Image Mosaicing

Yash Mewada, Pratik Baldota
Project 2 Submission
Submitted in partial fulfillment of the requirements for the course of ***EECE5639***
March 17, 2023
Northeastern University

***Abstract*—In this report, we present a framework for image mosaicing that achieves good performance by estimating a homography between corresponding corner features. The framework involves applying a Harris corner detector to locate corners in two images, finding corresponding features, estimating a homography, and warping one image into the coordinate system of the other to create a mosaic. Our approach was evaluated on sample images, and the results demonstrate its effectiveness in producing high-quality mosaics. The report includes a flowchart, input/output images, potential corner feature location matches, and source code. Overall, our framework provides a robust and efficient solution for image mosaicing.**

***Index Terms*—Harris Corner Detector, NCC,RANSAC, Image Mosacing.**

I. Motivation

A. Background

The main idea behind the Image mosaicing patchwork of tiles or blocks, with each tile or block representing a different part of the original image. There are many different software tools and algorithms available to create mosaic effects. The choice of algorithm and parameters can have a significant impact on the final result and can be adjusted to achieve different artistic or technical goals.

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.
4) Estimating the homography and RANSAC.
5) Image overlapping

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. 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

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

The measure of corner response is given by where

$$detM = \lambda 1 * \lambda 2 \tag{2}$$

$$traceM = \lambda 1 + \lambda 2 \tag{3}$$

$$R = detM - K(traceM)^2 \tag{4}$$

(K - empirical constant, K= 0.04- 0.06) over the image, and then do non-maximum suppression to get a sparse set of corner features. As for Harris R function computing, we first use derivative operators. After corners are detected, it was normalized for threshold selection. A threshold of **127** for one image and **153** for another image was used.

C. 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

$$I(n) = \frac{I(n) - \mu}{(I)}, n = 1,..,25 \tag{5}$$

$$NCC = \frac{\sum_{i=1}^{25} x(i)y(i)}{\sqrt{(\sum_{i=1}^{25} x^2 \sum_{i=1}^{25} y^2)}} \tag{6}$$

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.

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

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

E. 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.

$$\begin{bmatrix} wx' \\ wy' \\ w \end{bmatrix} = \begin{bmatrix} h11 & h12 & h13 \\ h21 & h22 & h23 \\ h31 & h32 & h33 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{7}$$

F. Parameters used

- NCC threshold = 0.9
- Harris threshold for img1 = 127
- Harris threshold for img2 = 153
- In Harris Corner blocksize = 2, aperture = 5, K = 0.04
- Corner size fro img1= 55
- Corner size fro img2= 143
- inlier distance < 1
- number of inliers detected = 40
- NCC correspondence found with these parameters = 47
- H Matrix with **all the corresponding points**

$$\begin{bmatrix} 0.7758 & -0.0042 & 141.1203 \\ -0.0592 & 0.9108 & 11.2916 \\ -0.0004 & -1.6418 & 1 \end{bmatrix} \tag{8}$$

- H Matrix with **just inliers**

$$\begin{bmatrix} 0.7721 & -0.0020 & 141.2212 \\ -0.0611 & 0.9116 & 11.4512 \\ -0.0004 & -8.3478 & 1 \end{bmatrix} \tag{9}$$

It is evident from the above two matrices that after performing RANSAC on top of just inliers the points had only scaling transformation and the affine transformation between the points remained the same.

G. Wrapping Images

In Image processing, homography refers to the process of transforming an image using a homography matrix. A matrix defines a 7x7 matrix to define a projective transformation between two images. This can be done by estimating the homography matrix that maps the point in one image to another image. This involves applying the homography matrix to each pixel in the input image to determine its corresponding locations in the output images. Warping is the process of transforming an image to match the geometry of another image or to correct for distortions. This can involve stretching, compressing, rotating, or otherwise modifying the image to align with a reference image or to correct for perspective distortions.

H. Blending schemes

Blending is a technique used to combine one or two images to create composite images. The goal of blending is to create a smooth transition between the overlapping regions of the input images, without visible seams or artifacts. Black and white alpha blending schemes refer to the process of blending two or more grayscale images together to create a composite grayscale image. The blending process is similar to that of color images, but instead of combining color channels, the grayscale values of the input images are combined.

III. Values of parameters used

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 wrap perspective function to wrap the image in C++.
Also as the general case was $3 * \sigma$ for threshold, in our case it resulted in 5 times.

IV. Input Images

A. Input Images

In this project 2 sample input images of the DANA office were taken to apply Harris's corner detection, and apply RANSAC followed by wrapping images with a blending scheme. As the size of the images was perfect we did not

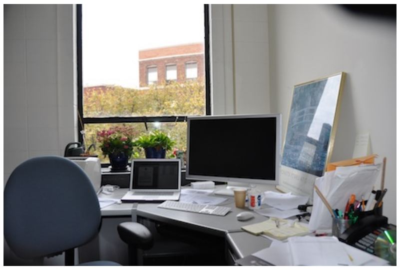

Fig. 1. Input Image 1

Fig. 2. Input Image 2

changed the scale of the images while reading them. All the processing was performed on grayscale images.

V. Output Images

A. Applying Harris corner

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

From the above output, fewer corners are detected than expected, due to the threshold used for the corner response values being too high, which causes the detector to ignore corners with fewer response values. In case lowering the threshold value may result in more corners detected.
We intentionally kept the threshold higher so that our output is not flooded with the detected corners.

B. Find correspondences between the images with RANSAC

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.
After NCC is calculated we only considered the points whose NCC value was greater than 0.9. 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.

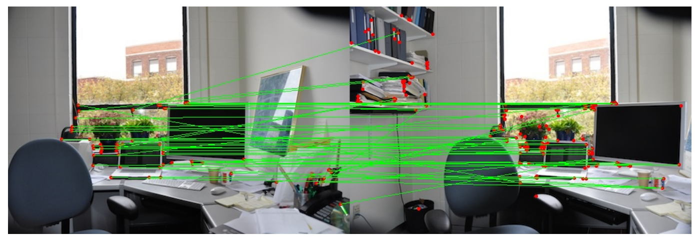

Fig. 4. Pre RANSAC output

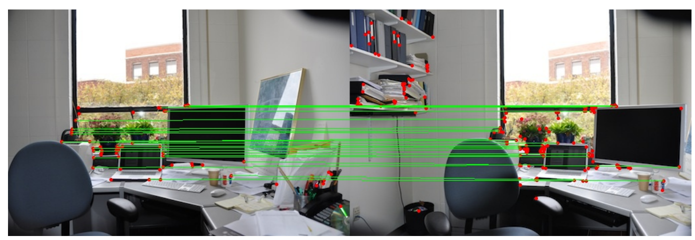

Fig. 5. Post RANSAC output

C. Wrapping the image

The below figure shows how the homography matrix turned out to be for just one image before wrapping it into another.
In this output, features were matched with corners and

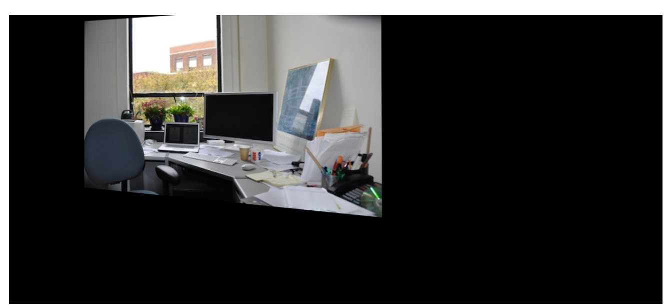

Fig. 6. Homography on one image

edges between two correspondence images. A further step was to blend the overlapping regions of the images to create a seamless transition between them.

Fig. 7. Stitched Image

D. Blending the image

Well, there are many algorithms for blending the images together. Some of them are Linear Blending, Laplacian Pyramid. For experimental purposes, we implemented Linear Blending. A linear blend operation sometimes referred to as Alpha Blending as below:

$$\mathbf{g}(\mathbf{x}) = (1-\alpha)f_0(x) + \alpha f_1(x) \tag{10}$$

By varying the value of α from $0 \rightarrow 1$ we get the temporal cross dissolve between two images. Below is the output for two different values of α. We tried implementing the average

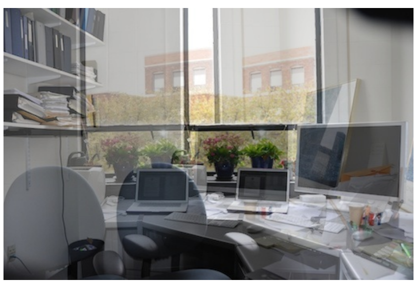

Fig. 8. Blending Image with $\alpha = 0.5$

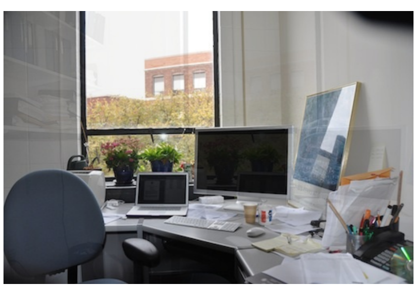

Fig. 9. Blending Image with $\alpha = 0.9$

blending technique where the pixels of the output image are replaced by the average pixel of both images, but it had no significant difference in the output and the image looked the same as of pre-averaging and post-averaging the pixels.

VI. Observations and conclusions

This report introduces a simple framework for image mosaicing 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, as well as advanced blending methods. Despite these challenges, the image-stitching algorithm remains a valuable tool in computer vision, robotics, and augmented reality.

We also found that the ***safe threshold*** for accurate corresponding points and RANSAC algorithm is between 160 and 170. Anything beyond this resulted in ***bad correspondences post RANSAC*** and anything below this resulted in ***Time complex algorithm***.

VII. Extra credit

Fig. 10. Extra credit Image

In Image processing software or a programming language that supports image processing libraries such as Python with OpenCV or MATLAB. The first step is to read in the two images in Matlab, one containing the image to be warped, and the other containing the frame (rectangle). Next, use a mouse click function like MATLAB's ginput to obtain four corresponding points on both images that represent the corners of the image to be warped and the four corners of the frame (rectangle) in the second image. These points will serve as inputs to the image-warping algorithm.

Once you have the corresponding points, you can use MATLAB's estgeotform2d function to compute a perspective transformation matrix. This matrix is then used to warp the image to fit within the frame (rectangle) in the second image. Finally, you can save the resulting image as a new file, or display it on the screen using an appropriate function.

VIII. FLOWCHART

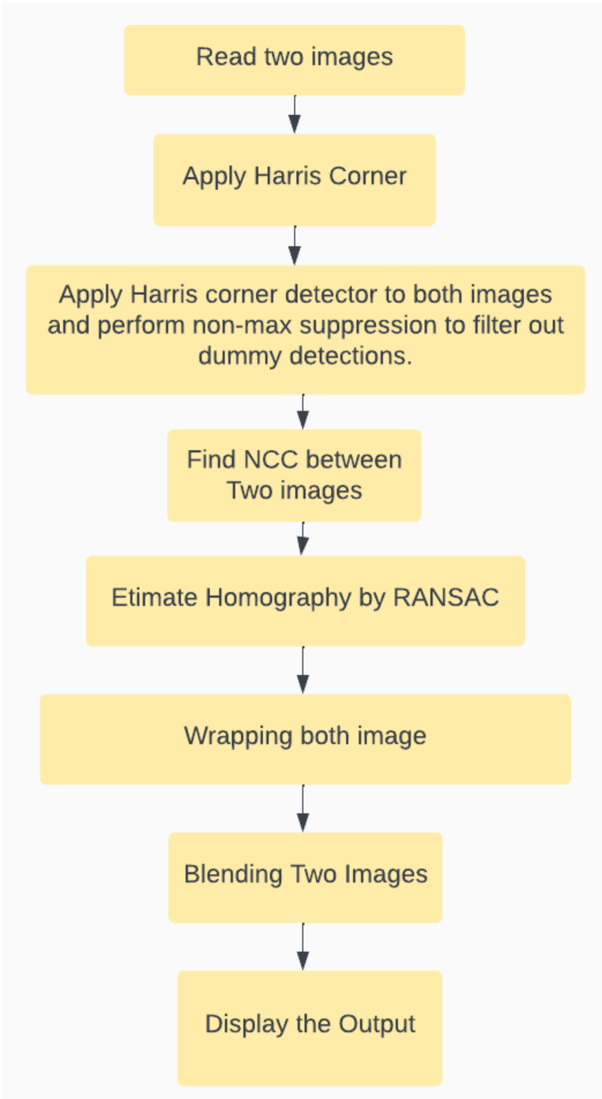

Fig. 11. Flowchart

IX. CODE

The code for our project can be found here : [GitHub]

X. Appendix

```cpp

#include <cstdio>
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

/* Class for Image Mosaicing function declaration*/
class Image_Mosaicing
{
private:
    /* data */
public:
    string path_to_images;
    Mat img1;
    Mat img2;
    Mat img1_gray;
    Mat img2_gray;
    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 = 1000;
    double ncc_thres = 0.5;
    Image_Mosaicing(string _path);

    pair<vector<Point>, vector<Point>> perform_harris(int thresh);

    double calc_NCC(Mat temp1, Mat temp2);

    vector<pair<Point, Point>> get_correspondences(vector<Point> c1,vector<Point> c2);

    void visualise_corress(vector<pair<Point, Point>> corresspondences);

    void compute_homography(Mat matched_corners1, Mat mathched_corners2);

    vector<Point> get_random_points(vector<Point> points, int n);

    Mat compute_homography(vector<Point> src_points, vector<Point> dst_points);

    vector<int> get_inliers(vector<Point> src_points, vector<Point> dst_points, Mat homography);

    Mat estimate_homography_ransac(vector<Point> src_points, vector<Point> dst_points);
    vector<Point> harris_detector_for_img1();
    vector<Point> harris_detector_for_img2();
    vector<Point2f> cvt_pts_pt2f(vector<Point> points);

};
Image_Mosaicing::Image_Mosaicing(string _path)
{
    cout << "This is a demo for Image Mosaicing code" << endl;
    this->path_to_images = _path;
    img1 = imread(_path + string("DSC_0311.JPG"));
    img2 = imread(_path + string("DSC_0312.JPG"));
    // resize(img1, img1, Size(), 0.75, 0.75);
    // resize(img2, img2, Size(), 0.75, 0.75);
    cvtColor(img1, img1_gray, COLOR_BGR2GRAY);
    cvtColor(img2, img2_gray, COLOR_BGR2GRAY);
}


/* Apply sobel mask to the images and Compute the harris R function along with the detected corners*/


pair<vector<Point>, vector<Point>> Image_Mosaicing::perform_harris(int thresh){
    Mat dst, dst_norm, dst_norm_scaled;
    Mat dst2, dst_norm2, dst_norm_scaled2;
    vector<Point> cor_1,cor_2;
    dst = Mat::zeros(img1_gray.size(), CV_32FC1);
    dst2 = Mat::zeros(img2_gray.size(), CV_32FC1);

    int blockSize = 2;
    int apertureSize = 5;
    double k = 0.04;

    cornerHarris(img1_gray, dst, blockSize, apertureSize, k, BORDER_DEFAULT);
    normalize(dst, dst_norm, 0, 255, NORM_MINMAX, CV_32FC1, Mat());
    convertScaleAbs( dst_norm, dst_norm_scaled );

    cornerHarris(img2_gray, dst2, blockSize, apertureSize, k, BORDER_DEFAULT);
    normalize(dst2, dst_norm2, 0, 255, NORM_MINMAX, CV_32FC1, Mat());
    convertScaleAbs( dst_norm2, dst_norm_scaled2 );

    vector<Point> corner_coor;

    for( int i = 0; i < dst_norm.rows ; i++ )
    {
        for( int j = 0; j < dst_norm.cols; j++ )
        {
            if( (int) dst_norm.at<float>(i,j) > thresh - 33
            // && dst_norm.at<float>(i, j) > dst_norm.at<float>(i - 1, j - 1)
            // && dst_norm.at<float>(i, j) > dst_norm.at<float>(i - 1, j)
            // && dst_norm.at<float>(i, j) > dst_norm.at<float>(i - 1, j + 1)
            // && dst_norm.at<float>(i, j) > dst_norm.at<float>(i, j - 1)
            // && dst_norm.at<float>(i, j) > dst_norm.at<float>(i, j + 1)
            // && dst_norm.at<float>(i, j) > dst_norm.at<float>(i + 1, j - 1)
            // && dst_norm.at<float>(i, j) > dst_norm.at<float>(i + 1, j)
            // && dst_norm.at<float>(i, j) > dst_norm.at<float>(i + 1, j + 1)
            )
            {
                // circle( img1, Point(j,i), 1, Scalar(0,0,255), 2, 8, 0 );
                cor_1.push_back(Point(j,i));
            }
        }
    }
    for( int i = 0; i < dst_norm2.rows ; i++ )
    {
        for( int j = 0; j < dst_norm2.cols; j++ )
        {
            if( (int) dst_norm2.at<float>(i,j) > thresh - 7
            // && dst_norm2.at<float>(i, j) > dst_norm2.at<float>(i - 1, j - 1)
            // && dst_norm2.at<float>(i, j) > dst_norm2.at<float>(i - 1, j)
```

```
            // && dst_norm2.at<float>(i, j) > dst_norm2.at<float>(i - 1, j + 1)
            // && dst_norm2.at<float>(i, j) > dst_norm2.at<float>(i, j - 1)
            // && dst_norm2.at<float>(i, j) > dst_norm2.at<float>(i, j + 1)
            // && dst_norm2.at<float>(i, j) > dst_norm2.at<float>(i + 1, j - 1)
            // && dst_norm2.at<float>(i, j) > dst_norm2.at<float>(i + 1, j)
            // && dst_norm2.at<float>(i, j) > dst_norm2.at<float>(i + 1, j + 1)
            )
            {
                // circle( img2, Point(j,i), 1, Scalar(0,0,255), 2, 8, 0 );
                cor_2.push_back(Point(j,i));
                // cout << Point(j,i) << endl;
            }
        }
    }
    Mat concated_img;
    cout << "corner1_size" << "  ";
    cout << cor_1.size() << "   ";
    cout << "corner2_size" << "  ";
    cout << cor_2.size() << endl;
    if (img1.cols != img2.cols) {
        double scale = (double)img1.cols / img2.cols;
        resize(img2, img2, Size(img1.cols, scale*img2.rows));
    }

    // Concatenate images vertically
    Mat result;
    cv::vconcat(img1, img2, concated_img);
    cv::namedWindow( "corners_window" );
    cv::imshow( "corners_window", concated_img);
    cv::waitKey(0);
    return make_pair(cor_1,cor_2);
}

double Image_Mosaicing::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>> Image_Mosaicing::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++) {
        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 (d != Point(0,0)){
                // pair<Point,Point> c;
                // c.first = c1[i];
                // c.second = d;
                // cout << temp_ncc << endl;
                // corres.push_back(c);
                // }
```

```
            }
        }
        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;
    return corres;
}

void Image_Mosaicing::visualise_corress(vector<pair<Poin Point>> fc){
    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);
    hconcat(img1, img2, img_matches);
    for (int i = 0; i < fc.size() ; i++) {
        Point pt1 =  fc[i].first;
        Point pt2 = Point(fc[i].second.x + img1.cols, fc[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);
        line(img_matches, pt1, pt2, Scalar(0, 255, 0), 1);
    }
    imshow( "result_window", img_matches );
    cv::imwrite("Correpondences w/o Homography.jpg",img_matches);
    cv::waitKey(0);
}

vector<Point> Image_Mosaicing::get_random_points(vector<Point> points, int n){
    // random_shuffle(points.begin(), points.end());
    vector<Point> random_points;
    // cout << points << endl;
    for (int i = 0; i < n; i++) {
        int random_num = rand() % points.size();
        random_points.push_back(points[random_num]);
    }
    return random_points;
}

Mat Image_Mosaicing::compute_homography(vector<Point> src_points, vector<Point> dst_points) {
    Mat homography = findHomography(src_points, dst_points, RANSAC, THRESHOLD);
    // cout << "found homography" << endl;
    // create a matrix of 8x9
    return homography;
}

vector<int> Image_Mosaicing::get_inliers(vector<Point> src_points, vector<Point> dst_points, Mat homography) {
    vector<int> inliers;
    vector<pair<Point, Point>> temp_corres;
    for (int i = 0; i < src_points.size(); i++) {
        Point src_point = src_points[i];
        Point dst_point = dst_points[i];
        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 = homography * src;
        pred_dst = pred_dst/pred_dst.at<double>(2, 0);
        double distance = norm(pred_dst - dst);
        if (distance < 1) {
            pair<Point,Point> c;
            c.first = src_points[i];
            c.second = dst_points[i];
            temp_corres.push_back(c);
            // cout << "got inliers" << endl;
            inliers.push_back(i);
            // cout << i  << endl;
        }
    }
    return inliers;
}

vector<Point2f> Image_Mosaicing::cvt_pts_pt2f(vector<Point> points){
    vector<Point2f> new_pts;
    for (int i = 0; i < points.size(); i++){
        new_pts.push_back(Point2f(points[i].x,points[i].y));
    }
    return new_pts;
}

Mat Image_Mosaicing::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);
    // int n = 0;
    // cout << src_points.size() << endl;
    // cout << dst_points.size() << endl;

    srand(time(NULL));
    for (int i = 0; i < MAX_ITERATIONS; i++) {
        // cout << i << endl;
        vector<Point> random_src_points = get_random_points(src_points, MIN_POINTS);
        vector<Point> random_dst_points = get_random_points(dst_points, MIN_POINTS);
        Mat homography = findHomography(cvt_pts_pt2f(random_src_points),cvt_pts_pt2f noArray(), 1000, 0.995);
        vector<Point> inliers1_tmp, inliers2_tmp;
        // cout << homography << endl;
        if (homography.empty()) {
            continue;
        }
        // vector<int> inliers = get_inliers(src_points, dst_points, homography);
        vector<int> inliers;
        int num_inliers = 0;
        vector<pair<Point, Point>> temp_corres;
        int inlier_idx = -1;
        vector<Point2f> curr_inliers;
```

```cpp
        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 = 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 = 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;
    // }
    // best_homography = findHomography(cvt_pts_pt2f(inlier_src_points),cvt_
    visualise_coress(bestCorrespondingPoints);
    return best_homography;
}
int main(){
    string path = "/home/yash/Documents/Computer_VIsion/CV_2Project/D
    Mat img1 = imread(path + string("DSC_0311.JPG"));
    Mat img2 = imread(path + string("DSC_0312.JPG"));
    Image_Mosaicing p3(path);
    vector<Point> cor_img1,cor_img2;
    // vector<Point> cor_img1,cor_img2;
    // cor_img1 = p3.harris_detector_for_img1();
    // cor_img2 = p3.harris_detector_for_img2();
    tie(cor_img1,cor_img2) = p3.perform_harris(160);
    for (auto e : cor_img2){
        cout << e << endl;
    }
    vector<pair<Point,Point>> corres;
    corres = p3.get_correspondences(cor_img1,cor_img2);
    cout << "done" << endl;
    // // cout << corres << endl;
    p3.visualise_coress(corres);
    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);
    }
    // std::vector<Point2f> points1, points2; // your corresponding points
    Mat best_h;
    best_h = p3.estimate_homography_ransac(src,dst);
    cout << best_h << endl;
    vector<Point> c1,c2;
    c1.push_back(Point(0,0));
    c1.push_back(Point(img1.cols,0));
    c1.push_back(Point(img1.cols,img1.rows));
    c1.push_back(Point(0,img1.rows));
    for (int i=0;i<4;i++){
        Point src_point = c1[i];
        Mat src = (Mat_<double>(3, 1) << src_point.x, src_point.y, 1);
        Mat pred_dst = best_h * src;
        pred_dst /= pred_dst.at<double>(2, 0);

    c2.push_back(Point(pred_dst.at<double>(0,0),pred_dst.at<dou
    }
    cout << "here" << endl;
    int min_x = min(min(c2[0].x,c2[1].x),min(c2[2].x,c2[3].x));
    int max_x = max(max(c2[0].x,c2[1].x),max(c2[2].x,c2[3].x));

    int min_y = min(min(c2[0].y,c2[1].y),min(c2[2].y,c2[3].y));
    int max_y = max(max(c2[0].y,c2[1].y),max(c2[2].y,c2[3].y));

    int height = max_y - min_y;
    int width = max_x - min_x;
    cout << max_x << endl;
    cout << max_y << endl;
    Mat output(1000,1000,img1.type());
    // warpPerspective(img1,output,best_h.inv(),output.size());
    // img1.copyTo(output(Rect(c2[0].x - min_x, c2[0].y - min_y, img1.cols, img1.rows)));
    // cout << "her3" << endl;
    Mat H_inv;
    invert(best_h, H_inv);
    // cout << "her4" << endl;
    warpPerspective(img2, output, best_h, output.size());
    // // cout << src<< "    ";
    // cout << src.size() << endl;
    // cout << dst.size() << endl;
    // Mat result;
    // warpPerspective(img1, result, best_h, Size(img1.cols + img2.cols, img1.rows));
    // // warpPerspective(img2, result, best_h, Size(img1.cols + img2.cols, img1.rows));
    // Mat roi(result, Rect(0, 0, img2.cols, img2.rows));
    // img2.copyTo(roi);
    // namedWindow("Stitched image", WINDOW_NORMAL);
    imshow("Stitched image", output);
```

```
    waitKey(0);
    destroyAllWindows();
    // cv::imwrite("stiched image.jpg", img1);

}



blending code
#include <opencv2/opencv.hpp>
using namespace cv;

int main() {
    // Define paths to input images
    std::string path1 = "/home/pratik/Desktop/pratik1/CV_2Project
    /DanaOffice/DSC_0310.JPG";
    std::string path2 = "/home/pratik/Desktop/pratik1/CV_2Project
    /DanaOffice/DSC_0311.JPG";

    // Load input images
    Mat img1 = imread(path1, IMREAD_COLOR);
    Mat img2 = imread(path2, IMREAD_COLOR);

    // Resize the images to the same size
    resize(img1, img1, img2.size());

    // Convert images to grayscale
    Mat gray1, gray2;
    cvtColor(img1, gray1, COLOR_BGR2GRAY);
    cvtColor(img2, gray2, COLOR_BGR2GRAY);

    // Create a mask to blend the images
    Mat mask = Mat::zeros(img2.size(), CV_8UC1);
    rectangle(mask, Rect(100, 100, 200, 200), Scalar(255), -1);

    // Blend the images using a weighted sum
    Mat blended;
    addWeighted(gray1, 0.5, gray2, 0.5, 0, blended);

    // Apply the mask to the blended image
    Mat masked;
    blended.copyTo(masked, mask);

    // Display the results
    imshow("Image 1", img1);
    imshow("Image 2", img2);
    imshow("Blended", masked);
    //imwrite(output_path, masked);
    cv::imwrite("Blended.jpg", masked);
    waitKey(0);

    return 0;
}


% stiching code
#include <iostream>
#include <opencv2/opencv.hpp>

using namespace std;
using namespace cv;

int main() {

    // Define paths to input images
    string path1 = "/home/pratik/Desktop/pratik1/CV_2Project
    /DanaOffice/DSC_0310.JPG";
    string path2 = "/home/pratik/Desktop/pratik1/CV_2Project
    /DanaOffice/DSC_0311.JPG";

    Mat img1 = imread(path1, IMREAD_COLOR);
    Mat img2 = imread(path2, IMREAD_COLOR);

    if (img1.empty() || img2.empty()) {
        cout << "Could not open or find the image(s)." << endl;
        return -1;
    }

    // Detect features in both images
    Ptr<FeatureDetector> detector = ORB::create();
    vector<KeyPoint> keypoints1, keypoints2;
    detector->detect(img1, keypoints1);
    detector->detect(img2, keypoints2);

    // Match features between the images
    Ptr<DescriptorExtractor> extractor = ORB::create();
    Mat descriptors1, descriptors2;
    extractor->compute(img1, keypoints1, descriptors1);
    extractor->compute(img2, keypoints2, descriptors2);

    Ptr<DescriptorMatcher> matcher = DescriptorMatcher::create("BruteForce-Hamming");
    vector<DMatch> matches;
    matcher->match(descriptors1, descriptors2, matches);

    // Filter matches using RANSAC algorithm
    vector<Point2f> points1, points2;
    for (size_t i = 0; i < matches.size(); i++) {
        points1.push_back(keypoints1[matches[i].
        queryIdx].pt);
        points2.push_back(keypoints2[matches[i].
        trainIdx].pt);
    }
    Mat mask;
    Mat homography = findHomography(points1, points2, RANSAC, 5.0, mask);
    // Mat homography = (Mat_<int>(3,3) << 1,0,0,0,1,0,0,0,1);

    // Warp image1 to align with image2
    Mat result;
    warpPerspective(img1, result, homography, Size(img1.cols + img2.cols, img1.rows));
    Mat roi(result, Rect(0, 0, img2.cols, img2.rows));
    img2.copyTo(roi);
    // Compute homography between images
    // Compute inverse of homography matrix
    Mat inverse_homography = homography.inv();
    // Print inverse homography matrix
    cout << "Inverse homography matrix: " << endl;
    for (int i = 0; i < inverse_homography.rows; i++) {
    for (int j = 0; j < inverse_homography.cols; j++) {
        cout << inverse_homography.at<double>(i,j) << " ";
    }
    cout << endl;
    }
    // Print homography matrix
    cout << "Homography matrix: " << endl;
    for (int i = 0; i < homography.rows; i++) {
    for (int j = 0; j < homography.cols; j++) {
        cout << homography.at<double>(i,j) << " ";
    }
    cout << endl;
    }
```

```
//    // Determine size of output image
    vector<Point2f> corners1(4), corners2(4);
    corners1[0] = Point2f(0, 0);
    corners1[1] = Point2f(img1.cols, 0);
    corners1[2] = Point2f(img1.cols, img1.rows);
   corners1[3] = Point2f(0, img1.rows);
    namedWindow("Stitched image", WINDOW_NORMAL);
    imshow("Stitched image", result);
    waitKey(0);
    destroyAllWindows();
    cv::imwrite("Stitched image.jpg", result);
    return 0;
}




 % Cross line code

 #include <iostream>
#include <vector>
#include <opencv2/opencv.hpp>
#include "opencv2/features2d.hpp"

using namespace std;
using namespace cv;

int main(int argc, char** argv)
{
// Read in the two images
Mat image1 = imread("/home/pratik/Desktop/pratik1/CV_2Project
/DanaOffice/DSC_0310.JPG");
Mat image2 = imread("/home/pratik/Desktop/pratik1/CV_2Project
/DanaOffice/DSC_0311.JPG");
// Convert images to grayscale
Mat gray1, gray2;
cvtColor(image1, gray1, COLOR_BGR2GRAY);
cvtColor(image2, gray2, COLOR_BGR2GRAY);
// Detect ORB keypoints and compute descriptors
Ptr<ORB> detector = ORB::create();
vector<KeyPoint> keypoints1, keypoints2;
Mat descriptors1, descriptors2;
detector->detectAndCompute(gray1, Mat(), keypoints1, descriptors1);
detector->detectAndCompute(gray2, Mat(), keypoints2, descriptors2);
// Match keypoints
BFMatcher matcher(NORM_HAMMING);
vector<DMatch> matches;
matcher.match(descriptors1, descriptors2, matches);
// Filter matches based on distance
double max_dist = 0;
double min_dist = 100;
for(int i = 0; i < descriptors1.rows; i++)
{
double dist = matches[i].distance;
if(dist < min_dist) min_dist = dist;
if(dist > max_dist) max_dist = dist;
}
vector<DMatch> good_matches;
for(int i = 0; i < descriptors1.rows; i++)
{
if(matches[i].distance < 3*min_dist)
{
good_matches.push_back(matches[i]);
}
}
// Draw matches
Mat img_matches;
drawMatches(image1, keypoints1, image2, keypoints2,
good_matches, img_matches, Scalar::all(-1),
Scalar::all(-1), vector<char>(),
DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS);
// Display matches
namedWindow("Matches", WINDOW_NORMAL);
imshow("Matches", img_matches);
cv::imwrite("Matches.jpg", img_matches);
waitKey(0);
return 0;
}


//extra credit code
string_img = imread("10.jpeg");
Tomdanahall_img1 = imread("1.JPG");
sizehall_src = size(string_img)
sizesource_dest = size(Tomdanahall_img1)
imshow(Tomdanahall_img1)
[x, y] = ginput(4);
coords = [x, y];
src_coords = [[0;0],[sizehall_src(2);0],[sizehall_src(2);sizehall_src(1)
src_coords'
best_inliers = [1,2,3,4];
h_f = estgeotform2d(src_coords',coords,"projective")
%h = estimateBestHomography(src_coords',coords,best_inliers)
H1_ = inv(h_f.A);

xlim = sizesource_dest(1);
ylim = sizesource_dest(2);
[xi yi] = meshgrid(1:ylim,1:xlim);
xx = (H1_(1,1)*xi + H1_(1,2) * yi + H1_(1,3)) ./ (H1_(3,1)*xi + H1_(3,2)*yi + H1_(3,3));
yy = (H1_(2,1)*xi + H1_(2,2) * yi + H1_(2,3)) ./ (H1_(3,1)*xi + H1_(3,2)*yi + H1_(3,3));

foo_R1 = uint8(interp2(double(string_img(:,:,1)),xx,yy));
foo_Ground = uint8(interp2(double(string_img(:,:,2)),xx,yy));
foo_Base = uint8(interp2(double(string_img(:,:,3)),xx,yy));
bw = ~poly2mask(coords(:,1)',coords(:,2)',sizesource_dest(1),siz

Tomdanahall_img1 = uint8(Tomdanahall_img1) .* uint8(bw);
final1_img = cat(3,foo_R1,foo_Ground,foo_Base);
Tomdanahall_img1 = Tomdanahall_img1 + final1_img;
imshow(Tomdanahall_img1,[0,255]);
figure;
```

Listing 1. Image Mosaicing