Stereo Vision

Yash Mewada, Pratik Baldota
Project 3 Submission
Submitted in partial fulfillment of the requirements for the course of ***EECE5639***
{mewada.y,baldota.p}@northeastern.edu
April 14, 2023
Northeastern University

***Abstract*—The aim of this project was to find interesting features and correspondences between the left and right images using either the CORNERS and NCC algorithms or SIFT features and descriptors. The results are displayed by connecting corresponding features with different colored lines to make it easier to visualize. A program is also developed to estimate the Fundamental Matrix for each pair using the correspondences above and RANSAC to eliminate outliers. Additionally, a dense disparity map is computed using the Fundamental Matrix to help reduce the search space. The output includes three images: one image with the vertical disparity component, another image with the horizontal disparity component, and a third image representing the disparity vector using color. The direction of the vector is coded by hue, and the length of the vector is coded by saturation. For grayscale display, the disparity values are scaled so that the lowest disparity is 0 and the highest disparity is 255. The results are discussed and sample images are presented in the report.**

***Index Terms*—Harris Corner Detector, NCC, RANSAC, Fundamental Matrix, Disparity Map.**

I. MOTIVATION

A. *Background*

The report should focus on the process of finding correspondences and computing a dense disparity map using the Fundamental Matrix. It should include a detailed explanation of the algorithms used, methodology, and results. Sample images should be provided. The report should also discuss limitations and future scope. Overall, it should provide a comprehensive understanding of the process for computer vision applications.

B. *Approach and description of algorithms*

We explored a total of 4 algorithms in this project. They are stated below.

1) Reading the Images.
2) Detecting Harris corner.
3) Compute normalized cross-correlation and RANSAC.
4) Estimating Fundamental Matrix.
5) Compute Dense Disparity Map.

II. INTRODUCTION

A. *Experiments and Parameters*

The performance of our framework mainly depends on the parameters we used in the stages shown in part 1. Hence, we will give a detailed description of the parameter selection and put a reasonable effort to estimate the best possible parameters.

B. *Reading the Images*

Reading an image in computer vision involves loading an image file into memory as a matrix of pixel values. In C++, OpenCV provides functions like imread() to read images in different formats. Once an image is loaded, it can be processed and analyzed using various techniques such as resizing, filtering, feature extraction, and matching.

C. *Detecting Harris Corners*

Computing the image gradient, obtain the elements of the structure tensor, smooth them, compute the Harris R function for each pixel on corner of the image, a threshold the Harris R function to identify candidate corner points, apply non-maximum suppression, and optionally refine the corner locations using sub-pixel accuracy. For detecting Harris corners, we first need to compute Harris R function with window function, shifted intensity and Intensity

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u,y+v) - I(x,y)]^2 \qquad (1)$$

D. *Computing Normalized Cross Correlation*

In this stage, we first remove all key points near the boundary. Then we choose a 7 × 7 image patch centered at each corner and reshape it as a 25 × 1 feature descriptor. To make it partially invariant to illumination changes, we normalized each descriptor by using if the matrix size is below 7 x 7 matrix it will lose the features where I am the feature descriptor. We compute normalized cross correlation using

$$I(n) = \frac{I(n) - \mu}{(I)}, n = 1, .., 25 \qquad (2)$$

$$NCC = \frac{\sum_{i=1}^{25} x(i)y(i)}{\sqrt{(\sum_{i=1}^{25} x^2 \sum_{i=1}^{25} y^2)}} \qquad (3)$$

Where x is one of the descriptors of the first image and y is one of the descriptors of the second image. Finally, we chose pair of corners such that they have the highest NCC value. Besides, we also set a threshold to keep only matches with a large NCC score.

E. *RANSAC - **RAN**dom **SA**mple **C**onsensus*

Below is the general overview of the RANSAC algorithm. RANSAC is an iterative process of determining the mathemat-

ical model of the data. It is popular because of its ability to work with outliers.
Here the *distance* parameter is generally the Euclidean distance between the predicted and actual point in the data.

- Randomly choose a subset of data points to fit the model (a sample)
- Points within some distance threshold t of the model are a consensus set
- Size of consensus set is model support
- Repeated for N samples; model with the biggest support is the most robust fit

F. Estimating Homography

Homography is a mathematical transformation that maps points in one plane to corresponding points in another. It's commonly used in computer vision and image processing for tasks such as image-stitching and object recognition. To estimate the homography, at least four corresponding points in both planes need to be identified, and a method called Direct Linear Transform (DLT) is used to calculate the homography matrix. The homography matrix can then be used to transform points between the two planes. To apply RANSAC to estimate the homography between two images, the following steps are taken:

- Repeatedly sample 4 points needed to estimate a homography.
- Compute a homography from these four points.
- Map all points using the homography and comparing distances between predicted and observed locations to determine the number of inliers.
- Compute a least-squares homography from all the inliers in the largest set of inliers.

In practice, we computed homography between the randomly sampled points and filtered out the inliers from those points. This whole process was iterated ***1000*** times and that led us to the homography matrix shown in the next section.

$$\begin{bmatrix} wx' \\ wy' \\ w \end{bmatrix} = \begin{bmatrix} h11 & h12 & h13 \\ h21 & h22 & h23 \\ h31 & h32 & h33 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{4}$$

G. Fundamental Matrix

The fundamental matrix works with uncalibrated cameras, while the essential matrix works with calibrated cameras. To estimate the Fundamental Matrix using correspondences and RANSAC, we first need to identify corresponding points in two images. These points can be used to calculate the position and orientation of objects in 3D space. However, not all correspondences will be accurate, and some may be outliers caused by noise, occlusion, or other factors. RANSAC is a robust estimation method that can be used to eliminate outliers and improve the accuracy of the Fundamental Matrix estimation. Once the Fundamental Matrix has been estimated using RANSAC, the inlier correspondences can be displayed in the same way as the original correspondences. These inlier correspondences are the ones that are most likely to be accurate and can be used for further analysis.

- Estimate the fundamental matrix

$$\begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} u_1 \\ v_1 \\ 1 \end{bmatrix}^{\top} \begin{bmatrix} u_2 \\ v_2 \\ 1 \end{bmatrix} = 0 \tag{5}$$

H. Epipolar lines

The term epipolar lines are the lines from the epipole of another camera of the stereo pair to the corresponding points in the image plane. Before rectification, the epipolar lines seem to converge at a point. They solve the stereo constraints of the camera.

I. Computing Dense Disparity

A dense disparity map is an image that shows the shift of pixels in the horizontal axis. The Fundamental matrix can help reduce the search space for matching corresponding points and improve the accuracy of the disparity map. To compute a dense disparity map using the Fundamental matrix, the images are first rectified, the search space is reduced, and the corresponding points are matched using techniques such as SAD, NCC, or SGM. combined.
In this project an inbuilt OpenCV function called SGBM (Semi-Global Block Matching) algorithm was used.

III. Results

As mentioned in the above section, the most crucial parameter that needed to be fine-tuned was threshold selection to obtain the Harris corner detector. We tried different experimental values initially then we implemented the Fundamental Matrix in and Dense Disparity on the image in C++.

IV. Input Images

Fig. 1. Input Image 1

Fig. 2. Input Image 2

In this project 3 sample input images of the building were taken to apply Harris's corner detection, and apply RANSAC followed by calculating Fundamental Matrix with a Dense Disparity Map. The images were rescaled to **75%** of the original image size to reduce the computational time. All the processing was performed on grayscale images.

V. Output Images

A. Corner Detection using harris corner detection algorithm

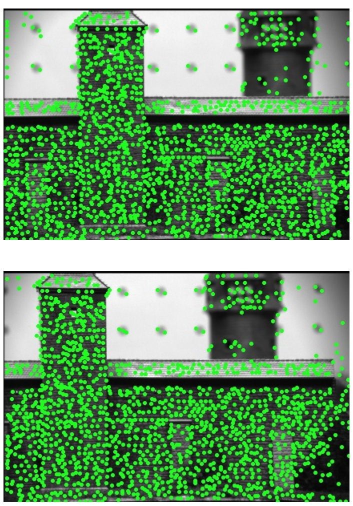

Fig. 3. Output of Harris Corner with Non-max Suppression

For both the images above a threshold of **230** was used. We intentionally kept the threshold higher so that our output is not flooded with the detected corners. For corner response matrix $k = 0.04$ was used.

B. Find correspondences between the images using RANSAC and obtain the inliers

After the corner points were detected, we computed Normalised Cross-Correlation (***NCC***) between the templates of two images in such a way that the detected corner points are at the center of this 7×7 window.
From the above output, we got fewer points than expected from the correspondence it may be due to the image do not contain many distinct points that can be matched.

As a final step we computed homography between all the inliers which resulted in a better and more accurate homography between the images.

$$H = \begin{bmatrix} 1.027 & -0.0038 & -49.7088 \\ 0.0113 & 1.0123 & -1.9848 \\ 7.2 \times 10^{-5} & -8.2 \times 10^{-6} & 1 \end{bmatrix} \quad (6)$$

Homography is a 3x3 matrix that maps corresponding points between two images taken from different viewpoints. It is used for computer vision applications like image stitching, object tracking, and augmented reality. To estimate the homography matrix, a set of corresponding points between the two images is required. This can be obtained through feature matching using NCC or other algorithms, or by manual selection. Once the corresponding points are known, the homography matrix can be computed using methods such as the Direct Linear Transformation (DLT) algorithm or the normalized DLT algorithm.
The reprojection cost for selecting the points to be an inliers was 1.

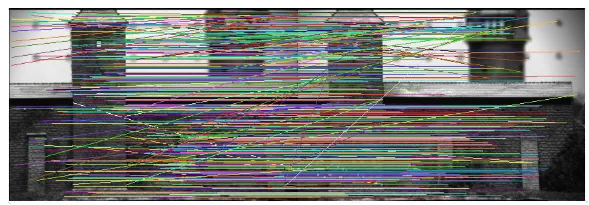

Fig. 4. Pre RANSAC output

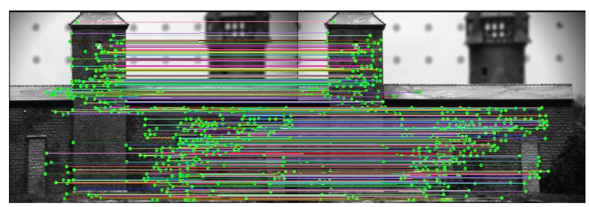

Fig. 5. Post RANSAC output

C. Fundamental Matrix for stereo paired images and rectify the images.

After the inliers are found as explained in the previous section. These inliers were used to estimate the fundamental matrix of the camera.

$$F = \begin{bmatrix} 2.04 \times 10^{-7} & -9.24 \times 10^{-5} & 0.017 \\ 0.0001 & -4.98 \times 10^{-6} & 0.127 \\ -0.0195 & -0.131 & 1 \end{bmatrix} \quad (7)$$

This fundamental matrix was then further used along with the inliers of left and right images to estimate the homography of both images to rectify them. This step is called **stereo image rectification for uncalibrated camera**. Finally, we got two homography matrices for two images which was then applied to the images to rectify them.

Fig. 6. Homography on one image

D. Find the epipolar lines

After the fundamental matrix was obtained it was then used to get the epipolar lines. The fundamental matrix is used to rectify the stereo images which basically moves the image planes in the same line and their optical centers in the same line. This reduces the stereo block matching algorithm complexity to 1D search. After the epipolar lines can be obtained, these lines can be further used for disparity/depth estimation.

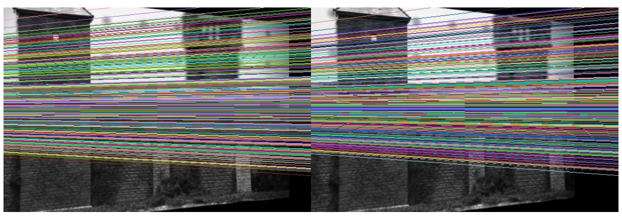

Fig. 7. Epipolar lines

E. Disparity Map

The disparity map is then found using these rectified images. Initially, a disparity map was obtained from the rectified images. Now as the disparity is nothing but a horizontal shift to obtain the vertical component every pixel from every column of this disparity map (horizontal map) was subtracted from the previous pixel of the same column and that gave us the vertical disparity.

$$VP = HP(y,x) - HP(y-1,x) \tag{8}$$

VP = *Vertical Disparity Pixel*
HP = *Horizontal Disparity Pixel*
The vertical disparity is a grayscale normalized image with noise hence you can see the pattern in it.

$$\vec{D} = \arctan\left(HP, VP\right) \tag{9}$$

$$D_{mag} = \sqrt{VP^2 + HP^2} \tag{10}$$

SSD = *Sum of Squared Differences*

$$SSD(win_L, win_R) = \Sigma x \Sigma\ y(Iwin_L(x,y) - Iwin_L(x,y))^2 \tag{11}$$

Window with Minimum SSD = Most Similar/Matching Window.

For disparity vector finding the $\arctan(y,x)$, we use the Fundamental matrix to compute a dense disparity map to reduce the search space for stereo matching. By rectifying the images using the Fundamental matrix, we can ensure that the corresponding points in both images lie on the same horizontal scanline, making it easier to find the disparity between them. The output includes three images: one with the vertical disparity component, another with the horizontal disparity component, and a third image representing the disparity vector using color. The direction of the vector is coded by hue, and the length of the vector is coded by saturation. For grayscale display, we scale the disparity values so that the lowest disparity is 0 and the highest is 255. These images can be used for further analysis and applications.

- Number of Disparities - 48
- Block size for template matching - 20
- Algorithm used here - cv2 inbuilt SBGM.

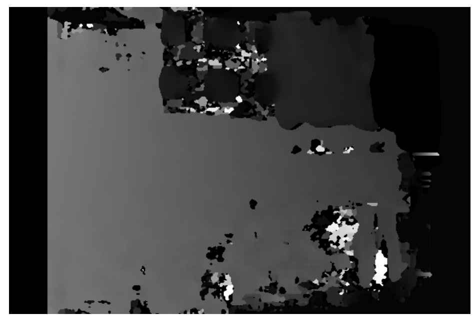

Fig. 8. Horizontal Disparity Component

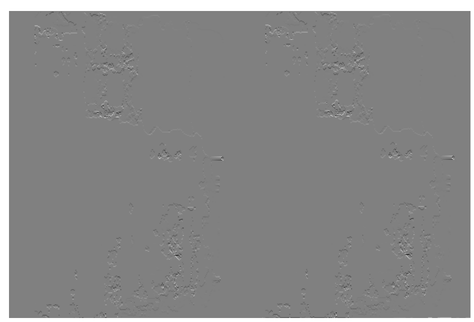

Fig. 9. Vertical Disparity Component

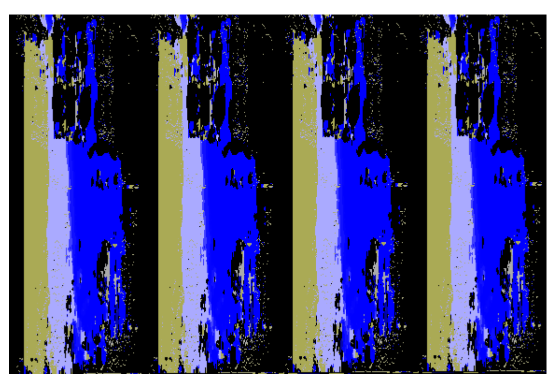

Fig. 10. Disparity map with Hue and saturation

Note: Here due to angle unwrapping the image has been squeezed 4 times in a single image but you can see that the near objects are color coded with a yellowish shade and farther with blueish.
A better disparity map can be generated by changing the number of disparities and block size. Further, the vertical disparity means the shift of pixels in the vertical direction, now was these images were captured by just translating the camera to X-axis the vertical disparity should be zero (a blank image theoretically).

VI. Extras

Below is the output of the experimental input images.

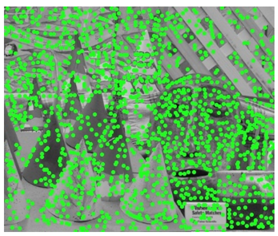

Fig. 11. Harris Corners on grayscale img1

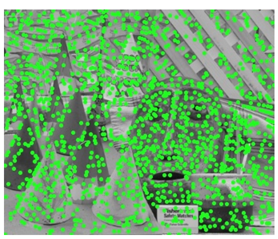

Fig. 12. Harris Corners on grayscale img2

Fig. 13. Correspondences pre RANSAC

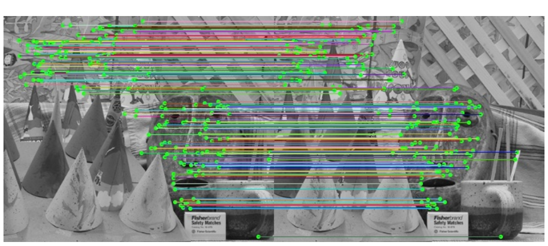

Fig. 14. Correspondences post RANSAC

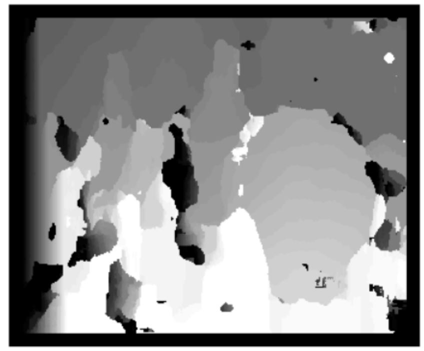

Fig. 15. Horizontal Disparity of Poster Image

- Number of Disparities - 48
- Block size for template matching - 23
- Algorithm used here - cv2 inbuilt SBGM.

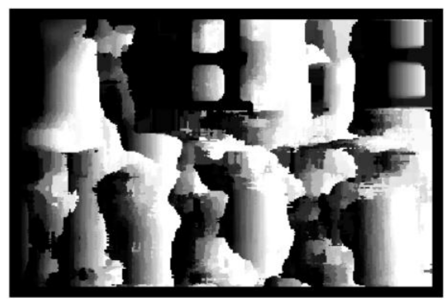

Fig. 16. Horizontal Disparity of Church Image

- Number of Disparities - 48
- Block size for template matching - 25
- Algorithm used here - cv2 inbuilt SBGM.

VII. Observations and conclusions

The main motto behind this project was to develop a framework to understand stereo vision and how depth estimation based on two stereo images work. The term disparity means a horizontal shift in the pixels of respective images. In order to obtain the vertical disparity the same block-matching algorithm can be applied to the columns and we can get the vertical disparity. But as in our case the baseline as pure translation in the x-axis the vertical disparity should be zero or null but due to the noise in images and fractional errors this image never turned out to be a black/Null image.
This report introduces a simple framework for stereo vision and estimating disparity using the Harris corner detector and normalized cross-correlation. We have provided detailed explanations of each stage and recommended specific parameter values for optimal results. The experimental results demonstrate that our framework has good performance, producing impressive panoramic images with accurate alignment and seamless blending. However, to achieve the best results, it is important to consider the limitations and challenges of

the approach, such as computational intensity and sensitivity to illumination and perspective differences, and experimental threshold values. Future research can explore more sophisticated feature extraction and matching techniques. Despite these challenges, the depth estimation algorithm remains a valuable tool in computer vision, robotics and many more.

We also found that the ***safe threshold*** for accurate corresponding points and the RANSAC algorithm is between 230. Anything beyond this resulted in ***bad correspondences post RANSAC*** and anything below this resulted in ***Time complex algorithm***.

- RANSAC Iterations for fundamental matrix inliers = 3000.
- Randomly sampled points to estimate fundamental matrix = 9
- NCC Threshold = 0.9

VIII. Code

The code for our project can be found here : GitHub

IX. Appendix

```cpp
#include <cstdio>
#include <opencv2/opencv.hpp>
#include </usr/include/eigen3/Eigen/Dense>
#include <random>
#include <cmath>


using namespace cv;
using namespace std;
using Eigen::MatrixXd;

/*
    Author - Yash Mewada (mewada.y@northeastern.edu) & Pratik Baldota (baldota.p@northeastern.edu)
    Created - April 8, 2023
*/

/* Class for Image Mosaicing function declaration*/
class imageMosaicing
{
private:
    /* data */
    Mat* lines1 = nullptr;
    Mat* lines2 = nullptr;
public:
    string path_to_images;
    Mat img1;
    Mat img2;
    Mat soble_x = (Mat_<float>(3,3) << -1,0,1,-2,0,2,-1,0,1);
    Mat soble_y = (Mat_<float>(3,3) << -1,-2,-1,0,0,0,1,2,1);
    const int MIN_POINTS = 4;
    const double THRESHOLD = 10;
    const int MAX_ITERATIONS = 3000;
    double ncc_thres = 0.9;
    const int SAMPLE_SIZE = 8;
    const int ERROR_THRES = 1;

    imageMosaicing(string _path);
    double calc_NCC(Mat temp1, Mat temp2);
    vector<pair<Point, Point>> get_correspondences(vector<Point> c1,vector<Point> c2);
    vector<pair<Point, Point>> estimate_homography_ransac(vector<Point> src_points, vector<Point> dst_points);
    vector<Point> harris_detector_for_img1(int thres = 250);
    vector<Point> harris_detector_for_img2(int thres = 250);
    Mat estimateFunMat(Mat A);
    Mat estimateFunndamentalRANSAC(vector<pair<Point,Point>> corees_pts);
    Mat kron(vector<pair<Point,Point>> corees_pts);
    void findEpipolarlines(vector<Point2f> inlierxl,vector<Point2f> inlierxr, Mat F, Mat imgRectR, Mat imgRectL,Mat* lines1,Mat* lines2);

};

imageMosaicing::imageMosaicing(string _path)
{
    cout << "This is a demo for Image Mosaicing code" << endl;
    this->path_to_images = _path;
    img1 = imread(path_to_images + string("building_left.png"),IMREAD_GRAYSCALE);
    img2 = imread(path_to_images + string("building_right.png"),IMREAD_GRAYSCALE);
    resize(img1, img1, Size(), 0.75, 0.75);
    resize(img2, img2, Size(), 0.75, 0.75);
    // cvtColor(img1, img1, COLOR_BGR2GRAY);
    // cvtColor(img2, img2, COLOR_BGR2GRAY);
}

vector<Point> imageMosaicing::harris_detector_for_img1(int thres){
    Mat gradient_x, gx, gxy;
    Mat gradient_y, gy;
    Mat r_norm;
    Mat r = Mat::zeros(img1.size(), CV_32FC1);

    filter2D(img1,gradient_x,CV_32F,soble_x);
    filter2D(img1,gradient_y,CV_32F,soble_y);

    gx = gradient_x.mul(gradient_x);
    gy = gradient_y.mul(gradient_y);
    gxy = gradient_x.mul(gradient_y);

    GaussianBlur(gx,gx,Size(5,5),1.4);
    GaussianBlur(gy,gy,Size(5,5),1.4);
    GaussianBlur(gxy,gxy,Size(5,5),1.4);

    for(int i = 0; i < img1.rows; i++){
        for(int j = 0; j < img1.cols; j++){
            float a = gx.at<float>(i, j);
            float b = gy.at<float>(i, j);
            float c = gxy.at<float>(i, j);
            float det = a*c - b*b;
            float trace = a + c;
            r.at<float>(i,j) = det - 0.04*trace*trace;
        }
    }

    normalize(r, r_norm, 0, 255, NORM_MINMAX, CV_32FC1, Mat());

    Mat corners = Mat::zeros(img1.size(),CV_8UC1);
    vector<Point> corner_coor;
    Mat cr;
    cvtColor(img1,cr,COLOR_GRAY2BGR);
     for (int i = 1; i < r_norm.rows; i++) {
        for (int j = 1; j < r_norm.cols; j++) {
            // Check if current pixel is a local maximum
            if ((int) r_norm.at<float>(i, j) > thres
                && r_norm.at<float>(i, j) > r_norm.at<float>(i - 1, j - 1)
                && r_norm.at<float>(i, j) > r_norm.at<float>(i - 1, j)
                && r_norm.at<float>(i, j) > r_norm.at<float>(i - 1, j + 1)
                && r_norm.at<float>(i, j) > r_norm.at<float>(i, j - 1)
                && r_norm.at<float>(i, j) > r_norm.at<float>(i, j + 1)
                && r_norm.at<float>(i, j) > r_norm.at<float>(i + 1, j - 1)
                && r_norm.at<float>(i, j) > r_norm.at<float>(i + 1, j)
                && r_norm.at<float>(i, j) > r_norm.at<float>(i + 1, j + 1)
                ) {
                corner_coor.push_back({j,i});
                circle(cr, Point(j,i), 1, Scalar(30, 255, 30), 2,8,0);
            }
        }
    }
    cv::imshow("corners",cr);
    //imwrite("cornerimg1.jpg",cr);
```

```cpp
    cv::waitKey(0);
    return corner_coor;
}

vector<Point> imageMosaicing::harris_detector_for_img2(int thres){
    Mat gradient_x, gx2, gxy;
    Mat gradient_y, gy2;
    Mat r_norm;
    Mat r = Mat::zeros(img2.size(), CV_32FC1);
    Mat corners = Mat::zeros(img2.size(),CV_8UC1);
    vector<Point> corner_coor;

    filter2D(img2,gradient_x,CV_32F,soble_x);
    filter2D(img2,gradient_y,CV_32F,soble_y);

    gx2 = gradient_x.mul(gradient_x);
    gy2 = gradient_y.mul(gradient_y);
    gxy = gradient_x.mul(gradient_y);

    GaussianBlur(gx2,gx2,Size(5,5),1.4);
    GaussianBlur(gy2,gy2,Size(5,5),1.4);
    GaussianBlur(gxy,gxy,Size(5,5),1.4);

    for(int i = 0; i < img2.rows; i++){
        for(int j = 0; j < img2.cols; j++){
            float a = gx2.at<float>(i, j);
            float b = gy2.at<float>(i, j);
            float c = gxy.at<float>(i, j);
            float det = a*c - b*b;
            float trace = a + c;
            r.at<float>(i,j) = det - 0.04*trace*trace;
        }
    }
    normalize(r, r_norm, 0, 255, NORM_MINMAX, CV_32FC1, Mat());


    Mat cr;
    cvtColor(img2,cr,COLOR_GRAY2BGR);
     for (int i = 1; i < r_norm.rows; i++) {
        for (int j = 1; j < r_norm.cols; j++) {
            // Check if current pixel is a local maximum
            if ((int) r_norm.at<float>(i, j) > thres
                && r_norm.at<float>(i, j) > r_norm.at<float>(i - 1, j - 1)
                && r_norm.at<float>(i, j) > r_norm.at<float>(i - 1, j)
                && r_norm.at<float>(i, j) > r_norm.at<float>(i - 1, j + 1)
                && r_norm.at<float>(i, j) > r_norm.at<float>(i, j - 1)
                && r_norm.at<float>(i, j) > r_norm.at<float>(i, j + 1)
                && r_norm.at<float>(i, j) > r_norm.at<float>(i + 1, j - 1)
                && r_norm.at<float>(i, j) > r_norm.at<float>(i + 1, j)
                && r_norm.at<float>(i, j) > r_norm.at<float>(i + 1, j + 1)
                ) {
                corner_coor.push_back({j,i});
                circle(cr, Point(j,i), 1, Scalar(30, 255, 30), 2,8,0);
            }
        }
    }
    cv::imshow("corners",cr);
    //imwrite("cornerimg2.jpg",cr);
    cv::waitKey(0);
    return corner_coor;
}

double imageMosaicing::calc_NCC(Mat temp1,Mat temp2){
    double mean1 = 0;
    for(int i=0; i<temp1.rows; i++)
    {
        for(int j=0; j<temp1.cols; j++)
        {
            mean1 += temp1.at<uchar>(i,j);
        }
    }
    mean1 = mean1/(temp1.rows*temp1.cols);
    double mean2 = 0;
    for(int i=0; i<temp2.rows; i++)
    {
        for(int j=0; j<temp2.cols; j++)
        {
            mean2 += temp2.at<uchar>(i,j);
        }
    }
    mean2 = mean2/(temp2.rows*temp2.cols);
    double std1 = 0;
    for(int i=0; i<temp1.rows; i++)
    {
        for(int j=0; j<temp1.cols; j++)
        {
            std1 += pow(temp1.at<uchar>(i,j) - mean1, 2);
        }
    }
    std1 = sqrt(std1/(temp1.rows*temp1.cols));
    double std2 = 0;
    for(int i=0; i<temp2.rows; i++)
    {
        for(int j=0; j<temp2.cols; j++)
        {
            std2 += pow(temp2.at<uchar>(i,j) - mean2, 2);
        }
    }
    std2 = sqrt(std2/(temp2.rows*temp2.cols));
    double ncc = 0;
    // int count = 0;
    for(int i=0; i<temp1.rows; i++)
    {
        for(int j=0; j<temp1.cols; j++)
        {
            ncc += (temp1.at<uchar>(i,j) - mean1)*(temp2.at<uchar>(i,j) - mean2);
            // count++;
        }
    }
    if (std1 > 0 && std2 > 0) {
        ncc = ncc/(temp1.rows*temp1.cols*std1*std2);
    }
    else {
        ncc = 0; // or set to some other default value
    }
    // ncc = ncc/(temp1.rows*temp1.cols*std1*std2);
    return ncc;

}

vector<pair<Point, Point>> imageMosaicing::get_correspondences(vector<Point> c1,vector<Point> c2){
    Mat t1,t2;
    vector<pair<Point,Point>> corres;
    Mat temp_path,temp_path2;
    Point d = Point(0,0);

    for (int i = 0; i < c1.size() ; i++) {
```

```cpp
        double ncc_max = 0;
        Point pt1 = c1[i];
        int p1x = pt1.x - 3;
        int p1y = pt1.y - 3;
        if (p1x < 0 || p1y < 0 || p1x + 7 >= img1.cols || p1y + 7 >= img1.rows){
            continue;
        }
        temp_path = img1(Rect(p1x, p1y, 7, 7));
        d = Point(0,0);
        int maxidx = -1;
        for (int j = 0; j < c2.size(); j++) {
            Point pt2 = c2[j];
            int p2x = pt2.x - 3;
            int p2y = pt2.y - 3;
            if (p2x < 0 || p2y < 0 || p2x + 7 >= img2.cols || p2y + 7 >= img2.rows){
                continue;
            }
            temp_path2 = img2(Rect(p2x,p2y, 7, 7));

            double temp_ncc = calc_NCC(temp_path,temp_path2);
            if (temp_ncc > ncc_max){
                ncc_max = temp_ncc;
                maxidx = j;
            }
        }
        if (c2[maxidx] != Point(0,0) && c1[i] != Point(0,0) && ncc_max > ncc_thres){
            pair<Point,Point> c;
            c.first = c1[i];
            c.second = c2[maxidx];
            // cout << "maxidx" << " ";
            // cout << maxidx << endl;
            corres.push_back(c);
        }
    }
    cout << corres.size() << endl;
    Mat img_matches;
    if (img1.cols != img2.cols) {
        double scale = (double)img1.cols / img2.cols;
        resize(img2, img2, Size(img1.cols, scale*img2.rows));
    }

    // Concatenate images vertically
    // vconcat(img1, img2, img_matches);
    Mat cr2,cr1;
    cvtColor(img2,cr2,COLOR_GRAY2BGR);
    cvtColor(img1,cr1,COLOR_GRAY2BGR);
    hconcat(cr1, cr2, img_matches);
    RNG rng(12345);
    for (int i = 0; i < corres.size() ; i++) {
        Point pt1 =  corres[i].first;
        Point pt2 = Point(corres[i].second.x + img1.cols, corres[i].second.y); // shift the x-coordinate of pt2 to the right of img1
        // Point pt2 = Point(fc[i].second.x, fc[i].second.y + img1.rows);
        Scalar color = Scalar(rng.uniform(0,255), rng.uniform(0, 255), rng.uniform(0, 255));
        line(img_matches, pt1, pt2, color, 1);
    }
    imshow( "result_window", img_matches);
    //imwrite("CorrepondencespreRansac.jpg",img_matches
    cv::waitKey(0);
    return corres;
}

vector<pair<Point, Point>> imageMosaicing::estimate_homography_ransac(vector<Point> src_points, vector<Point> dst_points) {
    vector<pair<Point, Point>> bestCorrespondingPoints;
    int max_inliers = 0;
    // src_points.resize(src_points.size());
    vector<int> best_inliers;
    vector<Point> inliers1;
    vector<Point> inliers2;
    Mat best_homography = Mat::eye(3, 3, CV_64F);
    best_homography =findHomography(src_points,dst_points,RANSAC);
    vector<int> inliers;
    int num_inliers = 0;
    vector<pair<Point, Point>> temp_corres;
    int inlier_idx = -1;
    double mean_dist = 0;
    //    vector<Point2f> curr_inliers;
    for (int j = 0; j < dst_points.size(); j++) {
        Point src_point = src_points[j];
        Point dst_point = dst_points[j];
        Mat src = (Mat_<double>(3, 1) << src_point.x, src_point.y, 1);
        Mat dst = (Mat_<double>(3, 1) << dst_point.x, dst_point.y, 1);
        Mat pred_dst = best_homography * src;
        pred_dst /= pred_dst.at<double>(2, 0);

        double distance = norm(pred_dst-dst);
        // cout << src <<" << norm , manual >> ";SSSSS
        // cout << p << endl;
        cout << distance << endl;
        if (distance < 1) {
            // cout << "got" << endl;
            num_inliers++;
            // inlier_idx = j;
            pair<Point,Point> c;
            c.first = src_point;
            c.second = dst_point;
            temp_corres.push_back(c);
            inliers.push_back(j);
            // cout << num_inliers << endl;
            // curr_inliers.push_back(src_points[j]);
        }
        if (num_inliers >= max_inliers) {
            // cout << "blahh" << endl;
            max_inliers = num_inliers;
            best_inliers = inliers;
            // best_inliers = curr_inliers;
            best_homography = best_homography;
            bestCorrespondingPoints = temp_corres;
            // best_homography = estimateAffinePartial2D(src_points, dst_points, inliers, RANSAC, THRESHOLD);
        }
    }
    // cout << "max_inliers" << "    ";
    // cout << max_inliers << endl;
    vector<Point> inlier_src_points;
    vector<Point> inlier_dst_points;
    for (int i = 0; i < best_inliers.size(); i++) {
        int idx = best_inliers[i];
        inlier_src_points.push_back(src_points[idx]);
        inlier_dst_points.push_back(dst_points[idx]);
        // cout << idx << endl;
    }
    // cout << best_homography << endl;
    best_homography = findHomography(inlier_src_points,inlier_dst_points,RANSAC);
```

```
    cout << "best_homography" << endl;
    cout << best_homography << endl;
    Mat img_matches;
    if (img1.cols != img2.cols) {
        double scale = (double)img1.cols / img2.cols;
        resize(img2, img2, Size(img1.cols, scale*img2.rows));
    }

    // Concatenate images vertically
    // vconcat(img1, img2, img_matches);
    Mat cr2,cr1;
    cvtColor(img2,cr2,COLOR_GRAY2BGR);
    cvtColor(img1,cr1,COLOR_GRAY2BGR);
    RNG rng(12345);
    hconcat(cr1, cr2, img_matches);
    for (int i = 0; i < bestCorrespondingPoints.size() ; i++) {
        Point pt1 = bestCorrespondingPoints[i].first;
        Point pt2 = Point(bestCorrespondingPoints[i].second.x + img1.cols, bestCorrespondingPoints[i].second.y); // shift the x-coordinate of pt2 to the right of img1
        // Point pt2 = Point(fc[i].second.x, fc[i].second.y + img1.rows);
        Scalar color = Scalar(rng.uniform(0,255), rng.uniform(0, 255), rng.uniform(0, 255));
        circle(img_matches, Point(pt1.x,pt1.y), 1, Scalar(30, 255, 30), 2,8,0);
        circle(img_matches, Point(pt2.x,pt2.y), 1, Scalar(30, 255, 30), 2,8,0);
        line(img_matches, pt1, pt2, color, 1);
    }
    imshow( "result_window", img_matches );
    //imwrite("CorrepondencespostRansac.jpg",img_matche
    cv::waitKey(0);
    // visualise_coress(bestCorrespondingPoints);

    return bestCorrespondingPoints;
}

Mat imageMosaicing::kron(vector<pair<Point, Point>> bestCorrespondingPoints){
    Mat kron(bestCorrespondingPoints.size(),9,CV_64F);
    // cout<<bestCorrespondingPoints.size()<<endl;
    // cout<<kron.size()<<endl;
    for (int i = 0; i < bestCorrespondingPoints.size() ; i++) {
        Point2d pt1 = bestCorrespondingPoints[i].first;
        Point2d pt2 = bestCorrespondingPoints[i].second;
        // cout<<pt1.x*pt2.x<<endl;
        Mat row = (Mat_<double>(9,1) << pt1.x*pt2.x, pt1.x*pt2.y,pt1.x,pt1.y*pt2.x,pt1.y*pt2.y, pt1.y, pt2.x,pt2.y,1);
        kron.at<double>(i,0) = row.at<double>(0,0);
        kron.at<double>(i,1) = row.at<double>(0,1);
        kron.at<double>(i,2) = row.at<double>(0,2);
        kron.at<double>(i,3) = row.at<double>(0,3);
        kron.at<double>(i,4) = row.at<double>(0,4);
        kron.at<double>(i,5) = row.at<double>(0,5);
        kron.at<double>(i,6) = row.at<double>(0,6);
        kron.at<double>(i,7) = row.at<double>(0,7);
        kron.at<double>(i,8) = row.at<double>(0,8);
        // Mat kron_row = kron.row(i).setTo(row);
        // row.copyTo(kron_row);
        //  = row;
        // cout << row << endl;
        // cout << kron.row(1) << endl;
    }
    // cout << "[";
    // for (int i = 0; i < kron.rows ; i++){
    //     for (int j = 0; j < kron.cols; j++){
    //         cout << kron.at<double>(i,j) << " ";
    //     }
    //     cout << endl;
    // }
    // cout << " ]"<< endl;
    return kron;
}

Mat imageMosaicing::estimateFunMat(Mat A){
    SVD svdTemp(A,SVD::FULL_UV);

    Mat fUtil = svdTemp.vt.row(8);
    Mat Ftemp(3,3,CV_64FC1);

    Ftemp.at<double>(0,0) = fUtil.at<double>(0,0);
    Ftemp.at<double>(0,1) = fUtil.at<double>(0,1);
    Ftemp.at<double>(0,2) = fUtil.at<double>(0,2);
    Ftemp.at<double>(1,0) = fUtil.at<double>(0,3);
    Ftemp.at<double>(1,1) = fUtil.at<double>(0,4);
    Ftemp.at<double>(1,2) = fUtil.at<double>(0,5);
    Ftemp.at<double>(2,0) = fUtil.at<double>(0,6);
    Ftemp.at<double>(2,1) = fUtil.at<double>(0,7);
    Ftemp.at<double>(2,2) = fUtil.at<double>(0,8);
    // cout << "Old F = [";
    // for (int i = 0; i < Ftemp.rows ; i++){
    //     for (int j = 0; j < Ftemp.cols; j++){
    //         cout << Ftemp.at<double>(i,j) << " ";
    //     }
    //     cout << endl;
    // }
    // cout << " ]"<< endl;
    Mat F(3,3,Ftemp.type());
    SVD svd(Ftemp,SVD::FULL_UV);
    // cout << svd.u.size() << endl;
    // cout << svd.vt.t().size() << endl;
    // cout << svd.w.size() << endl;
    svd.w.at<double>(0,2) = 0;
    Mat w(3,3,CV_64F);
    w.at<double>(0,0) = svd.w.at<double>(0,0);
    w.at<double>(0,1) = 0;
    w.at<double>(0,2) = 0;
    w.at<double>(1,0) = 0;
    w.at<double>(1,1) = svd.w.at<double>(0,1);
    w.at<double>(1,2) = 0;
    w.at<double>(2,0) = 0;
    w.at<double>(2,1) = 0;
    w.at<double>(2,2) = 0;
    F = svd.u*w*svd.vt;
    // cout << "New F = [";
    // for (int i = 0; i < Ftemp.rows ; i++){
    //     for (int j = 0; j < Ftemp.cols; j++){
    //         cout << Ftemp.at<double>(i,j) << " ";
    //     }
    //     cout << endl;
    // }
    // cout << " ]"<< endl;
    Ftemp.at<double>(3,3) = 1;

    return Ftemp;



}

Mat imageMosaicing::estimateFunndamentalRANSAC(vector<pair<Poin corees_pts){
    Mat bestF;

```

```
    int bestInliercnt = 0;
    vector<Point> xl,xr;
    for (int i = 0; i < corees_pts.size(); i++) {
        xl.push_back(corees_pts[i].first);
        xr.push_back(corees_pts[i].second);
    }
    // Get 8 Random points
    for(int i=0;i<MAX_ITERATIONS;i++){
        vector<Point2f> sampxl,sampxr;

    random_shuffle(corees_pts.begin(),corees_pts.end());
        for(int j =0;j<SAMPLE_SIZE;j++){
            sampxl.push_back(xl[j]);
            sampxr.push_back(xr[j]);
        }
        Mat F =
    findFundamentalMat(sampxl,sampxr,FM_8POINT);
        int inlierCount = 0;
        for (int k = 0; k < xl.size(); k++) {
            // Mat x1(xl[i].);
            // Mat x2(xr);
            double error = abs(xr[k].x *
    F.at<double>(0,0) * xl[k].x +
                                   xr[k].x *
    F.at<double>(0,1) * xl[k].y +
                                   F.at<double>(0,2) *
    xl[k].x +
                                   xr[k].y *
    F.at<double>(1,0) * xl[k].x +
                                   xr[k].y *
    F.at<double>(1,1) * xl[k].y +
                                   F.at<double>(1,2) *
    xl[k].y +
                                   F.at<double>(2,0) *
    xl[k].x +
                                   F.at<double>(2,1) *
    xl[k].y +
                                   F.at<double>(2,2));
            // cout << error << endl;
            if (error < ERROR_THRES){
                inlierCount++;
                // cout << "error < 1" << endl;
            }
        }
        if (inlierCount > bestInliercnt){
            bestInliercnt = inlierCount;
            bestF = F;
        }
    }
    vector<Point> inlierxL,inlierxR;
    for (int k = 0; k < xl.size(); k++) {
            // Mat x1(xl[i].);
            // Mat x2(xr);
            double error = abs(xr[k].x *
    bestF.at<double>(0,0) * xl[k].x +
                                   xr[k].x *
    bestF.at<double>(0,1) * xl[k].y +

    bestF.at<double>(0,2) * xl[k].x +
                                   xr[k].y *
    bestF.at<double>(1,0) * xl[k].x +
                                   xr[k].y *
    bestF.at<double>(1,1) * xl[k].y +

    bestF.at<double>(1,2) * xl[k].y +

    bestF.at<double>(2,0) * xl[k].x +

    bestF.at<double>(2,1) * xl[k].y +

    bestF.at<double>(2,2));
            if (error < ERROR_THRES){
                inlierxL.push_back(xl[k]);
```

```
                inlierxR.push_back(xr[k]);
                // cout << "error < 1" << endl;
            }
        }
    bestF =
    findFundamentalMat(inlierxL,inlierxR,FM_8POINT);
    return bestF;

}

void
    imageMosaicing::findEpipolarlines(vector<Point2f>
    inlierxl,vector<Point2f> inlierxr, Mat F, Mat
    imgRectR, Mat imgRectL,Mat* lines1,Mat* lines2){
    // Mat lines1;
    Mat color1;
    cvtColor(imgRectR,color1,COLOR_GRAY2BGR);

    computeCorrespondEpilines(cv::Mat(inlierxl),1,F,*lines1);
    RNG rng(12345);
    for (int i = 0; i < lines1->rows ; i++){
        for (int j = 0; j < lines1->cols; j++){
            Scalar color =
    Scalar(rng.uniform(0,255), rng.uniform(0, 255),
    rng.uniform(0, 255));

    line(color1,Point(0,-(lines1->at<Vec3f>(i,j))[2]/
    lines1->at<Vec3f>(i,j)[1]),Point(imgRectR.cols,-(lines1->at
    + lines1->at<Vec3f>(i,j)[0] * imgRectR.cols)/
    lines1->at<Vec3f>(i,j)[1]),color);
        }
    }


    // Mat lines2;
    Mat color2;
    cvtColor(imgRectL,color2,COLOR_GRAY2BGR);

    computeCorrespondEpilines(cv::Mat(inlierxr),2,F,*lines2);
    for (int i = 0; i < lines2->rows ; i++){
        for (int j = 0; j < lines2->cols; j++){
            Scalar color =
    Scalar(rng.uniform(0,255), rng.uniform(0, 255),
    rng.uniform(0, 255));

    line(color2,Point(0,-(lines2->at<Vec3f>(i,j))[2]/
    lines2->at<Vec3f>(i,j)[1]),Point(imgRectL.cols,-(lines2->at
    + lines2->at<Vec3f>(i,j)[0] * imgRectL.cols)/
    lines2->at<Vec3f>(i,j)[1]),color);
        }
    }
    Mat epipolarLines;
    hconcat(color1,color2,epipolarLines);
    imshow("Epipolarlines",epipolarLines);
    //imwrite("Epipolarlines.jpg",epipolarLines);
    waitKey(0);

    // return lines1,lines2;
}



int main(){
    string path =
    "/home/yash/Documents/Computer_VIsion/CV_Project3/CV_Projec
    Mat imgL = imread(path +
    string("building_left.png"),IMREAD_GRAYSCALE);
    Mat imgR = imread(path +
    string("building_right.png"),IMREAD_GRAYSCALE);

    imageMosaicing p3(path);
    vector<Point> cor_img1,cor_img2;

    cor_img1 = p3.harris_detector_for_img1(230);
```

```
    cor_img2 = p3.harris_detector_for_img2(230);

    vector<pair<Point,Point>> corres;
    corres = p3.get_correspondences(cor_img1,cor_img2);
    cout << "done" << endl;

    vector<Point> src,dst;
    vector<DMatch> matches;
    for (int i = 0; i < corres.size(); i++) {
        src.push_back(corres[i].first);
        dst.push_back(corres[i].second);
    }
    vector<pair<Point,Point>> inliers;

    inliers = p3.estimate_homography_ransac(src,dst);
    Mat A = p3.kron(inliers);
    Mat svdF = p3.estimateFunMat(A);

    Mat F = p3.estimateFunndamentalRANSAC(inliers);
    cout << "Ransac F = [";
    for (int i = 0; i < F.rows ; i++){
        for (int j = 0; j < F.cols; j++){
            cout << F.at<double>(i,j) << " ";
        }
        cout << endl;
    }
    cout << " ]"<< endl;

    cout << "SVD F = [";
    for (int i = 0; i < svdF.rows ; i++){
        for (int j = 0; j < svdF.cols; j++){
            cout << svdF.at<double>(i,j) << " ";
        }
        cout << endl;
    }
    cout << " ]"<< endl;


    vector<Point2f> inlierxl,inlierxr;
    for (int i = 0; i < inliers.size(); i++) {
        inlierxl.push_back(inliers[i].first);
        inlierxr.push_back(inliers[i].second);
    }
    // Rectify the images
    Mat H1, H2;
    stereoRectifyUncalibrated(inlierxl,inlierxr,F,imgL.
    Mat imgRectL, imgRectR,rectimgs;
    warpPerspective(imgL, imgRectL, H1, imgL.size());
    warpPerspective(imgR, imgRectR, H2, imgR.size());
    hconcat(imgRectL,imgRectR,rectimgs);
    imshow("rectL",rectimgs);
    //imwrite("Rectified Images.jpg",rectimgs);
    waitKey(0);
    Mat epi1,epi2;
    p3.findEpipolarlines(inlierxl,inlierxr,F,imgR,imgL,

   Ptr<StereoSGBM> stereo = StereoSGBM::create(0, 16, 3);

    // Compute disparity map
    Mat disp; Mat dispor;
    cv::Mat disp_x, disp_y; // horizontal and vertical disparity maps
    stereo->compute(imgRectL, imgRectR, disp);
    cv::Sobel(disp, disp_x, CV_32F, 1, 0);
    cv::Sobel(disp, disp_y, CV_32F, 0, 1);

    // Scale the disparity maps to the desired range
    double min_disp, max_disp;
    cv::minMaxLoc(disp, &min_disp, &max_disp);
    disp_x = 255 * (disp_x - min_disp) / (max_disp - min_disp);
    disp_y = 255 * (disp_y - min_disp) / (max_disp - min_disp);

    // Convert the disparity maps to 8-bit unsigned integers
    disp_x.convertTo(disp_x, CV_8U);
    disp_y.convertTo(disp_y, CV_8U);

    Mat vert_disp(disp.size(), CV_32F);
    Mat hor_disp(disp.size(), CV_32F);
    Mat disp_vec(disp.size(), CV_32F);
    for (int i =0;i<disp.rows;i++){
        for (int j =0;j<disp.cols;j++){
            Vec3f epiline = epi1.at<Vec3f>(i,j);
            if ((abs(epi1.at<Vec3f>(i,j)[1])) > (abs(epi1.at<Vec3f>(i,j)[0]))){
                hor_disp.at<uchar>(i,j) = 0.0;
                vert_disp.at<uchar>(i,j) = disp.at<uchar>(i,j)/(abs(epiline[1]));
                cout << "vertical" << endl;
            }
            else{
                vert_disp.at<uchar>(i,j) = 0.0;
                hor_disp.at<uchar>(i,j) = disp.at<uchar>(i,j)/(abs(epiline[0]));
            }
        }
    }
    Mat dispVec(disp.size(), CV_32FC3);

    for (int i = 0; i < disp.rows; i++) {
    for (int j = 0; j < disp.cols; j++) {
        Point2f pt(j, i);
        Mat pt1 = (Mat_<double>(3,1) << pt.x, pt.y, 1);
        Mat pt2 = F * pt1;
        float x = pt2.at<double>(0,0);
        float y = pt2.at<double>(1,0);
        float z = pt2.at<double>(2,0);
        float disp_val = disp.at<uchar>(i, j);
        if (z != 0.0) {
            float vert_disp_val = disp_val * y / z;
            float hor_disp_val = disp_val * x / z;
            vert_disp.at<uchar>(i, j) = vert_disp_val;
            hor_disp.at<uchar>(i, j) = hor_disp_val;
            float hue = atan2(y, x) * 180 / CV_PI / 2 + 0.5;
            float saturation = sqrt(x * x + y * y) / z * 255;
            Vec3b color(saturation, saturation, saturation);
            color.val[0] = hue;
            disp_vec.at<Vec3b>(i, j) = color;
        }
    }
}
    // cvtColor(disp, disp_vec, COLOR_GRAY2BGR);

    // Mat dispVec(disp.size(), CV_32FC3);
    // for (int y = 0; y < disp.rows; y++) {
    //     for (int x = 0; x < disp.cols; x++) {
    //         float dx = disp_x.at<float>(y, x);
    //         float dy = disp_y.at<float>(y, x);
    //         float mag = sqrt(dx*dx + dy*dy);
    //         float angle = atan2(dy, dx);
    //         if (mag > 0) {
    //             angle = (angle + CV_PI) / (2*CV_PI);
```

```
    //              mag = std::min(mag / 32.0f, 1.0f);
    //              dispVec.at<Vec3f>(y, x) = Vec3f(angle, mag, mag);
    //          }
    //          else {
    //              dispVec.at<Vec3f>(y, x) = Vec3f(0, 0, 0);
    //          }
    //      }
    // }

    // Display the horizontal and vertical disparity maps separately or combine them to visualize the disparity vectors using color.
    cv::imshow("Horizontal Disparity", hor_disp);
    cv::imshow("Vertical Disparity", vert_disp);
    // imwrite("Horizontal_Disparity.png",disp_x);
    // imwrite("Vertical_Disparity.png",disp_y);
    imshow("Disparity Vector Map", dispVec);
    cv::waitKey(0);


 ///////////////////////////////
 //MATLAB code for disparity
% Read in the left and right images
left_img = imread('left11.jpeg');
right_img = imread('right11.jpeg');

% Convert the images to grayscale
gray_img_l1 = rgb2gray(left_img);
gray_img_r1 = rgb2gray(right_img);

% Compute the size of the grayscale images
imsize_1 = size(gray_img_l1);

% Set the maximum disparity and block size parameters
max_disparity1 = 48;
block_size1 = 23;

% Initialize the disparity map
disparity_map1 = zeros(imsize_1, 'single');

% Compute the padding required for the block matching algorithm
padding1 = floor(block_size1/2);

% Loop over each pixel in the left image
for row_l1 = padding1+1 : imsize_1(1)-padding1
    for col_l1 = padding1+1 : imsize_1(2)-padding1

        % Extract the template from the left image
        template_1l = gray_img_l1(row_l1-padding1:row_l1+padding1, col_l1-padding1:col_l1+padding1);

        % Initialize the best SSD and disparity values
        best_ssd_1 = inf;
        best_disparity_1 = 0;

        % Loop over each possible disparity value
        for d = 0 : max_disparity1

            % Compute the corresponding column in the right image
            col_r1 = col_l1 - d;

            % Check that the right image template is within bounds
            if col_r1 >= padding1+1 && col_r1 <= imsize_1(2)-padding1

                % Extract the template from the right image
                template_r1 = gray_img_r1(row_l1-padding1:row_l1+padding1, col_r1-padding1:col_r1+padding1);

                % Compute the sum of squared differences (SSD) between the templates
                SSD1 = sum((template_1l(:)-template_r1(:)).^2);

                % Update the best SSD and disparity values if necessary
                if SSD1 < best_ssd_1
                    best_ssd_1 = SSD1;
                    best_disparity_1 = d;
                end
            end
        end

        % Store the best disparity value in the disparity map
        disparity_map1(row_l1,col_l1) = best_disparity_1;
    end
end

% Normalize the disparity map and convert it to an 8-bit unsigned integer
disparity_map1 = disparity_map1 ./ max(disparity_map1(:));
disparity_map1 = uint8(disparity_map1 * 255);

% Display the disparity map
imshow(disparity_map1);
}
```

Listing 1. Image Mosaicing