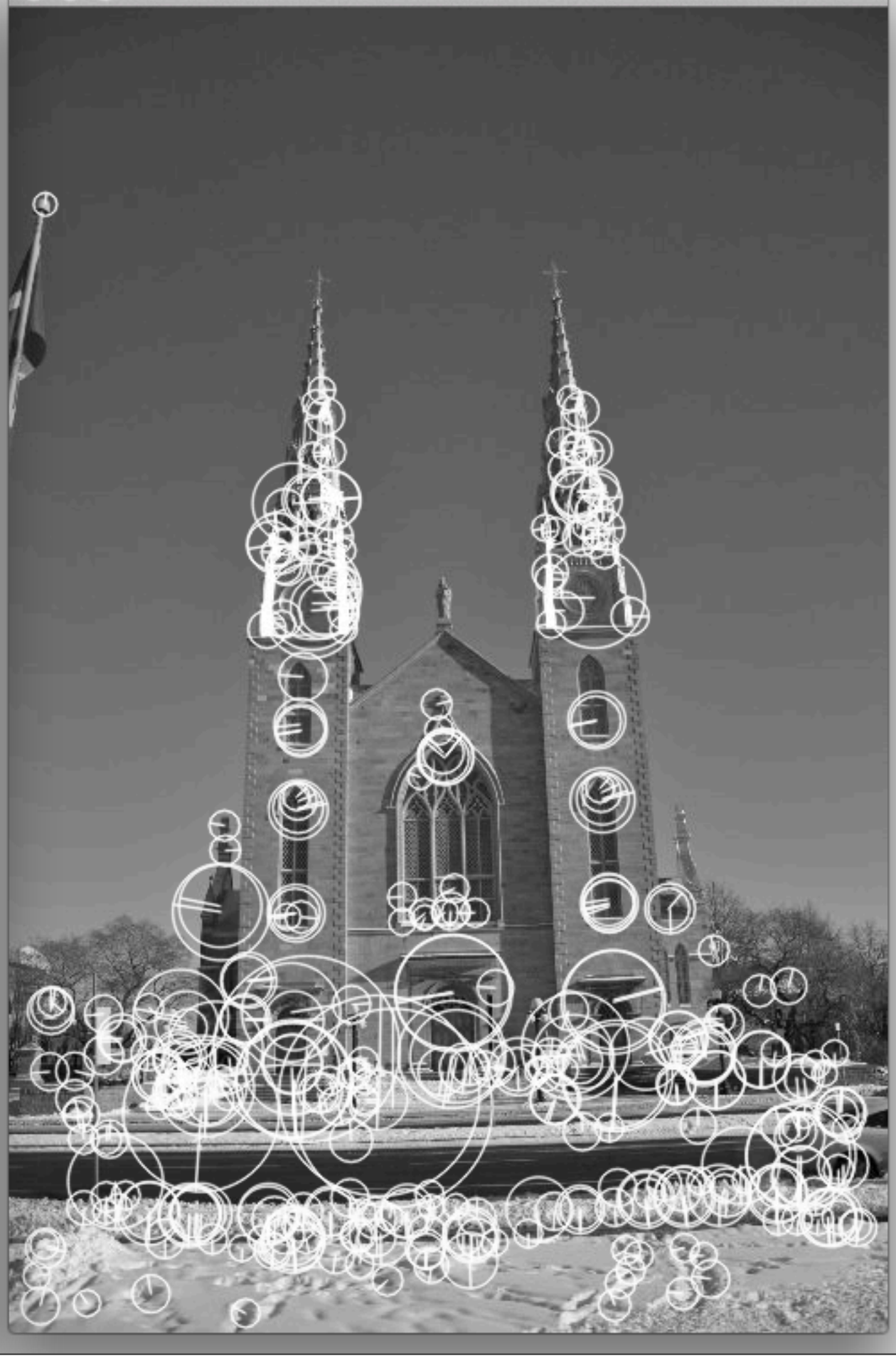
Derivation of the Fundamental Matrix F

```cpp
#include <iostream>
#include <vector>
#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/features2d/features2d.hpp>
#include <opencv2/calib3d/calib3d.hpp>

int main()
{
    // Read input images
    cv::Mat image1= cv::imread("church01.jpg",0);
    cv::Mat image2= cv::imread("church03.jpg",0);
    if (!image1.data || !image2.data)
         return 0;

    // vector of keypoints
    std::vector<cv::KeyPoint> keypoints1;
    std::vector<cv::KeyPoint> keypoints2;

    // Construction of the SURF feature detector
    cv::SurfFeatureDetector surf(3000);

    // Detection of the SURF features
    surf.detect(image1,keypoints1);
    surf.detect(image2,keypoints2);

    std::cout << "Number of SURF points (1): " << keypoints1.size() << std::endl;
    std::cout << "Number of SURF points (2): " << keypoints2.size() << std::endl;

    // Draw the kepoints
    cv::Mat imageKP;
    cv::drawKeypoints(image1,keypoints1,imageKP,cv::Scalar(255,255,255),cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
    cv::namedWindow("Right SURF Features");
    cv::imshow("Right SURF Features",imageKP);
    cv::drawKeypoints(image2,keypoints2,imageKP,cv::Scalar(255,255,255),cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
    cv::namedWindow("Left SURF Features");
    cv::imshow("Left SURF Features",imageKP);
```

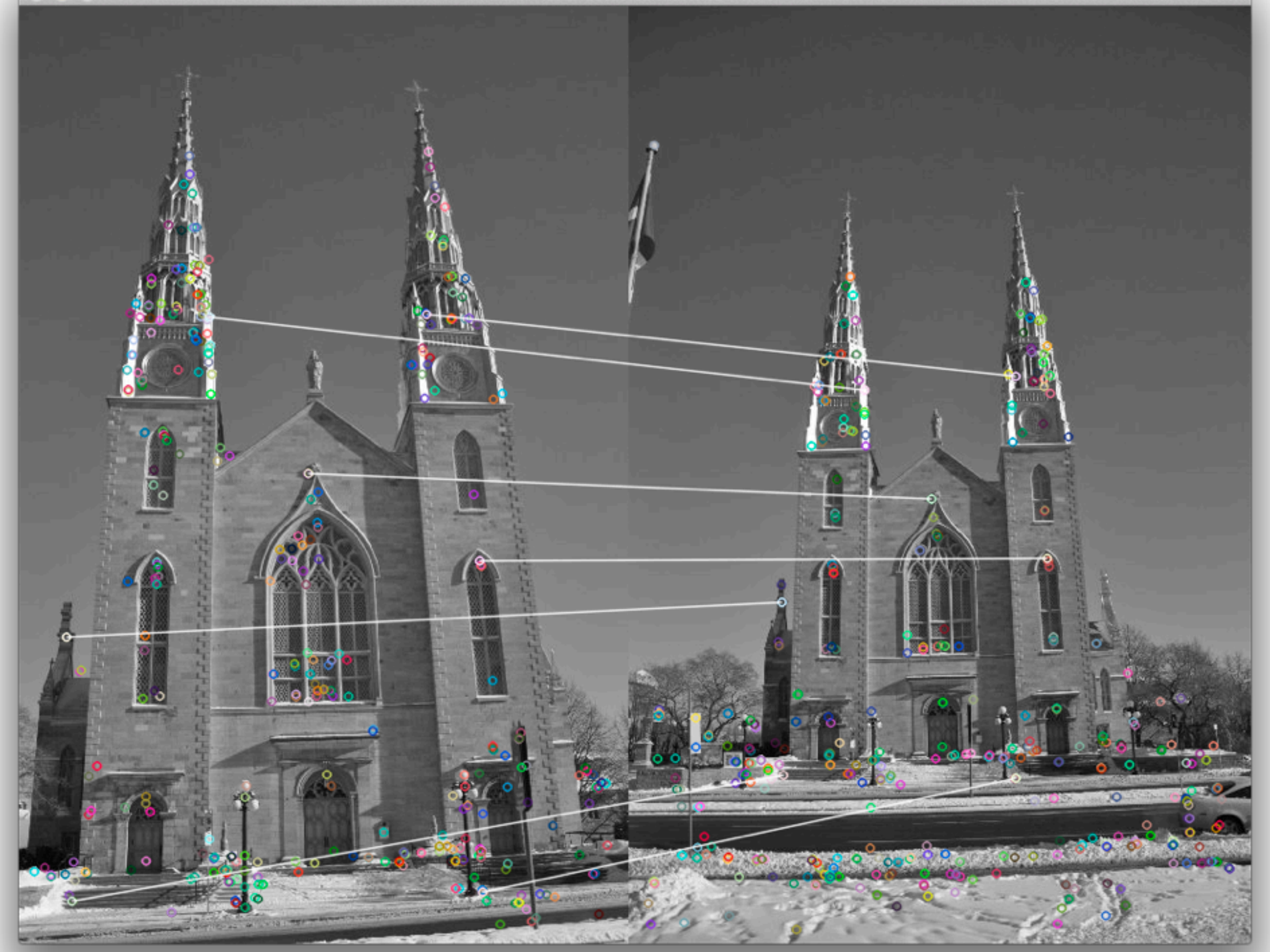
Right SURF Features



Left SURF Features

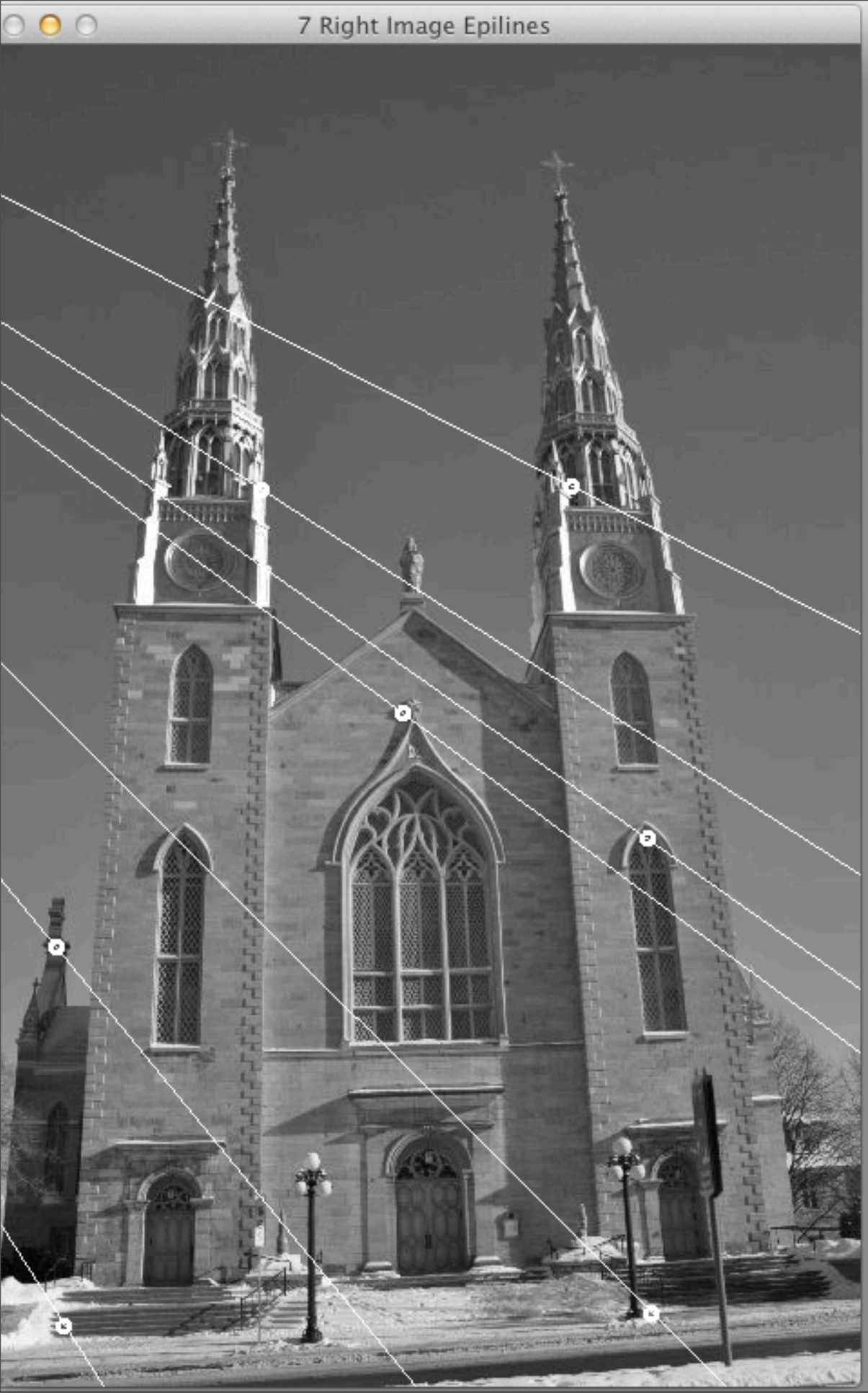


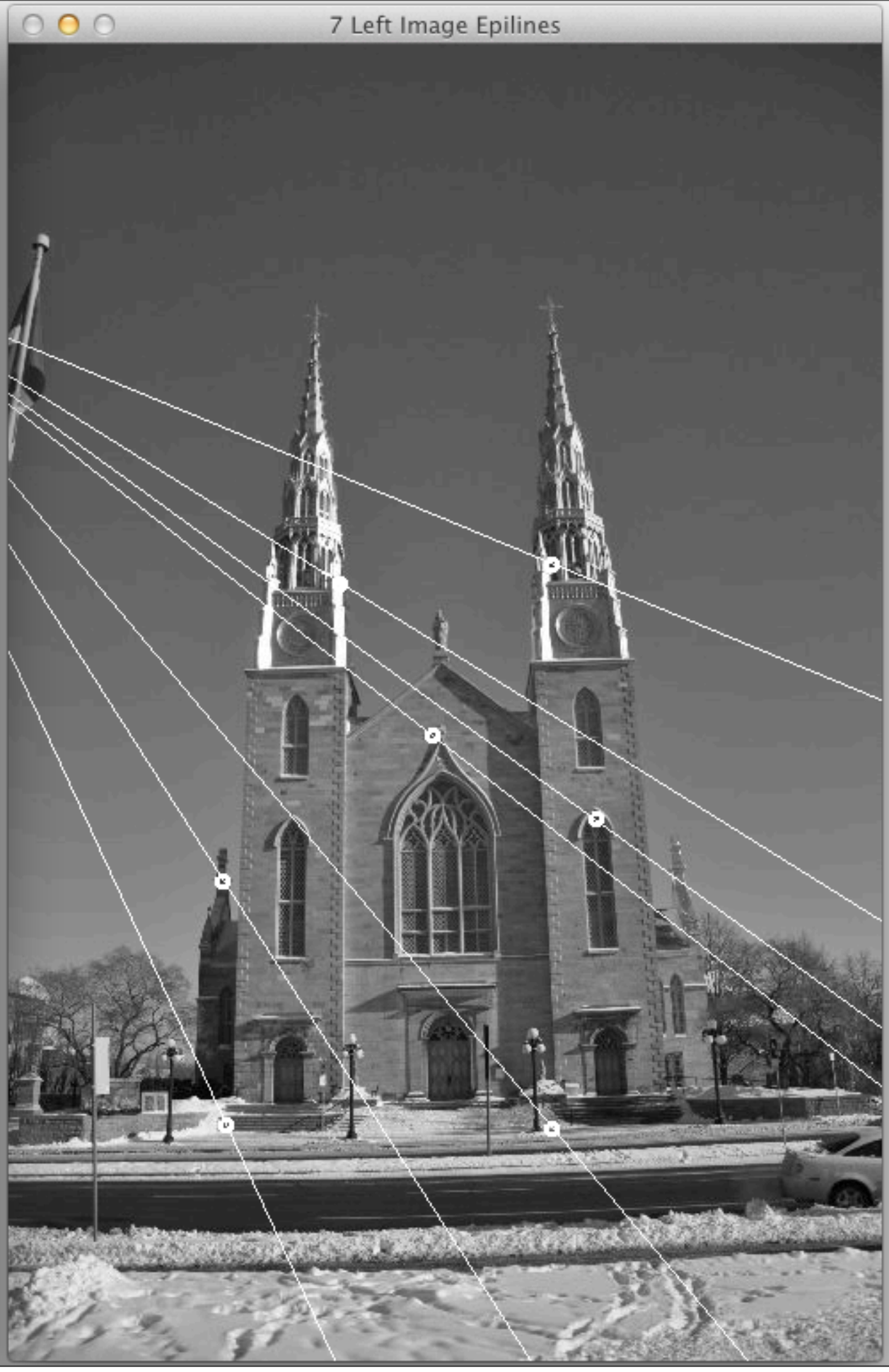
```
// Construction of the SURF descriptor extractor
cv::SurfDescriptorExtractor surfDesc;

// Extraction of the SURF descriptors
cv::Mat descriptors1, descriptors2;
surfDesc.compute(image1,keypoints1,descriptors1);
surfDesc.compute(image2,keypoints2,descriptors2);

std::cout << "descriptor matrix size: " << descriptors1.rows << " by " << descriptors1.cols << std::endl;

// Construction of the matcher
cv::BruteForceMatcher<cv::L2<float> > matcher;

// Match the two image descriptors
std::vector<cv::DMatch> matches;
matcher.match(descriptors1,descriptors2, matches);

std::cout << "Number of matched points: " << matches.size() << std::endl;

// Select few Matches
std::vector<cv::DMatch> selMatches;

selMatches.push_back(matches[14]);
selMatches.push_back(matches[16]);
selMatches.push_back(matches[141]);
selMatches.push_back(matches[146]);
selMatches.push_back(matches[235]);
selMatches.push_back(matches[238]);
selMatches.push_back(matches[274]);

// Draw the selected matches
cv::Mat imageMatches;
cv::drawMatches(image1,keypoints1,  // 1st image and its keypoints
                image2,keypoints2,  // 2nd image and its keypoints
                 selMatches,           // the matches
//                 matches,         // the matches
                 imageMatches,         // the image produced
                 cv::Scalar(255,255,255)); // color of the lines
cv::namedWindow("User Selected Matches");
cv::imshow("User Selected Matches",imageMatches);
```


User Selected Matches

```
// Convert 1 vector of keypoints into
// 2 vectors of Point2f
std::vector<int> pointIndexes1, pointIndexes2;
for (std::vector<cv::DMatch>::const_iterator it= selMatches.begin(); it!= selMatches.end(); ++it)
{
          // Get the indexes of the selected matched keypoints
          pointIndexes1.push_back(it->queryIdx);
          pointIndexes2.push_back(it->trainIdx);
}

// Convert keypoints into Point2f
std::vector<cv::Point2f> selPoints1, selPoints2;
cv::KeyPoint::convert(keypoints1,selPoints1,pointIndexes1);
cv::KeyPoint::convert(keypoints2,selPoints2,pointIndexes2);

// check by drawing the points
std::vector<cv::Point2f>::const_iterator it= selPoints1.begin();
while ( it!=selPoints1.end() ) {
     // draw a circle at each corner location
     cv::circle(image1,*it,3,cv::Scalar(255,255,255),2);
     ++it;
}
it= selPoints2.begin();
while (it!=selPoints2.end()) {
     cv::circle(image2,*it,3,cv::Scalar(255,255,255),2);
     ++it;
}

// Compute F matrix from 7 matches with the 7-point method
cv::Mat fundemental= cv::findFundamentalMat(cv::Mat(selPoints1), cv::Mat(selPoints2), CV_FM_7POINT);
```

```
// draw the left points corresponding epipolar lines in right image
std::vector<cv::Vec3f> lines1;
cv::computeCorrespondEpilines(
     cv::Mat(selPoints1), // image points
     1,                   // in image 1 (can also be 2)
     fundemental, // F matrix
     lines1);     // vector of epipolar lines

// for all epipolar lines
for (std::vector<cv::Vec3f>::const_iterator it= lines1.begin();
      it!=lines1.end(); ++it) {
          // draw the epipolar line between first and last column
          cv::line(image2,cv::Point(0,-(*it)[2]/(*it)[1]),
                          cv::Point(image2.cols,-((*it)[2]+(*it)[0]*image2.cols)/(*it)[1]),
                           cv::Scalar(255,255,255));
}

// draw the left points corresponding epipolar lines in left image
std::vector<cv::Vec3f> lines2;
cv::computeCorrespondEpilines(cv::Mat(selPoints2),2,fundemental,lines2);
for (std::vector<cv::Vec3f>::const_iterator it= lines2.begin();
      it!=lines2.end(); ++it) {

          // draw the epipolar line between first and last column
          cv::line(image1,cv::Point(0,-(*it)[2]/(*it)[1]),
                          cv::Point(image1.cols,-((*it)[2]+(*it)[0]*image1.cols)/(*it)[1]),
                           cv::Scalar(255,255,255));
}

// Display the images with points and epipolar lines
cv::namedWindow("7 Right Image Epilines");
cv::imshow("7 Right Image Epilines",image1);
cv::namedWindow("7 Left Image Epilines");
cv::imshow("7 Left Image Epilines",image2);
```


7 Right Image Epilines


7 Left Image Epilines

C++: Mat **findFundamentalMat**(InputArray **points1**, InputArray **points2**, int **method**=FM_RANSAC, double**param1**=3., double **param2**=0.99, OutputArray **mask**=noArray())

Parameters:

points1 – Array of N points from the first image. The point coordinates should be floating-point (single or double precision).

points2 – Array of the second image points of the same size and format as points1 .

method –

Method for computing a fundamental matrix.

- **CV_FM_7POINT** for a 7-point algorithm. $N = 7$
- **CV_FM_8POINT** for an 8-point algorithm. $N \geq 8$
- **CV_FM_RANSAC** for the RANSAC algorithm. $N \geq 8$
- **CV_FM_LMEDS** for the LMedS algorithm. $N \geq 8$

param1 – Parameter used for RANSAC. It is the maximum distance from a point to an epipolar line in pixels, beyond which the point is considered an outlier and is not used for computing the final fundamental matrix. It can be set to something like 1-3, depending on the accuracy of the point localization, image resolution, and the image noise.

param2 – Parameter used for the RANSAC or LMedS methods only. It specifies a desirable level of confidence (probability) that the estimated matrix is correct.

status – Output array of N elements, every element of which is set to 0 for outliers and to 1 for the other points. The array is computed only in the RANSAC and LMedS methods. For other methods, it is set to all 1's.

The epipolar geometry is described by the following equation:

$$[p_2; 1]^T F [p_1; 1] = 0$$

where F is a fundamental matrix, p_1 and p_2 are corresponding points in the first and the second images, respectively.

The function calculates the fundamental matrix using one of four methods listed above and returns the found fundamental matrix. Normally just one matrix is found. But in case of the 7-point algorithm, the function may return up to 3 solutions (9×3 matrix that stores all 3 matrices sequentially).

C++: void **computeCorrespondEpilines**(InputArray **points**, int **whichImage**, InputArray **F**, OutputArray **lines**)

Parameters:
points – Input points. $N \times 1$ or $1 \times N$ matrix of type `CV_32FC2` or `vector<Point2f>` .
whichImage – Index of the image (1 or 2) that contains the `points` .
F – Fundamental matrix that can be estimated using **findFundamentalMat()** or **stereoRectify()** .
lines – Output vector of the epipolar lines corresponding to the points in the other image. Each line $ax + by + c = 0$ is encoded by 3 numbers (a, b, c) .

For every point in one of the two images of a stereo pair, the function finds the equation of the corresponding epipolar line in the other image.

From the fundamental matrix definition (see **findFundamentalMat()**), line $l_i^{(2)}$ in the second image for the point $p_i^{(1)}$ in the first image (when `whichImage=1`) is computed as:

$$l_i^{(2)} = Fp_i^{(1)}$$

And vice versa, when `whichImage=2`, $l_i^{(1)}$ is computed from $p_i^{(2)}$ as:

$$l_i^{(1)} = F^T p_i^{(2)}$$

Line coefficients are defined up to a scale. They are normalized so that $a_i^2 + b_i^2 = 1$.

Now, computation of F with all the measurements

```
std::vector<cv::Point2f> points1, points2;
for (std::vector<cv::DMatch>::const_iterator it= matches.begin(); it!= matches.end(); ++it) {

    // Get the position of left keypoints
    float x= keypoints1[it->queryIdx].pt.x;
    float y= keypoints1[it->queryIdx].pt.y;
    points1.push_back(cv::Point2f(x,y));
    // Get the position of right keypoints
    x= keypoints2[it->trainIdx].pt.x;
    y= keypoints2[it->trainIdx].pt.y;
    points2.push_back(cv::Point2f(x,y));
}

std::cout << points1.size() << " " << points2.size() << std::endl;

// Compute F matrix using RANSAC
std::vector<uchar> inliers(points1.size(),0);
fundemental= cv::findFundamentalMat(
                                    cv::Mat(points1),cv::Mat(points2), // matching points
                                    inliers,      // match status (inlier ou outlier)
                                    CV_FM_RANSAC, // RANSAC method
                                    1,            // distance to epipolar line
                                    0.98);        // confidence probability
```

Confidence Probability?

The central idea behind the RANSAC algorithm is that the larger the support set is, the higher the probability that the computed matrix is the right one. Obviously, if one (or more) of the randomly selected matches is a wrong match, then the computed fundamental matrix will also be wrong, and its support set is expected to be small. This process is repeated a number of times, and at the end, the matrix with the largest support will be retained as the most probable one.

Therefore, our objective is to pick eight random matches several times so that eventually we select eight good ones which should give us a large support set. Depending on the number of wrong matches in the entire data set, the probability of selecting a set of eight correct matches will differ. We however know that the more selections we make, the higher the confidence will be that we have, among those selections, at least one good match set. More precisely, if we assume that the match set is made of n% inliers (good matches), then the probability that we select eight good matches is 8n. Consequently, the probability that a selection contains at least one wrong match is (1-n8). If we make k selections, the probability of having one random set containing only good matches is 1-(1-8n)k. This is the confidence probability *c*, and we want this probability to be as high as possible since we need at least one good set of matches in order to obtain the correct fundamental matrix. Therefore, when running the RANSAC algorithm, one needs to determine the number of selection k that needs to be made in order to obtain a given confidence level.

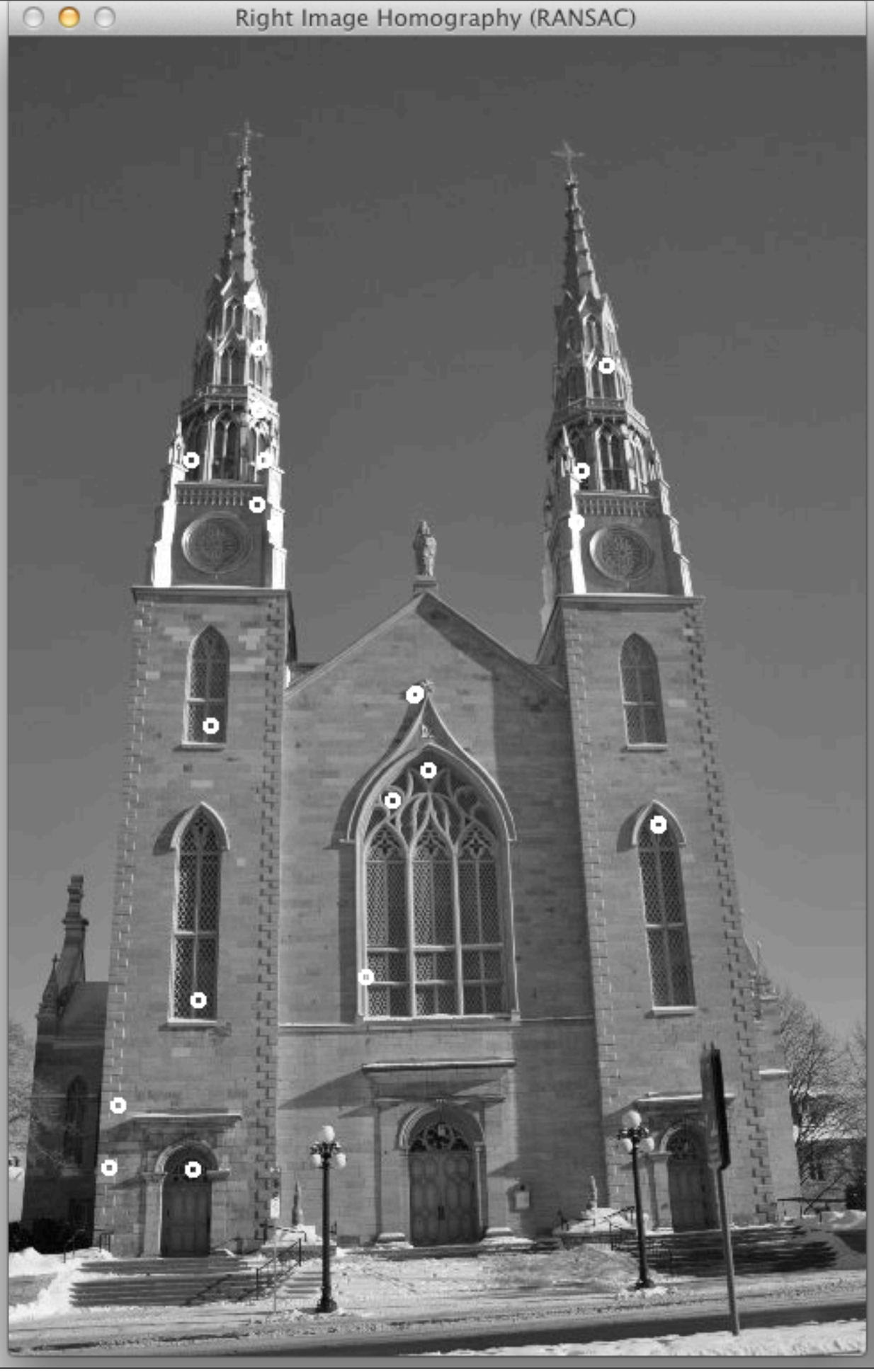
```
// Draw the epipolar line of few points
cv::computeCorrespondEpilines(cv::Mat(selPoints1),1,fundemental,lines1);

for (std::vector<cv::Vec3f>::const_iterator it= lines1.begin(); it!=lines1.end(); ++it) {

    cv::line(image2,cv::Point(0,-(*it)[2]/(*it)[1]),
             cv::Point(image2.cols,-((*it)[2]+(*it)[0]*image2.cols)/(*it)[1]),
             cv::Scalar(255,255,255));
}

cv::computeCorrespondEpilines(cv::Mat(selPoints2),2,fundemental,lines2);

for (std::vector<cv::Vec3f>::const_iterator it= lines2.begin(); it!=lines2.end(); ++it) {

    cv::line(image1,cv::Point(0,-(*it)[2]/(*it)[1]),
             cv::Point(image1.cols,-((*it)[2]+(*it)[0]*image1.cols)/(*it)[1]),
             cv::Scalar(255,255,255));
}
```

```
// Draw the inlier points
std::vector<cv::Point2f> points1In, points2In;
std::vector<cv::Point2f>::const_iterator itPts= points1.begin();
std::vector<uchar>::const_iterator itIn= inliers.begin();

while (itPts!=points1.end()) {
     // draw a circle at each inlier location
     if (*itIn) {
          cv::circle(image1,*itPts,3,cv::Scalar(255,255,255),2);
          points1In.push_back(*itPts);
     }
     ++itPts;
     ++itIn;
}

itPts= points2.begin();
itIn= inliers.begin();
while (itPts!=points2.end()) {
     // draw a circle at each inlier location
     if (*itIn) {
          cv::circle(image2,*itPts,3,cv::Scalar(255,255,255),2);
          points2In.push_back(*itPts);
     }
     ++itPts;
     ++itIn;
}

// Display the images with points
cv::namedWindow("Right Image Epilines (RANSAC)");
cv::imshow("Right Image Epilines (RANSAC)",image1);
cv::namedWindow("Left Image Epilines (RANSAC)");
cv::imshow("Left Image Epilines (RANSAC)",image2);
```

Right Image Epilines (RANSAC)

Left Image Epilines (RANSAC)

```
//
// Find Co-planar points among the inliner matches
//
cv::findHomography(cv::Mat(points1In),cv::Mat(points2In),inliers,CV_RANSAC,1.);

// Read input images
image1= cv::imread("church01.jpg",0);
image2= cv::imread("church03.jpg",0);

// Draw the inlier points
itPts= points1In.begin();
itIn= inliers.begin();
while (itPts!=points1In.end()) {

     // draw a circle at each inlier location
     if (*itIn)
          cv::circle(image1,*itPts,3,cv::Scalar(255,255,255),2);

     ++itPts;
     ++itIn;
}

itPts= points2In.begin();
itIn= inliers.begin();
while (itPts!=points2In.end()) {

     // draw a circle at each inlier location
     if (*itIn)
          cv::circle(image2,*itPts,3,cv::Scalar(255,255,255),2);

     ++itPts;
     ++itIn;
}

// Display the images with points
cv::namedWindow("Right Image Homography (RANSAC)");
cv::imshow("Right Image Homography (RANSAC)",image1);
cv::namedWindow("Left Image Homography (RANSAC)");
cv::imshow("Left Image Homography (RANSAC)",image2);
```


Right Image Homography (RANSAC)


Left Image Homography (RANSAC)

C++: Mat **findHomography**(InputArray **srcPoints**, InputArray **dstPoints**,
int **method**=0, double **ransacReprojThreshold**=3, OutputArray **mask**=noArray())

Parameters:

srcPoints – Coordinates of the points in the original plane, a matrix of the type `CV_32FC2` or `vector<Point2f>` .

dstPoints – Coordinates of the points in the target plane, a matrix of the type `CV_32FC2` or a `vector<Point2f>` .

method –
Method used to computed a homography matrix. The following methods are possible:

- **0** - a regular method using all the points
- **CV_RANSAC** - RANSAC-based robust method
- **CV_LMEDS** - Least-Median robust method

ransacReprojThreshold –
Maximum allowed reprojection error to treat a point pair as an inlier (used in the RANSAC method only). That is, if

$$\|\texttt{dstPoints}_i - \texttt{convertPointsHomogeneous}(H * \texttt{srcPoints}_i)\| > \texttt{ransacReprojThreshold}$$

then the point i is considered an outlier. If `srcPoints` and `dstPoints` are measured in pixels, it usually makes sense to set this parameter somewhere in the range of 1 to 10.

status – Optional output mask set by a robust method (`CV_RANSAC` or `CV_LMEDS`). Note that the input mask values are ignored.

The functions find and return the perspective transformation H between the source and the destination planes:

$$s_i \begin{bmatrix} x'_i \\ y'_i \\ 1 \end{bmatrix} \sim H \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

so that the back-projection error

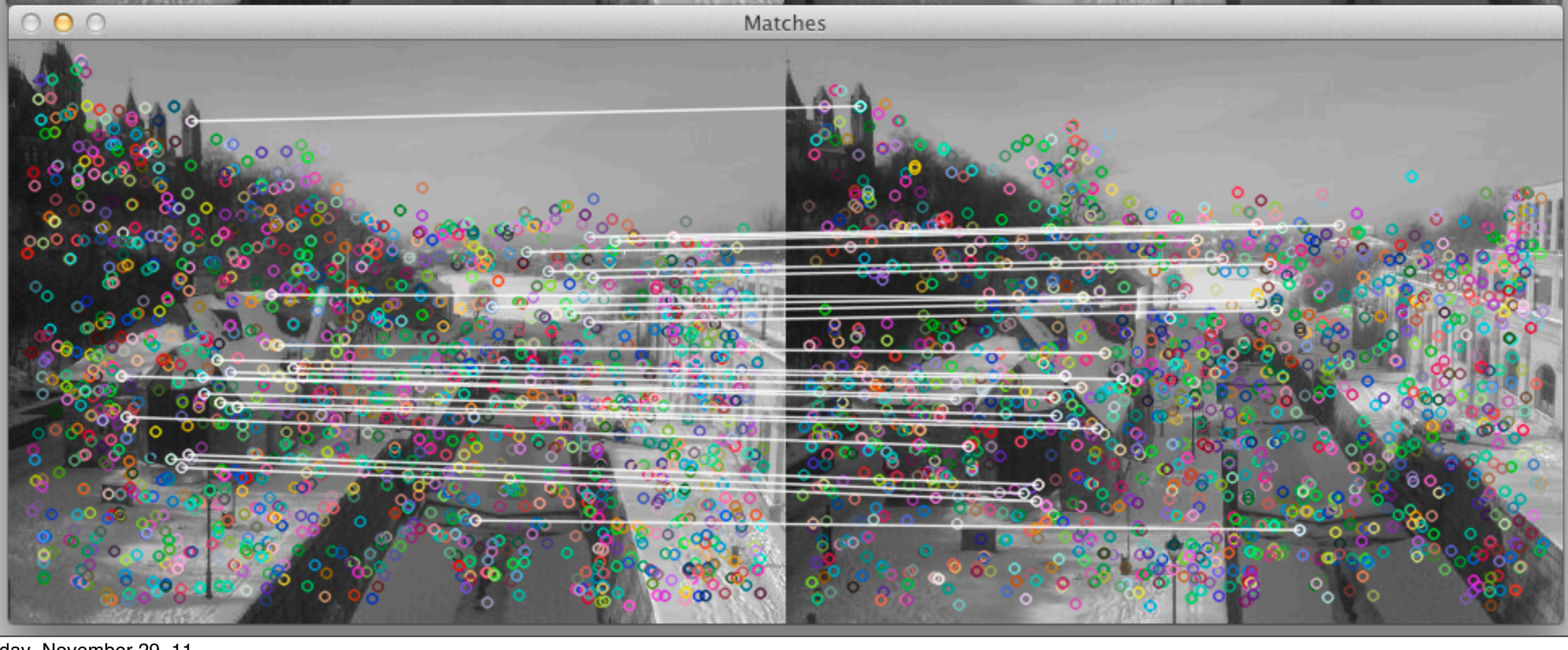
$$\sum_i \left(x'_i - \frac{h_{11} x_i + h_{12} y_i + h_{13}}{h_{31} x_i + h_{32} y_i + h_{33}} \right)^2 + \left(y'_i - \frac{h_{21} x_i + h_{22} y_i + h_{23}}{h_{31} x_i + h_{32} y_i + h_{33}} \right)^2$$

is minimized. If the parameter `method` is set to the default value 0, the function uses all the point pairs to compute an initial homography estimate with a simple least-squares scheme.

```cpp
#include <iostream>
#include <vector>
#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/features2d/features2d.hpp>
#include <opencv2/calib3d/calib3d.hpp>
#include "matcher.h"

int main()
{
    // Read input images
    cv::Mat image1= cv::imread("canal1.jpg",0);
    cv::Mat image2= cv::imread("canal2.jpg",0);
    if (!image1.data || !image2.data)
         return 0;

    // Display the images
    cv::namedWindow("Right Image");
    cv::imshow("Right Image",image1);
    cv::namedWindow("Left Image");
    cv::imshow("Left Image",image2);

    // Prepare the matcher
    RobustMatcher rmatcher;
    rmatcher.setConfidenceLevel(0.98);
    rmatcher.setMinDistanceToEpipolar(1.0);
    rmatcher.setRatio(0.65f);
    cv::Ptr<cv::FeatureDetector> pfd= new cv::SurfFeatureDetector(10);
    rmatcher.setFeatureDetector(pfd);

    // Match the two images
    std::vector<cv::DMatch> matches;
    std::vector<cv::KeyPoint> keypoints1, keypoints2;
    cv::Mat fundemental= rmatcher.match(image1,image2,matches, keypoints1, keypoints2);

    // draw the matches
    cv::Mat imageMatches;
    cv::drawMatches(image1,keypoints1,  // 1st image and its keypoints
                    image2,keypoints2,  // 2nd image and its keypoints
                     matches,         // the matches
                     imageMatches,         // the image produced
                     cv::Scalar(255,255,255)); // color of the lines
    cv::namedWindow("Matches");
    cv::imshow("Matches",imageMatches);
```


Right Image


Left Image


Matches

```
// Convert keypoints into Point2f
std::vector<cv::Point2f> points1, points2;

for (std::vector<cv::DMatch>::const_iterator it= matches.begin();
          it!= matches.end(); ++it) {

          // Get the position of left keypoints
          float x= keypoints1[it->queryIdx].pt.x;
          float y= keypoints1[it->queryIdx].pt.y;
          points1.push_back(cv::Point2f(x,y));
          cv::circle(image1,cv::Point(x,y),3,cv::Scalar(0,255,255),2);
          // Get the position of right keypoints
          x= keypoints2[it->trainIdx].pt.x;
          y= keypoints2[it->trainIdx].pt.y;
          cv::circle(image2,cv::Point(x,y),3,cv::Scalar(255,0,255),2);
          points2.push_back(cv::Point2f(x,y));
}

// Draw the epipolar lines
std::vector<cv::Vec3f> lines1;
cv::computeCorrespondEpilines(cv::Mat(points1),1,fundemental,lines1);

for (std::vector<cv::Vec3f>::const_iterator it= lines1.begin();
          it!=lines1.end(); ++it) {

          cv::line(image2,cv::Point(0,-(*it)[2]/(*it)[1]),
                          cv::Point(image2.cols,-((*it)[2]+(*it)[0]*image2.cols)/(*it)[1]),
                           cv::Scalar(255,255,255));
}

std::vector<cv::Vec3f> lines2;
cv::computeCorrespondEpilines(cv::Mat(points2),2,fundemental,lines2);

for (std::vector<cv::Vec3f>::const_iterator it= lines2.begin();
          it!=lines2.end(); ++it) {

          cv::line(image1,cv::Point(0,-(*it)[2]/(*it)[1]),
                          cv::Point(image1.cols,-((*it)[2]+(*it)[0]*image1.cols)/(*it)[1]),
                           cv::Scalar(255,255,255));
}

// Display the images with epipolar lines
cv::namedWindow("Right Image Epilines (RANSAC)");
cv::imshow("Right Image Epilines (RANSAC)",image1);
cv::namedWindow("Left Image Epilines (RANSAC)");
cv::imshow("Left Image Epilines (RANSAC)",image2);
```


Left Image Epilines (RANSAC)


Right Image Epilines (RANSAC)

```
#if !defined MATCHER
#define MATCHER

#include <vector>
#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/features2d/features2d.hpp>

class RobustMatcher {

private:

    // pointer to the feature point detector object
    cv::Ptr<cv::FeatureDetector> detector;
    // pointer to the feature descriptor extractor object
    cv::Ptr<cv::DescriptorExtractor> extractor;
    float ratio; // max ratio between 1st and 2nd NN
    bool refineF; // if true will refine the F matrix
    double distance; // min distance to epipolar
    double confidence; // confidence level (probability)

public:

    RobustMatcher() : ratio(0.65f), refineF(true), confidence(0.99), distance(3.0) {

        // SURF is the default feature
        detector= new cv::SurfFeatureDetector();
        extractor= new cv::SurfDescriptorExtractor();
    }
```

```
// Match feature points using symmetry test and RANSAC
// returns fundemental matrix
cv::Mat match(cv::Mat& image1, cv::Mat& image2, // input images
              std::vector<cv::DMatch>& matches, // output matches and keypoints
              std::vector<cv::KeyPoint>& keypoints1, std::vector<cv::KeyPoint>& keypoints2) {

    // 1a. Detection of the SURF features
    detector->detect(image1,keypoints1);
    detector->detect(image2,keypoints2);

    // 1b. Extraction of the SURF descriptors
    cv::Mat descriptors1, descriptors2;
    extractor->compute(image1,keypoints1,descriptors1);
    extractor->compute(image2,keypoints2,descriptors2);

    std::cout << "descriptor matrix size: " << descriptors1.rows << " by " << descriptors1.cols << std::endl;

    // 2. Match the two image descriptors

    // Construction of the matcher
    cv::BruteForceMatcher<cv::L2<float> > matcher;

    // from image 1 to image 2
    // based on k nearest neighbours (with k=2)
    std::vector<std::vector<cv::DMatch> > matches1;
    matcher.knnMatch(descriptors1,descriptors2, matches1, 2);        // return 2 nearest neighbours

    // from image 2 to image 1
    // based on k nearest neighbours (with k=2)
    std::vector<std::vector<cv::DMatch> > matches2;
    matcher.knnMatch(descriptors2,descriptors1, matches2, 2);        // return 2 nearest neighbours

    std::cout << "Number of matched points 1->2: " << matches1.size() << std::endl;
    std::cout << "Number of matched points 2->1: " << matches2.size() << std::endl;
```

```
        // 3. Remove matches for which NN ratio is > than threshold

        // clean image 1 -> image 2 matches
        int removed= ratioTest(matches1);
        std::cout << "Number of matched points 1->2 (ratio test) : " << matches1.size()-removed << std::endl;
        // clean image 2 -> image 1 matches
        removed= ratioTest(matches2);
        std::cout << "Number of matched points 1->2 (ratio test) : " << matches2.size()-removed << std::endl;

        // 4. Remove non-symmetrical matches
        std::vector<cv::DMatch> symMatches;
        symmetryTest(matches1,matches2,symMatches);

        std::cout << "Number of matched points (symmetry test): " << symMatches.size() << std::endl;

        // 5. Validate matches using RANSAC
        cv::Mat fundemental= ransacTest(symMatches, keypoints1, keypoints2, matches);

        // return the found fundemental matrix
        return fundemental;
    }
```

```
// Clear matches for which NN ratio is > than threshold
// return the number of removed points
// (corresponding entries being cleared, i.e. size will be 0)
int ratioTest(std::vector<std::vector<cv::DMatch> >& matches) {

     int removed=0;

    // for all matches
     for (std::vector<std::vector<cv::DMatch> >::iterator matchIterator= matches.begin();
          matchIterator!= matches.end(); ++matchIterator) {

        // if 2 NN has been identified
        if (matchIterator->size() > 1) {

            // check distance ratio
            if ((*matchIterator)[0].distance/(*matchIterator)[1].distance > ratio) {

                matchIterator->clear(); // remove match
                removed++;
            }

        } else { // does not have 2 neighbours

            matchIterator->clear(); // remove match
            removed++;
        }
    }

     return removed;
}
```

```
// Insert symmetrical matches in symMatches vector
void symmetryTest(const std::vector<std::vector<cv::DMatch> >& matches1,
                  const std::vector<std::vector<cv::DMatch> >& matches2,
                  std::vector<cv::DMatch>& symMatches) {

    // for all matches image 1 -> image 2
    for (std::vector<std::vector<cv::DMatch> >::const_iterator matchIterator1= matches1.begin();
         matchIterator1!= matches1.end(); ++matchIterator1) {

        if (matchIterator1->size() < 2) // ignore deleted matches
            continue;

        // for all matches image 2 -> image 1
        for (std::vector<std::vector<cv::DMatch> >::const_iterator matchIterator2= matches2.begin();
            matchIterator2!= matches2.end(); ++matchIterator2) {

            if (matchIterator2->size() < 2) // ignore deleted matches
                continue;

            // Match symmetry test
            if ((*matchIterator1)[0].queryIdx == (*matchIterator2)[0].trainIdx  &&
                (*matchIterator2)[0].queryIdx == (*matchIterator1)[0].trainIdx) {

                // add symmetrical match
                symMatches.push_back(cv::DMatch((*matchIterator1)[0].queryIdx,
                                                (*matchIterator1)[0].trainIdx,
                                                (*matchIterator1)[0].distance));
                break; // next match in image 1 -> image 2
            }
        }
    }
}
```

```
cv::Mat ransacTest(const std::vector<cv::DMatch>& matches,
                   const std::vector<cv::KeyPoint>& keypoints1, const std::vector<cv::KeyPoint>& keypoints2,
                   std::vector<cv::DMatch>& outMatches)
{
        // Convert keypoints into Point2f
        std::vector<cv::Point2f> points1, points2;
        for (std::vector<cv::DMatch>::const_iterator it= matches.begin(); it!= matches.end(); ++it) {
            // Get the position of left keypoints
            float x= keypoints1[it->queryIdx].pt.x; float y= keypoints1[it->queryIdx].pt.y;
            points1.push_back(cv::Point2f(x,y));
            // Get the position of right keypoints
            x= keypoints2[it->trainIdx].pt.x; y= keypoints2[it->trainIdx].pt.y;
            points2.push_back(cv::Point2f(x,y));
        }

        // Compute F matrix using RANSAC
        std::vector<uchar> inliers(points1.size(),0);
        cv::Mat fundemental= cv::findFundamentalMat(cv::Mat(points1),cv::Mat(points2), // matching points
                                                    inliers,      // match status (inlier ou outlier)
                                                    CV_FM_RANSAC, // RANSAC method
                                                    distance,     // distance to epipolar line
                                                    confidence);  // confidence probability

        // extract the surviving (inliers) matches
        std::vector<uchar>::const_iterator itIn= inliers.begin();
        std::vector<cv::DMatch>::const_iterator itM= matches.begin();
        // for all matches
        for ( ;itIn!= inliers.end(); ++itIn, ++itM)
            if (*itIn) // it is a valid match
                outMatches.push_back(*itM);

        std::cout << "Number of matched points (after cleaning): " << outMatches.size() << std::endl;

        if (refineF) {  // The F matrix will be recomputed with all accepted matches
            points1.clear(); points2.clear();
            for (std::vector<cv::DMatch>::const_iterator it= outMatches.begin(); it!= outMatches.end(); ++it) {
               float x= keypoints1[it->queryIdx].pt.x; float y= keypoints1[it->queryIdx].pt.y;
               points1.push_back(cv::Point2f(x,y));
               x= keypoints2[it->trainIdx].pt.x; y= keypoints2[it->trainIdx].pt.y;
               points2.push_back(cv::Point2f(x,y));
            }
            // Compute 8-point F from all accepted matches
            fundemental= cv::findFundamentalMat(cv::Mat(points1),cv::Mat(points2), // matching points
                                                CV_FM_8POINT); // 8-point method
        }
        return fundemental;
    }
```