Dixon Jin and Emerson Boyd
EECE 5639
11/3/17

Computer Vision Project 2 - Image Mosaicing

Abstract

The purpose of this project was to transform one image and overlay onto another in order to create a mosaic. The first step was to detect the edges and corners in both images, utilizing the Harris corner detector. Afterwards, the Harris R function was run over the image, and then a non-maximum suppression was produced to get a sparse set of corner features. Then, correspondences were found between the two images. Finally, from the correspondences, the homography was found between the two images. This homography was then used to warp one image onto the other in order to create a mosaic of the two given images.

Description of Algorithms

The first algorithm used was the Prewitt edge detector algorithm. Depending on whether or not the edge detector is vertical or horizontal, the filter would search for different edges, however, the algorithm behind them is the same. For both, the filter was a 3x3 matrix, and it calculated the value of the current pixel by taking the row or column of pixels after the current pixel and subtracted the row or column of pixels before the current pixel.

After coming up with horizontal and vertical Prewitt values for each pixel, the C matrix was calculated. The C matrix was a 2x2 matrix consisting of the vertical Prewitt value squared, the horizontal Prewitt value squared, and the vertical and horizontal Prewitt values multiplied together. After creating the C matrix, eigenvalues were calculated using built-in MatLab functions. Then, the determinant of the C matrix was calculated which is the R value for the given pixel. The R values were separated by a given threshold with those above a certain threshold would be given a 1 value, and values below the threshold would be given a 0 value.

The magnitude and orientation of the vertical and horizontal Prewitt values were also calculated. The magnitude was the square root of the horizontal Prewitt squared added to the vertical Prewitt squared. The orientation was the inverse tangent of the horizontal Prewitt divided by the vertical Prewitt. For orientation values that ended up being NaN, the pixel was set to 90.

Finally, the non-maximum suppression was used to come up with the local maxima within a given range of corners. Depending on the orientation of the pixel, different local maxima would be checked around the pixel. For example, if the orientation was 0, then the values to the left and right of the pixel would be used to find the local maxima. If there were values greater than

the current pixel value, then the value of the pixel was set to 0, otherwise, the pixel value would stay the same.

The non-maximum suppression was then used to calculate the normalized cross correlation. The normalized cross correlation compared the corners from the non-max suppression in the two separate images to see which ones matched up the best given a range. A 11x11 filter size was used to calculate normalized cross correlation values. The values that ended up with the highest values were used in the correlation values matrix for x and y. The correlation values gave the points that correlated most with the corners. For example, in the corrX matrix, at (10,10), it would give the X value in the second image that correlated the most with the corner in the first image. Combined with the corrY matrix, each corner pixel in the first image had a corner pixel in the second image that correlated the most and would be used for future calculations.

After calculating the correlation values for the two images, RANSAC was used to come up with a homography between the two images. RANSAC uses four pairs of points between the two images, and solves the system of linear equations to come up with the H matrix. Afterwards, every pair of points is used with the potential H matrix to come up with the number of inliers and outliers. This process is repeated with as many pairs of points as possible to come up with the H matrix that comes up with as few outliers as possible. Then, all outliers are removed from the data set, and a new H matrix is calculated using only the inliers. This is known as the least-squares homography and will be used to warp the source image.

The final algorithm used in this project was the warping of the source image and overlaying it over the destination image. To do this, the least-squares H matrix is applied to the source image. Then, another homography is used to calculate how far the source image needs to translate to match up to the destination image. For the final image, values are taken either from the source image or the destination image. Whenever there are values that can be used from both images, the average of the two is taken so that the pixels will be blended together. The result is a mosaic of the two given images.

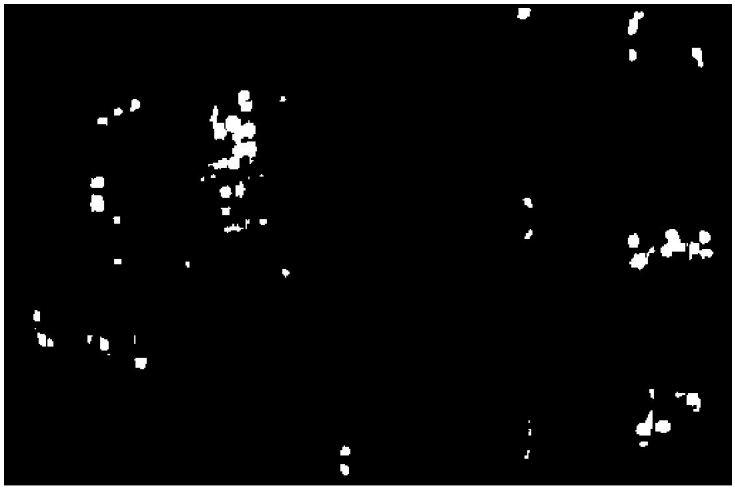
The full path of algorithms visualized can be seen in Appendix A1.

Experiments, Values of Parameters Used, and Observations

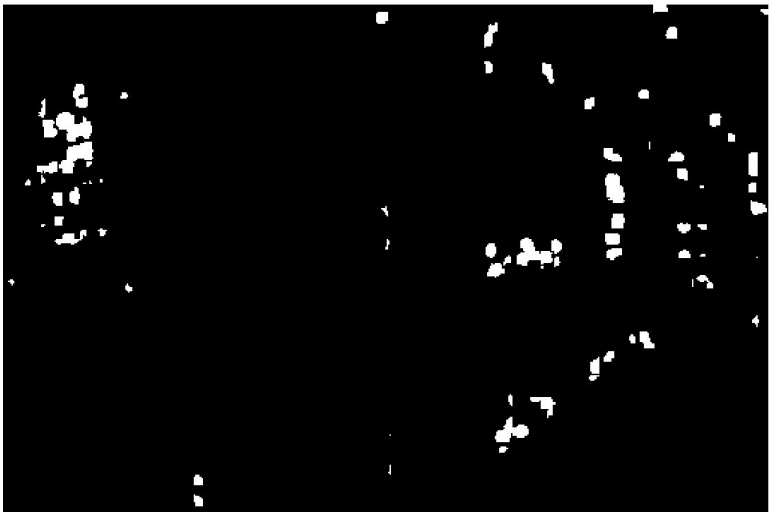
These experiments were run for every combination of images for each set of image samples given. Only one example is given for each operation, though, to demonstrate the general effects of each operation. Each operation was applied the same way and had similar outcomes for each set of images unless otherwise stated.

Below, Image 1 refers to DSC_0281.JPG and Image 2 refers to DSC_0282.JPG as given in the image set samples.

As the project document requested, below is a screenshot of the corners detected through the Harris corner detection algorithm:



R Corner Detection of Image 1



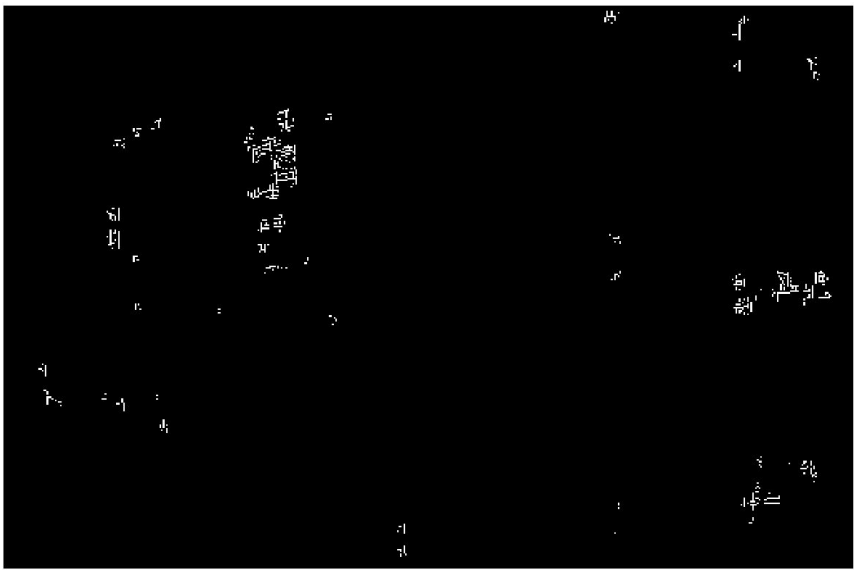
R Corner Detection of Image 2

We can see matching corner features between these two images. In the top left of Image 2, there is a large patch of corners that corresponds to the corner patch more toward the center of Image 1. Similarly, in the center right of Image 1, there is a corner patch that correlates to a corner patch toward the center of Image 2.

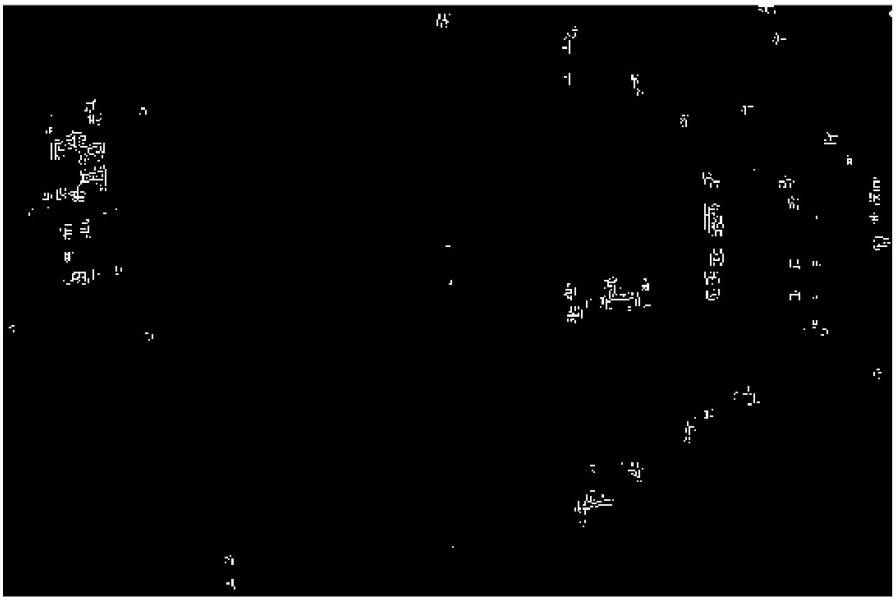
One distinct feature of these images to note out is how imprecise the corners are. The corners are essentially blobs, rather than sharp lines. It is for this reason that we need to perform non-max suppression on all of the corners detected through this method in order to get more

well-defined corners. Although humans can detect matching features between the two images, computers need more precise edges in order to perform NCC accurately.

Below are the results of performing non-max suppression after the Harris corner detection:



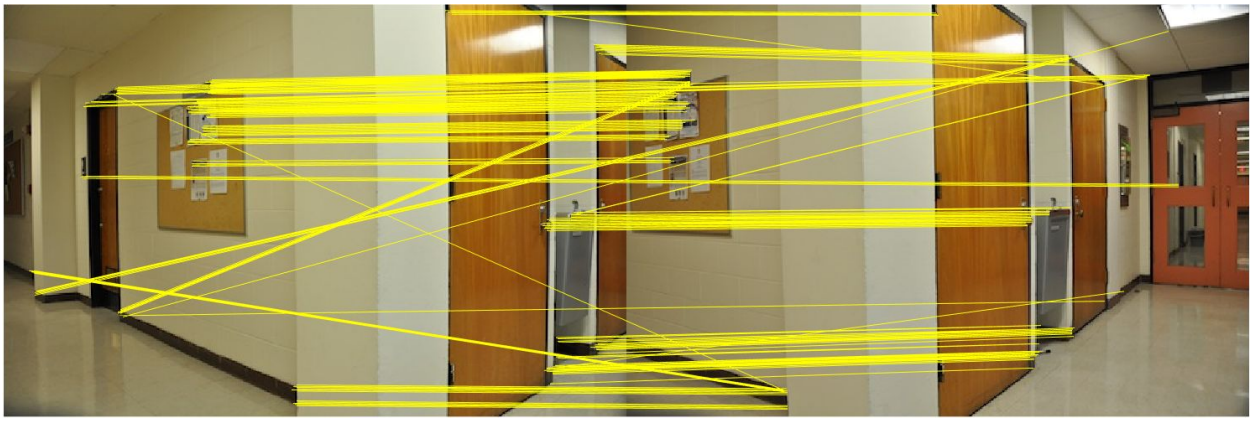
Non-Max Suppression of Image 1



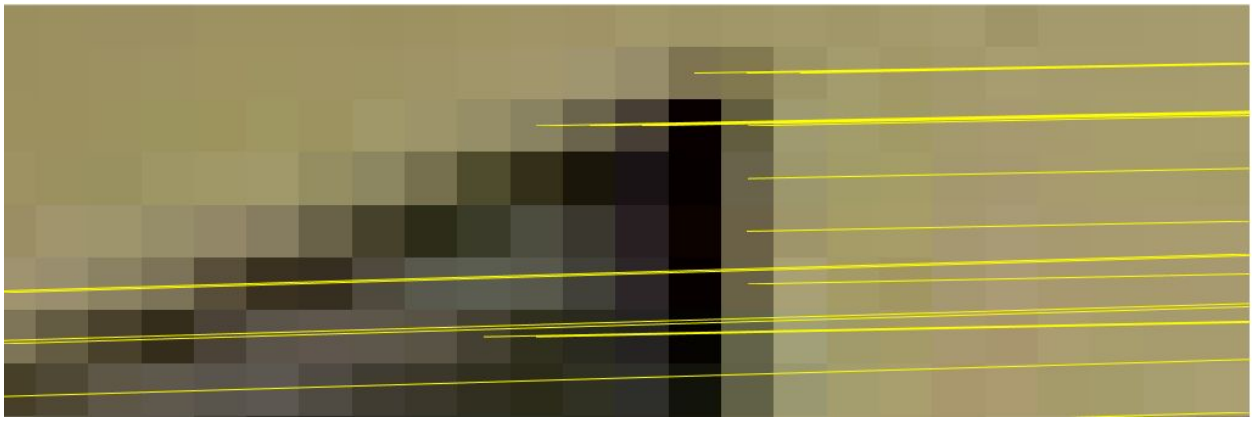
Non-Max Suppression of Image 2

The non-max suppression provides a much more detailed description of the orientation of each corner. Having orientation of corners rather than just location helps to perform normalized cross-correlation more effectively. The corresponding patches of corners mentioned above are still present, as well (the top left of Image 2 matches with the center patch in Image 1, for example).

Below are the results of performing normalized cross-correlation on Image 1 and Image 2 by using the results of non-max suppression. The two zoomed in images are of the dark sign located above the bulletin board:



Normalized Cross-Correlation of Image 1 and Image 2



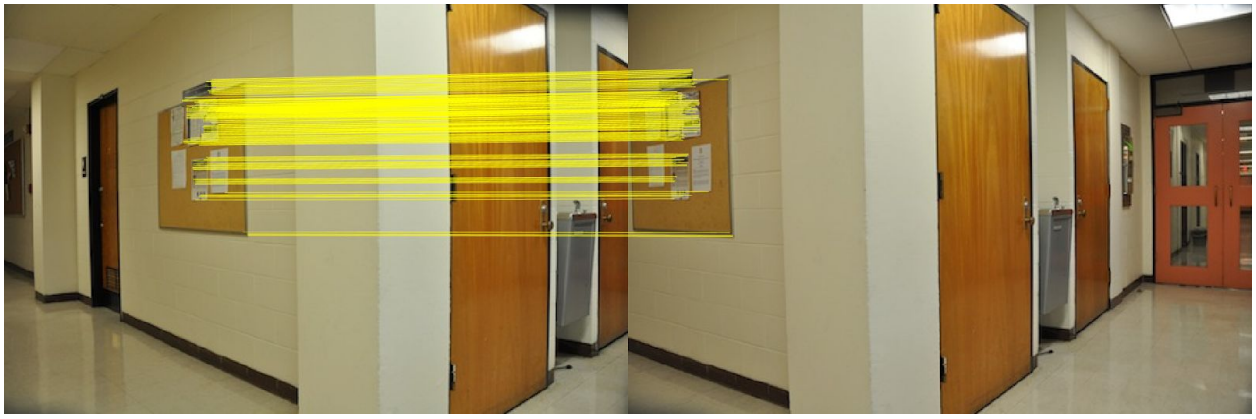
Zoomed-In NCC Patch of Image 1

Zoomed-In NCC Patch of Image 2

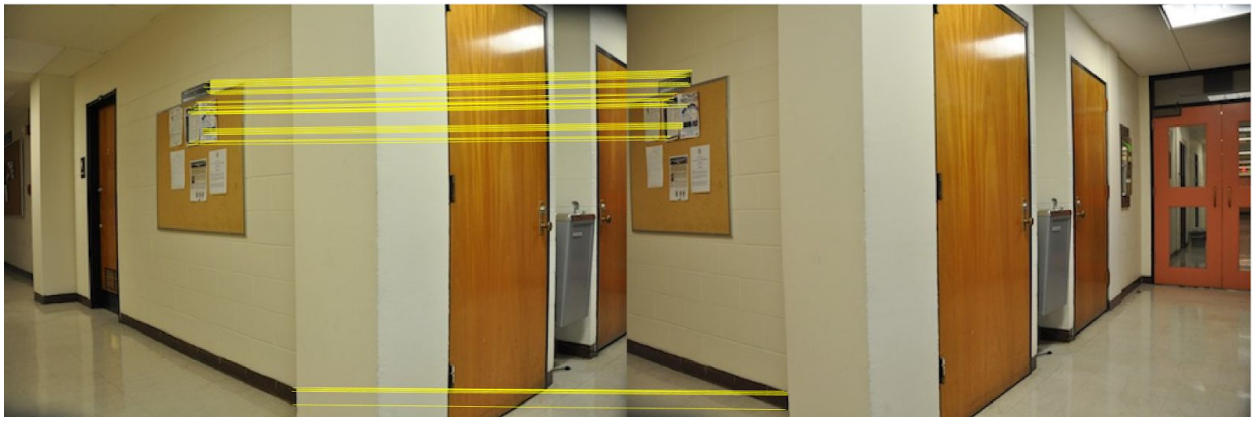
There are many points that correspond from Image 1 to Image 2. Upon observing the zoomed-in images of one of the mappings, there are clearly some points that align perfectly. There are several pixels on the very right of the sign above the bulletin board that map perfectly from Image 1 to Image 2. A handful of other points, as seen in the zoomed-out image, are very clearly mis-correlated.

Within the code, the matrices that contain mappings from corners in Image1 to corners in Image2 are called “roughInliers” due to the fact that there are the handful corner mappings that aren’t accurate mappings. It is for this reason that we need to apply the RANSAC method to develop a true homography matrix and only retain true inlier mappings.

Below are two applications of the RANSAC method, each with a different NCC score threshold being applied to the NCC part of the algorithm before performing RANSAC:



RANSAC Inliers of Image 1 and Image 2 Using NCC Score Threshold = 1.00



RANSAC Inliers of Image 1 and Image 2 Using NCC Score Threshold = 1.04

For the NCC threshold of 1.00, it is clear to see that RANSAC only maintains point mappings from one section of each image. Such a result would lead to an inaccurate H matrix solution, as the H matrix is a result of the transformation from these points to the other image. It is for this reason that a wider range of lines was desired to produce a more accurate H matrix, which was obtained from using a larger NCC threshold. The “best” NCC threshold that we found for these images was 1.04.

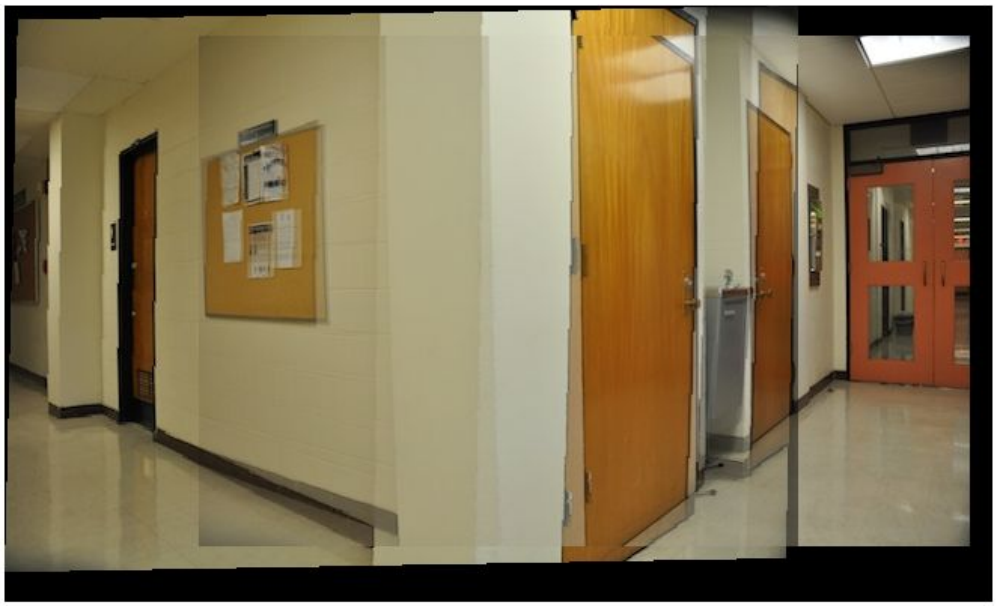
In the code, these point mappings are called “inliers” due to the fact that they are all accurate mappings from points in one image to points in the next.

Below is an image of Image 1 warped to fit the perspective of Image 2 based on the calculated H matrix from the RANSAC algorithm:

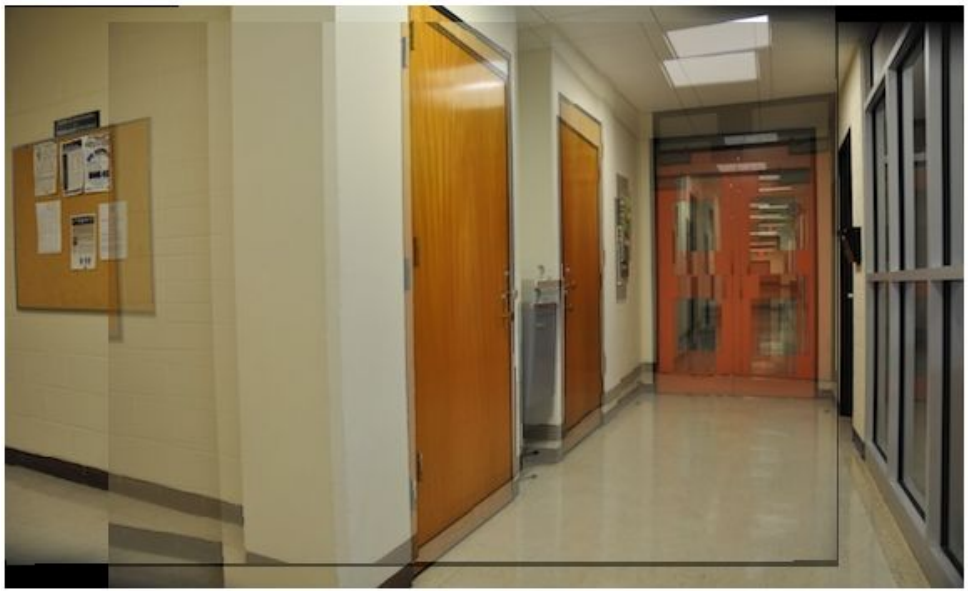
Image 1 Warped to the Perspective of Image 2

It is difficult to tell how accurately this image has warped to the perspective of Image 2 without blending the two images. The blending of the two images is shown below.

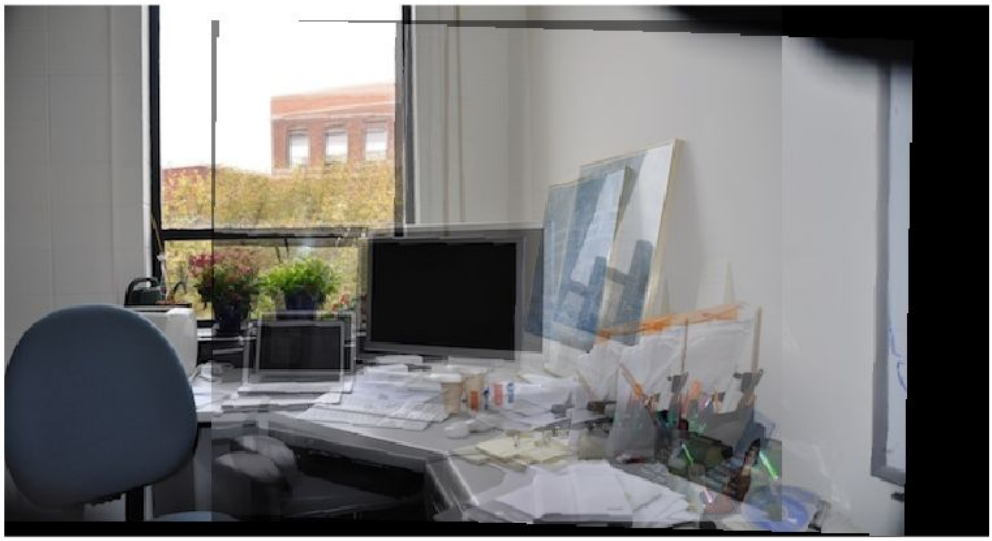
Below are the three blended images from the three different example images, each with a different NCC threshold to account for the similarity in points:



Blended images of Set #1



Blended images of Set #2



Blended images of Set #3

Clearly, all three sets of the blended images are a little off. However, the issue might stem from the H matrix since one part of the image usually matches, but the rest of the image does not, indicating that the transform might not been perfect.

The NCC thresholds for the three sets of images were varied because some images had more corners that correlated with corners in the second image. Therefore, the by making the threshold more or less restrictive, the code would properly filter out correlations that were not as strong as other correlations. Both the source and destination image can be seen in the ending image because the blend took the average of the two different images. Where the image lines up better, the blend is better because the pixel values were closer, but where they do not line up, the blend shows features from both.

Conclusion

It is important to note that, in order to develop the strongest corner detection/correlation, it is necessary to modify the R threshold and the NCC score threshold for each set of images. Also, the “best” threshold values is very much up to interpretation by the user within a range of values. For us, we liked the looks of the corner detection values that we obtained the most with an R threshold of 10,000,000 and an NCC score threshold of 1.04 for the first set of images. Those values might vary for another person. However, no matter the threshold, there should still be an H matrix that correctly correlates the points on one image to the points on another image.

Our images clearly do not align perfectly, however. In terms of the flow of our project, there is very precise corner correlation between the two images, but the image goes askew after the warping. Therefore, there is either something incorrect with the establishment of our homography matrix, or there is a bug in the production of our warped image. Due to several small bugs throughout the program, the problem took several days to narrow down. Had we been given more time on the project, we would have sought outside help to determine the solution to our problem.

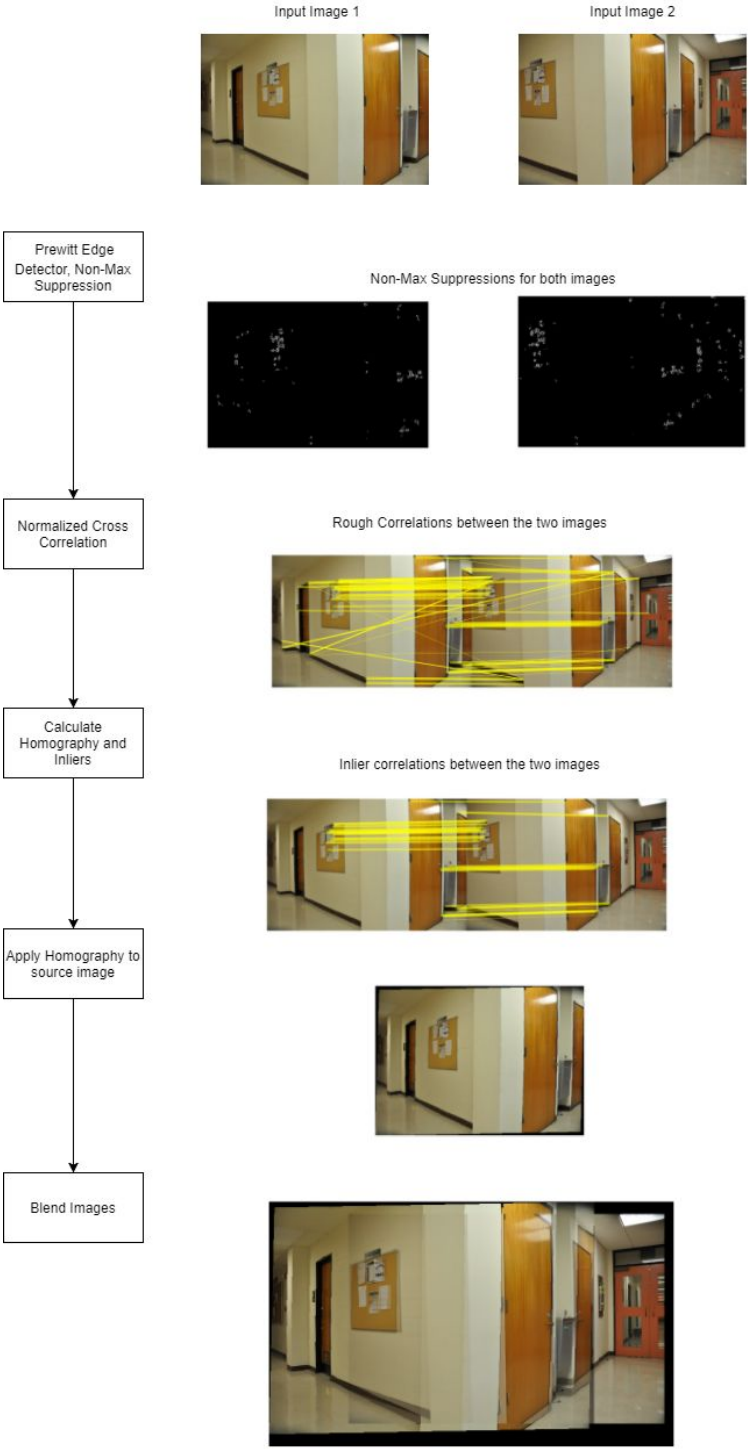
If the image warpings had been precise, the same process could be repeated for all sets of images, such that the blended image of Image 1 and Image 2 could then have corner detection applied on it alongside Image 3. The original blended image could then be blended with Image 3, and so on and so forth with each following image. Ideally, a panorama could be created of all of the images.

This project gives us a greater appreciation for the panorama features available in most smartphones today. There is a great deal of math that lies underneath determining how two images might overlay on one-another. In addition, most smartphones are able to process the overlapping of several high-resolution images within seconds of capturing the images. For us, it takes about 60 seconds to process the overlap of two medium-resolution images. There is clearly much more that can be optimized in terms of our RANSAC algorithm, which is the bottleneck of our project. That being said, smartphones, when taking panoramas, have information about how much the phone has been rotated and translated between image captures. Having such information in this project would provide us with greater knowledge about how the coordinate system of one image would map to the other.

One potential future experiment that evolves from the findings of this experiment would be to re-perform this project with given information between the photos like smartphones have. These features could include the rotation of the phone between each photo (in YAW) as well as the intrinsic parameters of the camera. We would then compare and contrast results from the original mosaic project and the new panorama project, including, but not limited to: blending accuracy, complexity of algorithm, and execution time differences between the two.

Appendix

A1. Flow Chart



```
clc
clear all
close all

warning off; % warnings were spawning from the linsolve(A,B) calls

tic

imageType = 1;

imageScale = 1;

if (imageType == 1)
    folderName = 'Images/DanaHallway1/';
    fileNamePrefix = 'DSC_0';
    fileNameSuffix = '.JPG';
    fileNumberMin = 281;
    fileNumberMax = 283;
    rThreshold = 10000000;
    nccThreshold = 1.04;
elseif (imageType == 2)
    folderName = 'Images/DanaHallway2/';
    fileNamePrefix = 'DSC_0';
    fileNameSuffix = '.JPG';
    fileNumberMin = 285;
    fileNumberMax = 287;
    rThreshold = 10000000;
    nccThreshold = 1.03;
elseif (imageType == 3)
    folderName = 'Images/DanaOffice/';
    fileNamePrefix = 'DSC_0';
    fileNameSuffix = '.JPG';
    fileNumberMin = 308;
    fileNumberMax = 310;
    rThreshold = 100000000;
    nccThreshold = 1.07;
end

grayImgList = [];

numImages = 0;
for i = fileNumberMin:fileNumberMax
    fullFileName = strcat(folderName, fileNamePrefix, sprintf('%3d',
 i), fileNameSuffix);
    img = imread(fullFileName);

    numImages = numImages + 1;
    rgbImgList(:, :, :, numImages) = double(img); % we need doubles so
 that we can perform necessary calculations later
    grayImgList(:, :, numImages) = rgb2gray(img);
end
```

```
% Harris Corner Detection, Non-Max Suppression and NCC
[corrX, corrY] = getCorners(grayImgList(:,:,1), grayImgList(:,:,2),
 rThreshold, nccThreshold);

plotCorrelations(rgbImgList(:,:,:,1), rgbImgList(:,:,:,2), corrX,
 corrY);

% RANSAC
[ roughH, roughInliersX, roughInliersY ] = ransac2(grayImgList(:,:,1),
 grayImgList(:,:,2), corrX, corrY);
[ H, inliersX, inliersY ] = ransac2(grayImgList(:,:,1),
 grayImgList(:,:,2), roughInliersX, roughInliersY);

figure
plotCorrelations(rgbImgList(:,:,:,1), rgbImgList(:,:,:,2), inliersX,
 inliersY);

% Image Warping
warpedImage = warpImage(rgbImgList(:,:,:,1), H);

figure;
imshow(warpedImage/255);

grayWarpedImage = round(rgb2gray(warpedImage/255)*255);

% Find the Corners of the Warped Image Matching with the Second Image
[corrXWarped, corrYWarped] = getCorners(grayWarpedImage,
 grayImgList(:,:,2), rThreshold, nccThreshold);
[ roughHWarped, roughInliersWarpedX, roughInliersWarpedY ]
 = ransac2(grayWarpedImage, grayImgList(:,:,2), corrXWarped,
 corrYWarped);
[ HWarped, inliersXWarped, inliersYWarped ] = ransac2(grayWarpedImage,
 grayImgList(:,:,2), roughInliersWarpedX, roughInliersWarpedY);
XOffset = round(HWarped(1, 3));
YOffset = round(HWarped(2, 3));

% Image Blending
blendedImage = blendImages(warpedImage, rgbImgList(:,:,:,2), XOffset,
 YOffset);
figure
imshow(blendedImage/255);

timeElapsed = toc
```

Published with MATLAB® R2017a

```
function [corrX, corrY] = getCorners(image1, image2, rThreshold,
 nccThreshold)
    verticalPrewittFilter = [-1 0 1;
                             -1 0 1;
                             -1 0 1];

    horizontalPrewittFilter = [-1 -1 -1;
                               0 0 0;
                               1 1 1];

    % Harris Corner Detection
    verticalPrewitt1 = imfilter(image1,
 verticalPrewittFilter, 'replicate');
    horizontalPrewitt1 = imfilter(image1,
 horizontalPrewittFilter, 'replicate');

    verticalPrewitt2 = imfilter(image2,
 verticalPrewittFilter, 'replicate');
    horizontalPrewitt2 = imfilter(image2,
 horizontalPrewittFilter, 'replicate');

    R1 = CMatrix(verticalPrewitt1, horizontalPrewitt1);
    R2 = CMatrix(verticalPrewitt2, horizontalPrewitt2);

    magnitude1 = sqrt(horizontalPrewitt1.^2 + verticalPrewitt1.^2);
    orientation1 = atand(horizontalPrewitt1./verticalPrewitt1);
    magnitude2 = sqrt(horizontalPrewitt2.^2 + verticalPrewitt2.^2);
    orientation2 = atand(horizontalPrewitt2./verticalPrewitt2);

    R1(R1 < rThreshold) = 0;
    R1(R1 >= rThreshold) = 255;
    orientation1(isnan(orientation1)) = 90; % get rid of bad
 orientation values

    R2(R2 < rThreshold) = 0;
    R2(R2 >= rThreshold) = 255;
    orientation2(isnan(orientation2)) = 90; % get rid of bad
 orientation values

    % Non-Max Suppression
    nonMaxSuppress1 = nonMaxSuppression(orientation1, magnitude1, R1);
    nonMaxSuppress2 = nonMaxSuppression(orientation2, magnitude2, R2);

    % NCC
    [corrX, corrY] = normalizedCrossCorrelation(image1, image2,
 nonMaxSuppress1, nonMaxSuppress2, nccThreshold);
end
```

Published with MATLAB® R2017a

```
function [DetMatrix] = CMatrix(verticalPrewitt, horizontalPrewitt)
    verticalHorizontalPrewitt = verticalPrewitt .* horizontalPrewitt;

    verticalPrewittSquared = verticalPrewitt .* verticalPrewitt;

    horizontalPrewittSquared = horizontalPrewitt .* horizontalPrewitt;

    boxFilterSize = 7;
    boxFilter = ones(boxFilterSize, boxFilterSize)./(boxFilterSize *
 boxFilterSize);

    boxFilterVerticalHorizontalPrewitt =
 imfilter(verticalHorizontalPrewitt, boxFilter);
    boxFilterVerticalPrewitt = imfilter(verticalPrewittSquared,
 boxFilter);
    boxFilterHorizontalPrewitt = imfilter(horizontalPrewittSquared,
 boxFilter);

    sizeMatrix = size(verticalHorizontalPrewitt);
    sizeX = sizeMatrix(1);
    sizeY = sizeMatrix(2);

    DetMatrix = zeros(sizeX, sizeY);
    for i = 1:sizeX
        for j = 1:sizeY
            C = [boxFilterVerticalPrewitt(i, j)
 boxFilterVerticalHorizontalPrewitt(i, j);
                boxFilterVerticalHorizontalPrewitt(i, j)
 boxFilterHorizontalPrewitt(i, j)];
            D = eig(C);
            det = D(1) * D(2);
            trace = D(1) + D(2);
            DetMatrix(i,j) = det - (0.05 * (trace^2));
        end
    end
end
```

Published with MATLAB® R2017a

```
function [outputMatrix] = nonMaxSuppression(orientation, magnitude, R)
    totalSize = size(orientation);
    sizeX = totalSize(1);
    sizeY = totalSize(2);

    outputMatrix = R;

    for i = 2:sizeX-1
        for j = 2:sizeY-1
            angle = closestAngle(orientation(i,j));
            if (angle == 0)
                compareIndex1 = magnitude(i, j-1);
                compareIndex2 = magnitude(i, j+1);
            elseif (angle == 45)
                compareIndex1 = magnitude(i-1, j+1);
                compareIndex2 = magnitude(i+1, j-1);
            elseif (angle == 90)
                compareIndex1 = magnitude(i-1, j);
                compareIndex2 = magnitude(i+1, j);

            elseif (angle == 135)
                compareIndex1 = magnitude(i-1, j-1);
                compareIndex2 = magnitude(i+1, j+1);
            end
            if (or((magnitude(i,j) < compareIndex1),(magnitude(i,j) <
 compareIndex2)))
                outputMatrix(i,j) = 0;
            else
                outputMatrix(i,j) = R(i,j);
            end
        end
    end
end
```

Published with MATLAB® R2017a

```
function [angle] = closestAngle(orientation)
    compareArray = [0 45 90 135];
    if (abs(mod(orientation,180)) >= (180-22.5))
        angle = 0;
    else
        for i = 1:4
            if (abs(mod(orientation,180)-compareArray(i)) <= 22.5)
                angle = compareArray(i);
            end
        end
    end
end
```

Published with MATLAB® R2017a

```
function [corrX, corrY] = normalizedCrossCorrelation(grayImg1,
 grayImg2, nonMaxSuppress1, nonMaxSuppress2, nccThreshold)
    nccBoxFilterSize = 11;

    totalImageSize1 = size(grayImg1);
    imageHeight1 = totalImageSize1(1);
    imageWidth1 = totalImageSize1(2);
    totalImageSize2 = size(grayImg2);
    imageHeight2 = totalImageSize2(1);
    imageWidth2 = totalImageSize2(2);

    nccImg1 = zeros(size(grayImg1));
    nccImg2 = zeros(size(grayImg2));

    halfNccBoxFilterSizeFloored = floor(nccBoxFilterSize / 2);
    xMin = nccBoxFilterSize - halfNccBoxFilterSizeFloored;
    yMin = xMin;
    xMax1 = imageHeight1 - halfNccBoxFilterSizeFloored;
    yMax1 = imageWidth1 - halfNccBoxFilterSizeFloored;
    xMax2 = imageHeight2 - halfNccBoxFilterSizeFloored;
    yMax2 = imageWidth2 - halfNccBoxFilterSizeFloored;

    for x = xMin:xMax1
        for y = yMin:yMax1
            if (nonMaxSuppress1(x, y) == 255)
                xRange = x - halfNccBoxFilterSizeFloored : x +
 halfNccBoxFilterSizeFloored;
                yRange = y - halfNccBoxFilterSizeFloored : y +
 halfNccBoxFilterSizeFloored;
                grayImg1Snippet = grayImg1(xRange, yRange);
                nccImg1(x, y) = norm(grayImg1Snippet);
            end
        end
    end

    for x = xMin:xMax2
        for y = yMin:yMax2
            if (nonMaxSuppress2(x, y) == 255)
                xRange = x - halfNccBoxFilterSizeFloored : x +
 halfNccBoxFilterSizeFloored;
                yRange = y - halfNccBoxFilterSizeFloored : y +
 halfNccBoxFilterSizeFloored;
                grayImg2Snippet = grayImg2(xRange, yRange);
                nccImg2(x, y) = norm(grayImg2Snippet);
            end
        end
    end

    corrX = zeros(size(grayImg1));
    corrY = zeros(size(grayImg1));

    for x1 = xMin:xMax1
```

```
        for y1 = yMin:yMax1
            if (nonMaxSuppress1(x1,y1) == 255)
                x1Range = x1 - halfNccBoxFilterSizeFloored : x1 +
halfNccBoxFilterSizeFloored;
                y1Range = y1 - halfNccBoxFilterSizeFloored : y1 +
halfNccBoxFilterSizeFloored;
                grayImg1Snippet = grayImg1(x1Range, y1Range);

                c = zeros(size(grayImg2));

                for x2 = xMin:xMax2
                    for y2 = yMin:yMax2
                        if (nonMaxSuppress2(x2,y2) == 255)
                            x2Range = x2 -
halfNccBoxFilterSizeFloored : x2 + halfNccBoxFilterSizeFloored;
                            y2Range = y2 -
halfNccBoxFilterSizeFloored : y2 + halfNccBoxFilterSizeFloored;
                            grayImg2Snippet = grayImg2(x2Range,
y2Range);

                            ccPoint = grayImg1Snippet .*
grayImg2Snippet;
                            nccPoint = sum(sum(ccPoint)) /
(nccImg1(x1,y1) * nccImg2(x2,y2));
                            c(x2,y2) = nccPoint;
                        end
                    end
                end

                [cMaxVal, cMaxIndex] = max(c(:));
                if (cMaxVal > nccThreshold)
                    [cMaxX, cMaxY] = ind2sub(size(c), cMaxIndex);
                    corrX(x1,y1) = cMaxX;
                    corrY(x1,y1) = cMaxY;
                end
            end
        end
    end
end
```

Published with MATLAB® R2017a

```
function [] = plotCorrelations(grayImg1, grayImg2, corrX, corrY)

totalImageSize1 = size(grayImg1);
imageHeight1 = totalImageSize1(1);
imageWidth1 = totalImageSize1(2);

grayImgFull = horzcat(grayImg1, grayImg2);

totalImageSizeFull = size(grayImgFull);
imageHeightFull = totalImageSizeFull(1);
imageWidthFull = totalImageSizeFull(2);

imshow(grayImgFull/255)
hold on
for x = 1:imageHeight1
    for y = 1:imageWidth1
        if ((corrX(x,y) ~= 0) && (corrY(x,y) ~= 0))
            img2XCoordinate = imageWidth1 + corrY(x,y);
            line([y img2XCoordinate], [x
 corrX(x,y)], 'Color', 'yellow')
        end
    end
end

axis([1 imageWidthFull 1 imageHeightFull])
daspect([1 1 1])

hold off

end
```

Published with MATLAB® R2017a

```
function [ H, inliersX, inliersY ] = ransac2( imgList1, imgList2,
 corrX, corrY )
    totalImageSize1 = size(imgList1);
    imageHeight1 = totalImageSize1(1);
    imageWidth1 = totalImageSize1(2);

    j = 0;
    for x = 1:imageHeight1
        for y = 1:imageWidth1
            if ((corrX(x,y) ~= 0) && (corrY(x,y) ~= 0))
                j = j + 1;
                img1(j,:) = [x y];
                img2(j,:) = [corrX(x,y) corrY(x,y)];
            end
        end
    end

    bestH = zeros(3,3);
    bestHNumCorrect = -1;
    bestInliers1 = zeros(1,2);
    bestInliers2 = zeros(1,2);

    for i = 1:(j*10)
        rng('shuffle');
        loc1 = ceil(rand() * j);
        loc2 = ceil(rand() * j);
        loc3 = ceil(rand() * j);
        loc4 = ceil(rand() * j);

        x1 = img1(loc1, 1);
        y1 = img1(loc1, 2);
        xNew1 = img2(loc1, 1);
        yNew1 = img2(loc1, 2);
        x2 = img1(loc2, 1);
        y2 = img1(loc2, 2);
        xNew2 = img2(loc2, 1);
        yNew2 = img2(loc2, 2);
        x3 = img1(loc3, 1);
        y3 = img1(loc3, 2);
        xNew3 = img2(loc3, 1);
        yNew3 = img2(loc3, 2);
        x4 = img1(loc4, 1);
        y4 = img1(loc4, 2);
        xNew4 = img2(loc4, 1);
        yNew4 = img2(loc4, 2);

        potentialH = generateHFromPoints(x1, y1, xNew1, yNew1, x2, y2,
 xNew2, yNew2, x3, y3, xNew3, yNew3, x4, y4, xNew4, yNew4);
        [potentialHNumCorrect, potentialInliers1, potentialInliers2] =
 calculateNumCorrectPoints(img1, img2, potentialH);

        if (potentialHNumCorrect > bestHNumCorrect)
```

```
            bestH = potentialH;
            bestHNumCorrect = potentialHNumCorrect;
            bestInliers1 = potentialInliers1;
            bestInliers2 = potentialInliers2;
        end
    end

    H = bestH;
    inliers1 = bestInliers1;
    inliers2 = bestInliers2;

    totalInliersSize = size(inliers1);
    inliersLength = totalInliersSize(1);

    inliersX = zeros(size(imgList1));
    inliersY = zeros(size(imgList2));

    for k = 1:inliersLength
        x = inliers1(k, 1);
        y = inliers1(k, 2);

        inliersX(x,y) = inliers2(k,1);
        inliersY(x,y) = inliers2(k,2);
    end
end
```

Published with MATLAB® R2017a

```
function [H] = generateHFromPoints(x1, y1, xNew1, yNew1, x2, y2,
 xNew2, yNew2, x3, y3, xNew3, yNew3, x4, y4, xNew4, yNew4)
    A1 = [x1 y1 1; x2 y2 1; x3 y3 1; x4 y4 1];
    B1 = [xNew1; xNew2; xNew3; xNew4];
    A2 = A1;
    B2 = [yNew1; yNew2; yNew3; yNew4];
    A3 = A2;
    B3 = [1; 1; 1; 1];

    H1 = linsolve(A1,B1);
    H2 = linsolve(A2,B2);
    H3 = linsolve(A3,B3);
    H = [transpose(H1); transpose(H2); transpose(H3)];
end
```

Published with MATLAB® R2017a

```
function [numCorrect, inliers1, inliers2] =
 calculateNumCorrectPoints(points1, points2, H)

totalSizePoints = size(points1);
numPoints = totalSizePoints(1);

numCorrect = 0;
j = 0;

inliers1 = zeros(1,2);
inliers2 = zeros(1,2);

accuracyThreshold = 1;

for i = 1:numPoints
    pointOld = [points1(i,1); points1(i,2); 1];
    pointNew = H * pointOld;
    if (abs(points2(i,1) - pointNew(1)) <= accuracyThreshold &&
 abs(points2(i,2) - pointNew(2)) <= accuracyThreshold)
        j = j + 1;
        inliers1(j,1) = points1(i,1);
        inliers1(j,2) = points1(i,2);
        inliers2(j,1) = points2(i,1);
        inliers2(j,2) = points2(i,2);
        numCorrect = numCorrect + 1;
    end
end
```

Published with MATLAB® R2017a

```
function [ warpedImage ] = warpImage( rgbImage, H )
    totalImageSize = size(rgbImage);
    imageHeight = totalImageSize(1);
    imageWidth = totalImageSize(2);

    dummyVal = -100000;
    minX = dummyVal;
    minY = dummyVal;
    maxX = dummyVal;
    maxY = dummyVal;

    for x = 1:imageHeight
        for y = 1:imageWidth
            newPoints = H * [x; y; 1];
            newX = round(newPoints(1));
            newY = round(newPoints(2));
            if (newX < minX || minX == dummyVal)
                minX = newX;
            end
            if (newY < minY || minY == dummyVal)
                minY = newY;
            end
            if (newX > maxX || maxX == dummyVal)
                maxX = newX;
            end
            if (newY > maxY || maxY == dummyVal)
                maxY = newY;
            end
        end
    end

    if (minX < 0)
       offsetX = minX * -1 + 1;
       offsetY = minY * -1 + 1;
    else
       offsetX = minX;
       offsetY = minY;
    end

    warpedImageSize = [(maxX - minX + 1) (maxY - minY + 1) 3];
    warpedImage = zeros(warpedImageSize);
    warpedImageHeight = warpedImageSize(1);
    warpedImageWidth = warpedImageSize(2);

    for x = 1:warpedImageHeight
        for y = 1:warpedImageWidth
            oldPoints = zeros(2, 1);
            if (minX < 0)
                oldPoints = H \ [x - offsetX; y - offsetY; 1];
            else
                oldPoints = H \ [x + offsetX; y + offsetY; 1];
            end
```

```
            oldX = round(oldPoints(1));
            oldY = round(oldPoints(2));

            if (oldX >= 1 && oldX <= imageHeight && oldY >=1 && oldY
 <= imageWidth)
                warpedImage(x, y, :) = rgbImage(oldX, oldY, :);
            end
        end
    end
end
```

Published with MATLAB® R2017a

```
function [newWarpedImage] = blendImages(image1, image2, XOffset,
 YOffset)
    sizeImage1 = size(image1);
    sizeImage1X = sizeImage1(1);
    sizeImage1Y = sizeImage1(2);
    sizeImage2 = size(image2);
    sizeImage2X = sizeImage2(1);
    sizeImage2Y = sizeImage2(2);

    image1XOffset = 0;
    image1YOffset = 0;
    image2XOffset = 0;
    image2YOffset = 0;

    if (XOffset < 0)
        image2XOffset = abs(XOffset);
    else
        image1XOffset = XOffset;
    end
    if (YOffset < 0)
        image2YOffset = abs(YOffset);
    else
        image1YOffset = YOffset;
    end

    newWarpedImageX = max(sizeImage1X, sizeImage2X) + abs(XOffset);
    newWarpedImageY = max(sizeImage1Y, sizeImage2Y) + abs(YOffset);
    newWarpedImage = ones(newWarpedImageX, newWarpedImageY, 3) * -1;

    if (XOffset < 0 && YOffset < 0)
        for i = 1:sizeImage1X
            for j = 1:sizeImage1Y
                newWarpedImage(i, j, :) = image1(i, j, :);
            end
        end

        for k = 1:sizeImage2X
            for l = 1:sizeImage2Y
                if (newWarpedImage(k + abs(image2XOffset), l +
 abs(image2YOffset), :) == -1)
                    newWarpedImage(k + abs(image2XOffset), l +
 abs(image2YOffset), :) = image2(k,l, :);
                else
                    pixelAverage = (image2(k,l, :) + newWarpedImage(k
 + abs(image2XOffset), l + abs(image2YOffset), :)) / 2;
                    newWarpedImage(k + abs(image2XOffset), l +
 abs(image2YOffset), :) = pixelAverage;
                end
            end
        end
    end
```

```
    if (XOffset > 0 && YOffset > 0)
        for i = 1:sizeImage2X
            for j = 1:sizeImage2Y
                newWarpedImage(i, j, :) = image2(i, j, :);
            end
        end

        for k = 1:sizeImage1X
            for l = 1:sizeImage1Y
                if (newWarpedImage(k + abs(image1XOffset), l +
 abs(image1YOffset), :) == -1)
                    newWarpedImage(k + abs(image1XOffset), l +
 abs(image1YOffset), :) = image1(k,l, :);
                else
                    pixelAverage = (image1(k,l, :) + newWarpedImage(k
 + abs(image1XOffset), l + abs(image1YOffset), :)) / 2;
                    newWarpedImage(k + abs(image1XOffset), l +
 abs(image1YOffset), :) = pixelAverage;
                end
            end
        end
    end

    newWarpedImage(newWarpedImage == -1) = 0;
end
```

Published with MATLAB® R2017a