Image Mosaicing

Anush Sriram Ramesh
MS Robotics, College of Engineering
Northeastern University
Boston, USA
ramesh.anu@northeastern.edu

Anuj Shrivatsav Srikanth
MS Robotics, College of Engineering
Northeastern University
Boston, USA
srikanth.anu@northeastern.edu

***Abstract*—The technology of panorama has been around for over a decade on our mobile devices now and this project aims at breaking down the fundamentals in operation behind the feature. Several consecutive images, taken with the camera moved slightly from the location of the previous photo, have many commonalities between them. We use this implication to estimate the spatial relation between two images and stitch them together into one embodiment of a very large field of view. We use Harris Corner Detector to identify features in one picture. These feature locations are matched to their corresponding image in the 2nd image using the Normalized Cross Correlation measure. These correspondences are very noisy with many outliers, which are begging to be cleared up with the good old RANSAC algorithm using which the best set of correspondences and best homography corresponding to that is estimated. This is finally used to warp and blend both images together.**

***Index Terms*—Computer Vision, Image Mosaicing, Harris corner detection, RANSAC, Homography**

I. Description of Algorithms

A. Harris Corner Detection

The Harris Corner Detector is a popular algorithm used in computer vision to detect corners in images. It was proposed by Chris Harris and Mike Stephens in 1988 and has since become a standard technique in the field.

The Harris Corner Detector algorithm works by analyzing changes in the intensity of an image in small windows or patches. Specifically, it computes a corner response function for each pixel in the image, which measures the likelihood that the pixel is part of a corner. The corner response function is defined as:

$$R = \det(M) - k(\text{trace}(M))^2 \tag{1}$$

where M is the covariance matrix of the image patch centered at the pixel, and k is a constant that controls the relative importance of the determinant and trace terms. The covariance matrix is computed by convolving the image with a Gaussian kernel. The Gaussian kernel is used to smooth the image and reduce the effect of noise. The value of k is usually set to 0.04.

The structure tensor M is defined as follows:

$$M = \begin{bmatrix} \sum_{x,y} w(x,y)(I_x)^2 & \sum_{x,y} w(x,y)(I_x)(I_y) \\ \sum_{x,y} w(x,y)(I_x)(I_y) & \sum_{x,y} w(x,y)(I_y)^2 \end{bmatrix}$$

where $I_x and I_y$ are the partial derivatives of the image with respect to x and y, respectively. The Gaussian kernel w is defined as follows:

$$w(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2}$$

where σ is the standard deviation of the Gaussian kernel.

The Harris Corner Detector algorithm computes the corner response function for every pixel in the image and then applies a threshold to select the pixels with the highest response. These high-response pixels are considered to be corners. The threshold is usually set to a value that is a small multiple of the maximum response in the image.

B. Normalized Cross Correlation Scores

We use the normalized cross-correlation score to find the best matches between the features in the two images. The normalized cross-correlation score is defined as:

$$NCC = \frac{\sum_{x,y} w(x,y)(I_1(x,y)-\bar{I}_1)(I_2(x,y)-\bar{I}_2)}{\sqrt{\sum_{x,y} w(x,y)(I_1(x,y)-\bar{I}_1)^2}\sqrt{\sum_{x,y} w(x,y)(I_2(x,y)-\bar{I}_2)^2}} \tag{3}$$

where I_1 and I_2 are the two images, and $\bar{I}_1$ and $\bar{I}_2$ are the mean intensities of I_1 and I_2, respectively.

We create a template centered around each corner feature in image 1. We compare this with similar-sized patches in image 2. We compute the normalized cross-correlation score for each patch in image 2 and select the patch with the highest score as the best match for the template. We repeat this process for all the corner features in image 1.

The NCC algorithm normalizes the correlation coefficient by dividing the covariance of the two signals by the product of their standard deviations. This normalization ensures that the correlation coefficient is between -1 and 1, where a value of 1 indicates a perfect match between the two signals. The NCC algorithm is invariant to the translation, rotation, and scaling of the two signals. This makes it a good choice for matching features in images.

C. RANSAC and Estimating Best Homography

RANSAC is an iterative algorithm for the robust estimation of parameters from a subset of inliers from the complete data set. It is used to estimate the best homography between two images. The algorithm is as follows:

Algorithm 1 RANSAC Algorithm

procedure RANSAC(Correspondences, &BestCorrespondences)
Require: Correspondences
Ensure: Best Homography
Initialize:
Set the number of iterations
Set the threshold
Set the number of inliers
Set the best homography
Set the best inliers
Set the best inlier count
For each iteration:
Randomly select 4 correspondences
Compute the homography using the 4 correspondences
For each correspondence:
Compute the distance between the correspondence and the homography
If the distance is less than the threshold:
Increment the inlier count
Add the correspondence to the inliers
If the inlier count is greater than the best inlier count:
Set the best inlier count to the inlier count
Set the best inliers to the inliers
Set the best homography to the homography
Return the best homography

D. Warping and Blending

Warping is the process of transforming an image to a new coordinate system. We use the best homography to warp image 1 to the new image's coordinate system. We then blend the warped image with the original image 2 to create the final image. We use the following equation to blend the two images:

$$I(x, y) = \alpha I_1(x, y) + (1 - \alpha) I_2(x, y) \tag{4}$$

where I_1 and I_2 are the two images, and α is the blending factor. We set α to 0.5 for the final image. This is the same as taking the average of the two images at pixel locations where the two images overlap.

The size of the new image is determined by identifying the minimum and maximum of x and y in the transformed image coordinate frame. Thus a new image with the size of a union of pixels from both images warped is created.

II. Experiments

With the provided images of *DanaHallway* and *DanaOffice*, we tested our program by sampling two images out of the given datasets and applied the concept of planar warping to obtained a mosaic with the two images. We take in two images apply Harris corner detector on them and then find the correspondences between them using Normalized Cross-Correlation. To remove the outliers we perform RANSAC and get the best correspondences which is then used for warping the images to obtain the Mosaic. Outputs of all the above mentioned methods are shown below:

A. Dana Office Dataset

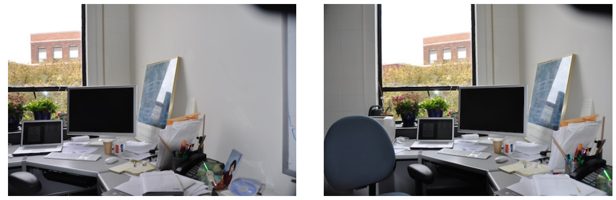

Fig. 1. input images

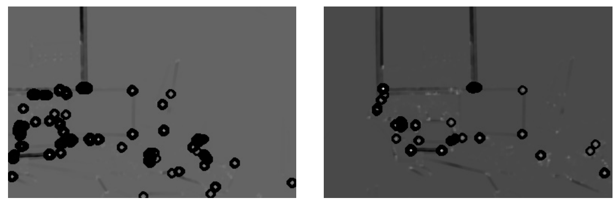

Fig. 2. Detected Harris corners on the input images

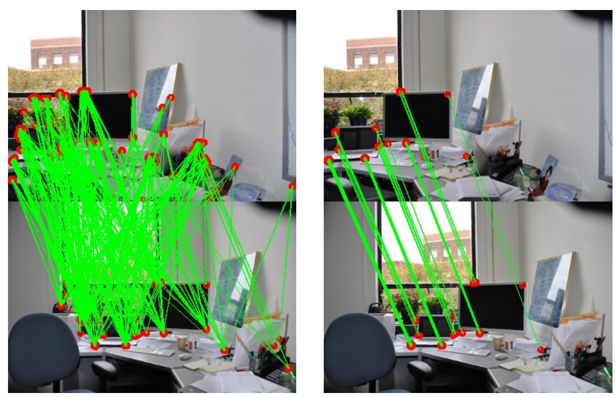

Fig. 3. Feature Correspondences before and after RANSAC

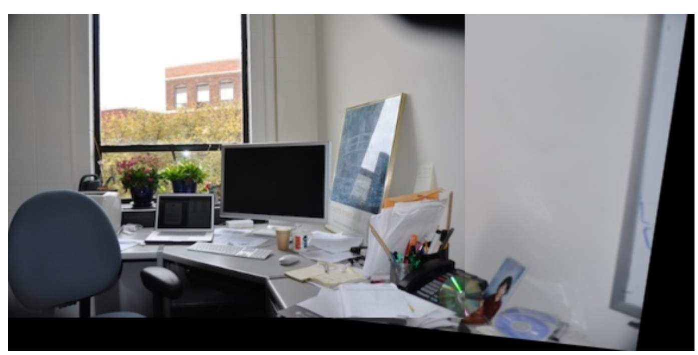

Fig. 4. Mosaic formed by warping the two images without blending

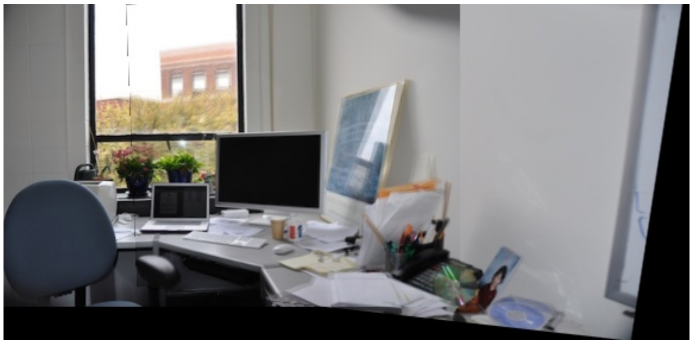

Fig. 5. Mosaic formed by warping the two images with blending

B. Dana Hallway Dataset

Fig. 6. input images

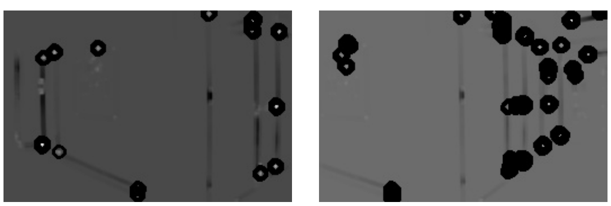

Fig. 7. Detected Harris corners on the input images

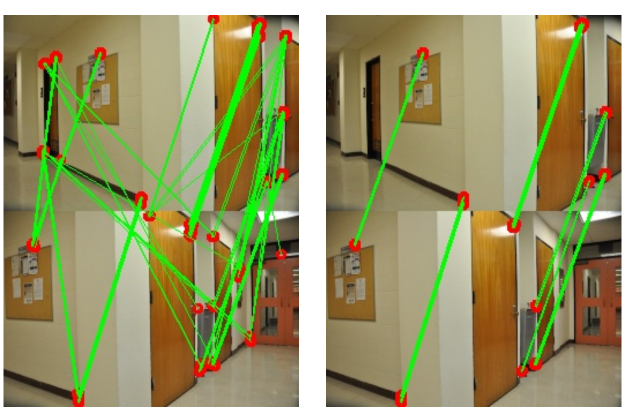

Fig. 8. Feature Correspondences after applying Harris Corner with NCC

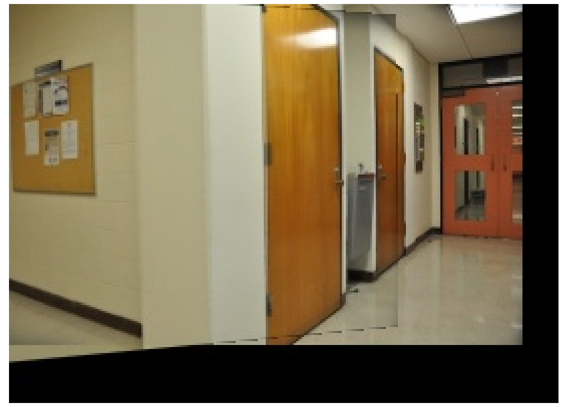

Fig. 9. Mosaic formed by warping the two images without blending

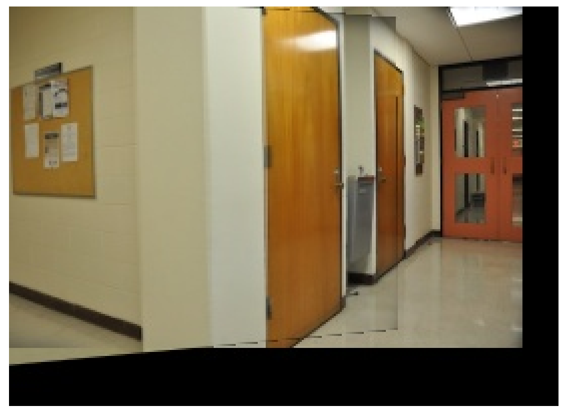

Fig. 10. Mosaic formed by warping the two images with blending

C. Extra Credit

Using the *FindHomography* method used in the RANSAC procedure. We find the homography that maps the corners of the sample image to a specified ROI in the input image. Using this homography and *WarpPerspective* method the sample image is written on to the input image with the rest of the pixels in the input image retained as they are.

Fig. 11. Sample image warped on to the original image

III. Values of Parameters

[h] The values of the parameters used in the experiments for each section of the processing are as follows.

A. Read Images

Parameter	Value
Resize Factor	0.5

TABLE I
Values of Parameters

B. Harris Corner Detector

Parameter	Value
Kernel Size	5
Threshold	0.04

TABLE II
Parameters used for Harris Corner Detector

C. Normalized Cross Correlation Scores

Parameter	Value
Kernel Size	15
Lower Bound Threshold for NCC	0.8

TABLE III
Parameters used for Normalized Cross Correlation Scores

D. RANSAC and Estimating Best Homography

Parameter	Value
Number of Iterations	1000
Threshold	1
Number of Correspondences	4

TABLE IV
Parameters used for RANSAC and Estimating Best Homography

E. Warping and Blending

Parameter	Value
Blending Factor	0.5

TABLE V
Parameters used for Warping and Blending

IV. Observation and Conclusion

The observation from the experiments is that the Harris Corner Detector is a good choice for finding corner features in images. The RANSAC is very efficient in finding the best homography between two images.

More the number of corners retained in the corner detection step, more the number of correspondences found in the next step. This leads to a better homography estimation in the RANSAC step. Although the computation time is high. The NCC algorithm is a $O(N^2)$ algorithm, and the computation time increases very quickly with the template size. We found that the best results were obtained at a sweet spot of 15x15 kernel size for template matching.

V. Appendix

A detailed overview of the project's file structure, along with the code is available in the GitHub Repository.

Below are snippets from the code for easy evaluation. We kindly request you browse through the GitHub repo before a complete evaluation of the code.

A. Normalized Cross Correlation

```
double NCC(Mat t1, Mat t2)
{   //cout << ">>NCC" << endl;
    // calculate the mean of the template
       1
    double mean1 = 0;
    for(int i=0; i<t1.rows; i++)
    {
        for(int j=0; j<t1.cols; j++)
        {
            mean1 += t1.at<uchar>(i,j);
        }
    }
    mean1 = mean1/(t1.rows*t1.cols);
    // calculate the mean of the template
       2
    double mean2 = 0;
    for(int i=0; i<t2.rows; i++)
    {
        for(int j=0; j<t2.cols; j++)
        {
            mean2 += t2.at<uchar>(i,j);
        }
    }
    mean2 = mean2/(t2.rows*t2.cols);
    // calculate the standard deviation
       of the template 1
    double std1 = 0;
    for(int i=0; i<t1.rows; i++)
    {
        for(int j=0; j<t1.cols; j++)
        {
            std1 += pow(t1.at<uchar>(i,j)
                - mean1, 2);
        }
    }
    std1 = sqrt(std1/(t1.rows*t1.cols));
    // calculate the standard deviation
       of the template 2
    double std2 = 0;
    for(int i=0; i<t2.rows; i++)
    {
        for(int j=0; j<t2.cols; j++)
        {
            std2 += pow(t2.at<uchar>(i,j)
                - mean2, 2);
        }
    }
```

```
    std2 = sqrt(std2/(t2.rows*t2.cols));
    // calculate the normalized cross-
        correlation
    double ncc = 0;
    for(int i=0; i<t1.rows; i++)
    {
        for(int j=0; j<t1.cols; j++)
        {
            ncc += (t1.at<uchar>(i,j) -
                mean1)*(t2.at<uchar>(i,j)
                - mean2);
        }
    }
    ncc = ncc/(t1.rows*t1.cols*std1*std2)
        ;
    return ncc;
}
```

B. Read Images

```
vector<Mat> read_images(vector<string>
    directory)
{
    cout<<"read_images"<<endl;
    // create a vector to store the
        images
    vector<Mat> imgs;
    // read the images from the directory
    for(int i = 0; i < directory.size();
        i++) {
        Mat img = imread(directory[i]);
        resize(img, img, Size(), 0.75,
            0.75);
        imgs.push_back(img);
    }
    // print the size of imput images
    return imgs;
}
```

C. Find Corner Features

```
tuple<vector<Point>, Mat> find_corners(
    Mat img)
{
    cout<<"find_corners"<<endl;
    // convert the image to grayscale
    Mat gray;
    cvtColor(img, gray, COLOR_BGR2GRAY);
    // apply the harris corner detector
    Mat dst, dst_norm, dst_norm_scaled;
    dst = Mat::zeros(gray.size(),
        CV_32FC1);
    // compute the harris R matrix over
        the image
    cornerHarris(gray, dst, 5, 5, 0.04,
        BORDER_DEFAULT);
    // normalize the R matrix
    normalize(dst, dst_norm, 0, 255,
        NORM_MINMAX, CV_32FC1, Mat());
    convertScaleAbs(dst_norm,
        dst_norm_scaled);
    // threshold the R matrix to find the
         corners
    // push corner points into a vector
    vector<Point> corners;
    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) > 120) {
                circle(dst_norm_scaled,
                    Point(j,i), 5, Scalar
                    (0), 2, 8, 0);
                corners.push_back(Point(j
                    ,i));
            }
        }
    }
    // return the corners vector and the
        image with the corners marked
    tuple<vector<Point>, Mat> corners_img
        ;
    corners_img = make_tuple(corners,
        dst_norm_scaled);
    return corners_img;
}
```

D. Find Correspondences

```
vector<pair<Point, Point>>
    find_correspondences(Mat img1, Mat
    img2, vector<Point> corners1, vector<
    Point> corners2)
{   cout<<"find_correspondences"<<endl;
    // convert the images to grayscale
    // cvtColor(img1, img1,
        COLOR_BGR2GRAY);
    // cvtColor(img2, img2,
        COLOR_BGR2GRAY);
    Mat template1, template2;
    double Threshold = 0.8;
    Point p2 = Point(0,0);
    // initialize a vector to store the
        correspondences as a pair of
        points
    vector<pair<Point,Point>> corres;
    for(int i=0; i<corners1.size(); i++)
    {
        // check if the template is
            within the image
```

```
        //cout<<"Creating templates"<<
            endl;
        int WindowSize = 5;
        if(corners1[i].x - WindowSize/2
            <= 0 || corners1[i].x +
            WindowSize/2 >= img1.cols ||
            corners1[i].y - WindowSize/2
            <= 0 || corners1[i].y +
            WindowSize/2 >= img1.rows)
             continue;
        // create a roi around the corner
             point
        cv::Rect roi(corners1[i].x -
            WindowSize/2, corners1[i].y -
            WindowSize/2, WindowSize,
            WindowSize);
        template1 = img1(roi);
        p2 = Point(0,0);
        double currentMax = 0;
        for(int j=0; j<corners2.size(); j
            ++)
        {
            // check if the template is
                within the image
            if(corners2[j].x - WindowSize
                /2 <= 0 || corners2[j].x +
                 WindowSize/2 >= img2.cols
                 || corners2[j].y -
                WindowSize/2 <= 0 ||
                corners2[j].y + WindowSize
                /2 >= img2.rows)
                 continue;
            cv::Rect roi2(corners2[j].x -
                 WindowSize/2, corners2[j
                ].y - WindowSize/2,
                WindowSize, WindowSize);
            template2 = img2(roi2);
            double value = NCC(template1,
                 template2);
            if(value > Threshold && value
                 > currentMax){
                 currentMax = value;
                 p2 = corners2[j];
            }
        }
        // push the pair of points into
            the vector
        if (p2 != Point(0,0))
        {pair<Point, Point> p;
        p.first = corners1[i];
        p.second = p2;
        corres.push_back(p);}
    }
    return corres;
}
```

E. Find Homography from 4 correspondences

```
Mat findHomography(vector<Point> corners1
    , vector<Point> corners2)
{
    // create a matrix of 8x9
    Mat A = Mat::zeros(8, 9, CV_64F);
    // fill the matrix with the values
    for(int i=0; i<4; i++)
    {
        A.at<double>(2*i, 0) = corners1[i
            ].x;
        A.at<double>(2*i, 1) = corners1[i
            ].y;
        A.at<double>(2*i, 2) = 1;
        A.at<double>(2*i, 6) = -corners1[
            i].x*corners2[i].x;
        A.at<double>(2*i, 7) = -corners1[
            i].y*corners2[i].x;
        A.at<double>(2*i, 8) = -corners2[
            i].x;
        A.at<double>(2*i+1, 3) = corners1
            [i].x;
        A.at<double>(2*i+1, 4) = corners1
            [i].y;
        A.at<double>(2*i+1, 5) = 1;
        A.at<double>(2*i+1, 6) = -
            corners1[i].x*corners2[i].y;
        A.at<double>(2*i+1, 7) = -
            corners1[i].y*corners2[i].y;
        A.at<double>(2*i+1, 8) = -
            corners2[i].y;
    }
    // make the A matrix square
    Mat At = A.t();
    Mat AtA = At*A;
    // find the SVD of the matrix
    SVD svd(AtA);
    // the last column of the V matrix is
         the solution
    Mat h = svd.vt.row(8);
    // reshape the vector to a 3x3 matrix
    Mat H = h.reshape(1, 3);
    // normalize the matrix
    H = H/H.at<double>(2,2);
    return H;
}
```

F. RANSAC and best homography estimation

```
Mat RANSAC(vector<pair<Point, Point>>
    correspondingPoints, vector<pair<Point
    , Point>>& bestCorrespondingPoints)
{
    vector<pair<Point, Point>>
        tempCorrespondingPoints;
```

```
// initialize the best homography
   matrix
Mat bestH = Mat::zeros(3, 3, CV_64F);
// sample 4 points randomly from the
   vector of corresponding points
vector<Point> corners1, corners2;
int maxInliers = 0;
for(int i=0; i<1000; i++)
{
    corners1.clear();
    corners2.clear();
    for(int j=0; j<4; j++)
    {
        int index = rand() %
           correspondingPoints.size()
           ;
        corners1.push_back(
           correspondingPoints[index
           ].first);
        corners2.push_back(
           correspondingPoints[index
           ].second);
    }
    // find the homography matrix
       using the 4 points
    Mat H = findHomography(corners1,
       corners2);
    // find the inliers
    int inliers = 0;
    tempCorrespondingPoints.clear();
    for(int j=0; j<
       correspondingPoints.size(); j
       ++)
    {
        // transform the point in
           img1 using the homography
           matrix
        Mat p = Mat::zeros(3, 1,
           CV_64F);
        p.at<double>(0, 0) =
           correspondingPoints[j].
           first.x;
        p.at<double>(1, 0) =
           correspondingPoints[j].
           first.y;
        p.at<double>(2, 0) = 1;
        Mat p2 = H*p;
        p2 = p2/p2.at<double>(2, 0);
        // check if the transformed
           point is within a
           threshold distance from
           the corresponding point in
            img2
        if(sqrt(pow(p2.at<double>(0,
           0) - correspondingPoints[j
           ].second.x, 2) + pow(p2.at
           <double>(1, 0) -
           correspondingPoints[j].
           second.y, 2)) < 1)
            {
                inliers++;
                tempCorrespondingPoints
                   .push_back(
                   correspondingPoints
                   [j]);
            }
        }
        // if the number of inliers is
           greater than the previous best
           , update the best homography
           matrix
        if(inliers > maxInliers)
        {
            maxInliers = inliers;
            bestCorrespondingPoints =
               tempCorrespondingPoints;
            bestH = H;
        }
    }
    return bestH;
}
```

G. *Warp Images*

```
Mat warpImage(Mat img1, Mat img2, Mat H)
{
    cout<<"Warping the image..."<<endl;
    // find the corners of the image
    vector<Point> corners1, corners2;
    corners1.push_back(Point(0, 0));
    corners1.push_back(Point(img1.cols,
       0));
    corners1.push_back(Point(img1.cols,
       img1.rows));
    corners1.push_back(Point(0, img1.rows
       ));
    // transform the corners using the
       homography matrix
    cout<<"Transforming the corners..."<<
       endl;
    for(int i=0; i<4; i++)
    {
        Mat p = Mat::zeros(3, 1, CV_64F);
        p.at<double>(0, 0) = corners1[i].
           x;
        p.at<double>(1, 0) = corners1[i].
           y;
        p.at<double>(2, 0) = 1;
        Mat p2 = H*p;
        p2 = p2/p2.at<double>(2, 0);
```

```
        corners2.push_back(Point(p2.at<
            double>(0, 0), p2.at<double
            >(1, 0)));
    }
    cout<<"Done"<<endl;
    // find the minimum and maximum x and
         y values
    int minX = min(min(corners2[0].x,
        corners2[1].x), min(corners2[2].x,
         corners2[3].x));
    int maxX = max(max(corners2[0].x,
        corners2[1].x), max(corners2[2].x,
         corners2[3].x));
    int minY = min(min(corners2[0].y,
        corners2[1].y), min(corners2[2].y,
         corners2[3].y));
    int maxY = max(max(corners2[0].y,
        corners2[1].y), max(corners2[2].y,
         corners2[3].y));
    // create a new image with the size
        of the minimum and maximum values
    Mat img3 = Mat::zeros(maxY, maxX,
        img1.type());
    // warp the image using the
        homography matrix
    warpPerspective(img1, img3, H, img3.
        size());
    cout<<"Done"<<endl;
    // blend the second image onto the
        new image
    cout<<"Blending the images..."<<endl;
    //img2.copyTo(img3(Rect(0, 0, img2.
        cols, img2.rows)));
    // traverse the new image pixel by
        pixel
    for(int i=0; i<img3.rows; i++)
    {
        for(int j=0; j<img3.cols; j++)
        {
            // if the pixel is black,
                copy the pixel from the
                second image
            if(img3.at<Vec3b>(i, j) ==
                Vec3b(0, 0, 0))
            {
                if(i < img2.rows && j <
                    img2.cols)
                     img3.at<Vec3b>(i, j)
                        = img2.at<Vec3b>(i
                        , j);
            }

            // if the pixel is not black,
                average the pixel with
                the pixel from the second
                image
            else
            {
                if(i < img2.rows && j <
                    img2.cols)
                {
                    img3.at<Vec3b>(i, j)
                        [0] = (img3.at<
                        Vec3b>(i, j)[0] +
                        img2.at<Vec3b>(i,
                        j)[0])/2;
                    img3.at<Vec3b>(i, j)
                        [1] = (img3.at<
                        Vec3b>(i, j)[1] +
                        img2.at<Vec3b>(i,
                        j)[1])/2;
                    img3.at<Vec3b>(i, j)
                        [2] = (img3.at<
                        Vec3b>(i, j)[2] +
                        img2.at<Vec3b>(i,
                        j)[2])/2;
                }
            }
        }
    }
    return img3;
}
```

H. Extra Credit

```
Mat outputImage(Mat img1, Mat img2,
    vector<Point> dstPoints) {
// find the corners of the image
vector<Point> corners1, corners2;
corners1.push_back(Point(0, 0));
corners1.push_back(Point(img1.cols, 0));
corners1.push_back(Point(img1.cols, img1.
    rows));
corners1.push_back(Point(0, img1.rows));
// find the homography matrix for
    transforming the image 1 to a window
    of size dstPoints
Mat H1 = findHomography(corners1,
    dstPoints);
// duplicate the image 2 and warp image 1
     on top of the duplicate image 2
Mat img3 = img2.clone();
warpPerspective(img1, img3, H1, img3.size
    ());
// traverse the image pixel by pixel
for(int i=0; i<img3.rows; i++)
{
    for(int j=0; j<img3.cols; j++)
    {
        // if the pixel is black, copy
            the pixel from the second
            image
```

```
        if(img3.at<Vec3b>(i, j) == Vec3b
            (0, 0, 0))
        {
            if(i < img2.rows && j < img2.
                cols)
                img3.at<Vec3b>(i, j) =
                    img2.at<Vec3b>(i, j);
        }
    }
}
return img3;
}
```