Image Mosaicing

Sahasrajit Anantharamakrishnan[1], **Eli MacColl**[2]

[1]Department of Electrical and Computer Engineering, anantharamakrishn.sa@northeastern.edu
[2]Department of Electrical and Computer Engineering, maccoll.e@northeastern.edu

March 17, 2023

Abstract

In this project, we aim to develop a computer vision algorithm for image mosaicing with two images by applying a Harris corner detector, identifying corresponding features, and estimating a homography between them. The homography describes how one image can be warped into the coordinate system of the second image to create a mosaic that has the union of all pixels in the two images where overlapping pixels are blended via a blending scheme. Normalized cross-correlation (NCC) is used to find corresponding features between two images such that only the best matches for each corner feature are considered. The homography is estimated using the random sample consensus (RANSAC) algorithm which iteratively identifies the homography that produces the largest set of inliers amongst the noisy correspondences. A least-squares homography can then be calculated using all of the inliers from that largest set of inliers and then used for warping one image onto the other. Using this process, we can effectively generate an image mosaic using two input images.

1. Introduction

Image mosaicing is a fundamental problem in computer vision that involves stitching multiple images together to form a larger, panoramic view. It is a powerful technique that has a wide range of applications across a variety of fields such as 3D modeling, surveillance, medical imaging, photography, and many more. The process of image mosaicing typically consists of several steps, including feature detection, feature matching between the images, image alignment using those corresponding features, and finally image warping and blending to create a seamless composite image. There are several options for algorithms and techniques to carry out these steps.

The methods we are implementing include a Harris corner detector for feature detection, compute normalized cross-correlation (NCC) for feature matching and use random sample consensus (RANSAC) for determining the homography for image alignment. The Harris corner detector is a popular algorithm for detecting corners in images. It is used to identify significant features in images that can be used for matching and registration.

NCC is a technique used to identify corresponding features in different images. The NCC scores correspond to how close of a match two features are on a scale of -1 to 1. NCC is a powerful method for feature matching. Homography estimation is used to estimate the geometric relationship between two images. It is a transformation matrix that maps one image onto another image, allowing the two to be combined. In order to robustly estimate the homography, RANSAC is used on the noisy correspondences to determine the homography that produces the largest set of inliers that should then be used to estimate the homography for warping.

The goal of this project is to implement an image mosaicing algorithm that uses a Harris corner detector to find features in the images, compute NCC to identify corresponding features, use RANSAC to eliminate outliers and estimate the homography matrix between the images, and then warp one image into the coordinate system of the other to create a mosaic.

2. Algorithms

2.1. Harris Corner Detector

Harris corner detector is a feature detection algorithm that is used to identify points of interest in an image. These key points are significant because they are the required inputs for various other tasks. In our case, they are used for feature matching. The Harris corner detector gives a mathematical approach for determining changes in intensity within small sections/windows as they are moved across an image. In particular, it looks for areas where the intensity changes in multiple directions, meaning they are likely to be corners or edges. The algorithm computes a corner response function for each pixel in the image using the image gradients to ultimately determine if a given pixel is a corner.

Algorithm 1 Harris Corner Detector

Input: Image (I).
Input: Empirically determined constant (k) in the range $0.04 \leq x \leq 0.06$. Defaults to 0.04.
Input: Window Size (N) of ($N \times N$) used for SOBEL mask and Gaussian Blur.

1: Compute the image gradients, I_x and I_y, with the SOBEL masks G_x^S and G_y^S in the x and y direction respectively:

$$I_x = G_x^S \times I \tag{1}$$
$$I_y = G_y^S \times I \tag{2}$$

2: Compute products of derivatives at each pixel:

$$I_x^2 = I_x \times I_x \tag{3}$$
$$I_y^2 = I_y \times I_y \tag{4}$$
$$I_x I_y = I_x \times I_y \tag{5}$$

3: Compute the sums of the products at each pixel using a window averaging:

$$S_x^2 = G_{S'} \times I_x^2 \tag{6}$$
$$S_y^2 = G_{S'} \times I_y^2 \tag{7}$$
$$S_{xy} = G_{S'} \times I_{xy} \tag{8}$$

▷ Here $G_{S'}$ is a Gaussian Mask

4: Define the Matrix at each pixel:

$$M = \begin{bmatrix} S_x^2 & S_{xy} \\ S_{xy} & S_y^2 \end{bmatrix} \tag{9}$$

5: Compute the Response (R):

$$R = \det(M) - k \cdot (\text{trace}(M))^2 \tag{10}$$

6: Threshold R.
7: Compute Nonmax suppression.

The Equation for Harris Corner Detector is:

$$E(u,v) = \sum_{x,y} \underbrace{w(x,y)}_{\text{Window Function}} \left[\underbrace{I(x+u,y+v)}_{\text{Change in Intensity}} - I(x,y) \right]^2 \tag{11}$$

$$E(u,v) \approx \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix} \tag{12}$$

Where,

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \tag{13}$$

2.2. Normalized Cross Correlation

Normalized Cross Correlation (NCC) is a technique used in image processing for measuring the similarity between two image patches. Higher NCC scores coincide with the best match and mean it is less likely that the match was identified incorrectly. The NCC values range from -1 to 1 where -1 means completely uncorrelated and 1 indicates a perfect match. NCC is a normalized version of cross correlation that mitigates an issue by comparing the same area in a scene with different illumination intensities. This is accomplished by normalizing the pixels in the patches before comparing them. NCC involves subtracting the mean from the pixels in the image patch, dividing each pixel by the standard deviation, and computing the cross-correlation as the sum of the products of the normalized image patches. Each patch is a unit norm vector and the NCC is their dot product. In this project, NCC is used during feature matching to identify corresponding points across the two images.

The Equation of Normalized Cross Correlation is given by:

$$N_{fg} = C_{\hat{f}\hat{g}} = \sum_{[i,j]\in\mathbb{R}} \hat{f}(i,j)\hat{g}(i,j) \tag{14}$$

Where,

$$\hat{f} = \frac{f}{\|f\|}; \quad \hat{g} = \frac{g}{\|g\|} \tag{15}$$

2.3. RANdom SAmple Consensus (RANSAC)

RANdom SAmple Consensus, or RANSAC, is a robust estimation algorithm that is used to find the best model that fits a set of noisy data. In this project, RANSAC is used to estimate the homography between two images based on a set of noisy correspondences. The algorithm, for lines, involves drawing a sample from the data, uniformly and at random, fitting to that sample, testing the distance from every other data point to that fitted line in order to determine if it is an inlier or an outlier, and repeating that process a certain number of times to find the best fit.

Parameters such as how many points to sample, the distance threshold, and the number of iterations to perform depend on the task and the data being used. The equations for determining the number of samples, N, are given below. It is dependent on the desired probability of finding a sample free from outliers. For example, a value of $p = 0.99$ for a certain N gives a 99% chance of getting a good sample of all inliers within the corresponding number of iterations. The number of iterations required is determined using Table 1 below based on the number of points per sample size, s, and the probability of a point being an outlier, e. For our purposes, we require a sample of $s = 4$ points and opted to perform 72 iterations with a distance threshold of 5 pixels. This means that for a given iteration, points with a distance from the fitted line less than 5 pixels are considered inliers and those with a distance greater than 5 pixels are outliers. After 72 iterations, the largest set of inliers is used to estimate the homography.

Equation (17) and Table 1 gives the number of samples (Iteration) to compute given other paramerters.

$$p = 1 - (1 - (1 - e)^s)^N \tag{16}$$

Rearranging to get the value of N:

$$N = \frac{\ln(1 - p)}{\ln(1 - (1 - e)^s)} \tag{17}$$

Here,

- e — Probability that a point is an outlier.
- N — Number of samples (We want to compute this).
- s — Number of points in a sample.
- p — Desired probability that we get a good sample.

	Proportion of outliers e						
s	5%	10%	20%	25%	30%	40%	50%
2	2	3	5	6	7	11	17
3	3	4	7	9	11	19	35
4	3	5	9	13	17	34	72
5	4	6	12	17	26	57	146
6	4	7	16	24	37	97	293
7	4	8	20	33	54	163	588
8	5	9	26	44	78	272	1177

Table 1. Choosing the number of iterations N given a sample size s.

Algorithm 2 RANdom SAmple Consensus (RANSAC)

Input:

n — the smallest number of points required.
k — the number of iterations required.
t — the threshold used to identify a point that fits well.
d — the number of nearby points required to assert a model fits well.

Algorithm:

repeat
    Draw a sample of n point from the data uniformly and at random.
    Fit to that set of n points.
    for all points outside the sample **do**
        if the distance from the point to the line is less than t **then**
            The point is close.
        else
            The point is far.
        end if
    end for
    if there are d or more points close to the line **then**
        There is a good fit.
        Refit the line using all these points.
    end if
until k iterations have occurred.
Use the best fit from this collection, using the fitting error as a criterion.

2.4. Homography Estimation

A homography is a mathematical transformation that describes the relationship between two coordinate systems. For the purposes of this project, we estimate a homography to map coordinates of points in one image to points in the other image. This is necessary for the final warping step to create a single composite image as a mosaic of the two images. The equations for homography estimation can be seen below.

Equations to find Homography Matrix (H):

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \sim \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{18}$$

In Equation form:

$$x' = \frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + h_{33}} \tag{19}$$

$$y' = \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + h_{33}} \tag{20}$$

Since we are using the algebraic distance method to solve the homography matrix the above equation holds true only when the condition $\|h\| = 1$ is met. So,

$$(h_{31}x + h_{32}y + h_{33}) \cdot x' = h_{11}x + h_{12}y + h_{13} \tag{21}$$

$$(h_{31}x + h_{32}y + h_{33}) \cdot y' = h_{21}x + h_{22}y + h_{23} \tag{22}$$

$$\implies h_{11}x + h_{12}y - h_{31}xx' - h_{32}yx' - h_{33}x' + h_{13} = 0 \tag{23}$$

$$\implies h_{21}x + h_{22}y - h_{31}xy' - h_{32}yy' - h_{33}y' + h_{23} = 0 \tag{24}$$

In Matrix form:

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x_1' & -y_1x_1' & -x_1' \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y_1' & -y_1y_1' & -y_1' \\ & & & & & \vdots & & & \end{bmatrix} \cdot \begin{bmatrix} h_{11} \\ h_{12} \\ \vdots \\ h_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{25}$$

Simply,

$$\underbrace{A}_{2N\times 9} \cdot \underbrace{h}_{9\times 1} = \underbrace{0}_{2N\times 1} \tag{26}$$

The above Homogenous matrix equation can be solved using Singular Value Decomposition (SVD).

2.5. Image Warping and Blending

Warping and blending are the final steps in producing a mosaic of multiple images. Prior to warping, the size of the output image must be determined so that it is big enough to contain the union of all pixels in the two images. The image that does not need to be warped is first placed into the output frame. The second is then warped into the output image using the estimated homography or potentially it's inverse. A blending scheme is used in order to combine colors in areas of overlapping pixels. There are several blending schemes, including straight averaging, feathering which considers the distance from the image border, and equalization for adjusting intensities.

Approaches to Blending:

1. Straight Averaging

$$P = \frac{(P_1 + P_2)}{2} \tag{27}$$

2. Feathering

$$P = \frac{(w_1 * P_1 + w_2 * P_2)}{(w_1 + w_2)} \tag{28}$$

Where w_i is the distance from the image border

3. Equalize intensity statistics (gain, offset)

3. Experiments

The workflow of the program goes as follows:

(a) Read in two image.

(b) Apply Harris corner detector to each image to identify corner features.

(c) Find correspondences between the two images by computing NCC to determine the best matches.

(d) Estimate the homography with the above correspondences using RANSAC for robust estimation.

(e) Warp one image onto the other, blending overlapping pixels to create a single composite image.

Following this fundamental outline, we implemented the techniques described in the Algorithms section above in order to determine the values of parameters and develop our image mosaicing program.

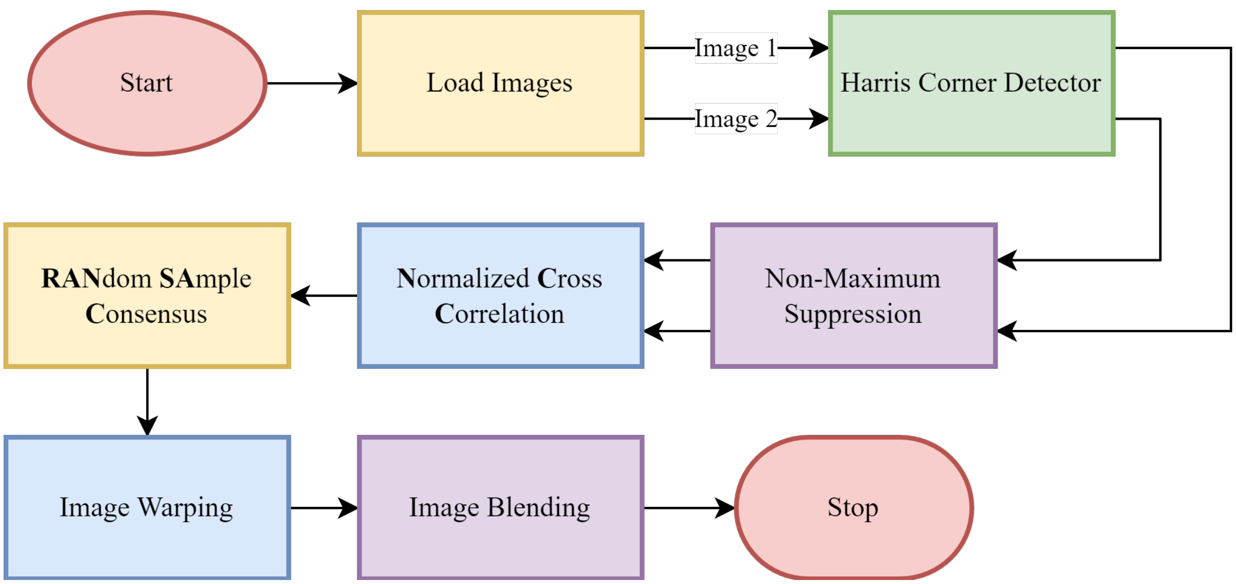


Figure 1. Image Mosaicing Flowchart.

3.1. Hallway Dataset

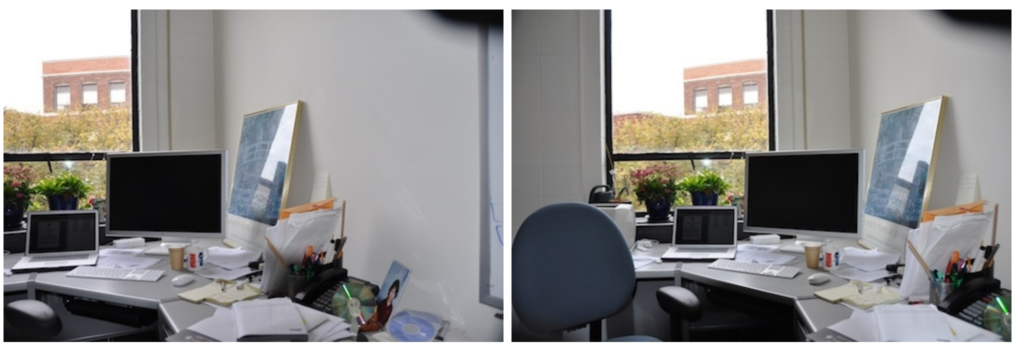
The hallway dataset contains images of a hallway scene from 3 distinct angles. They contain little visual clutter and contain a few large distinguishable features, such as bulletin board, doors, and a water fountain. Additionally, the illumination intensity appears relatively uniform throughout the scene.

(a) (b)

Figure 2. Hallway dataset - Input Images

3.2. Office Dataset

The office dataset contains images of an office scene from 10 distinct angles, effectively providing a 360° panoramic view of the room across the set of images. This scene is more complex than the hallway scene in the sense that there is a lot more going on, visually. There are numerous small objects scattered across a cluttered desk and some reflective surfaces such as the glass on a picture frame and a DVD. Moreover, there is a window as a main source of light, causing the illumination intensity to differ throughout the scene in places like underneath the desk.



(a) (b)

Figure 3. Office dataset - Input Images

4. Results and Discussion

The images displayed in the following results represent an image pair from both datasets, the hallway, and the office. These select outputs aim to best demonstrate our overall results since we cannot include them all for the sake of brevity.

4.1. Feature Detection

Following the program outline described in the Experiments section above, the first step after loading the image pair being used to create a mosaic is to perform feature detection. We apply a Harris corner detector over each image and then do non-maximum suppression to obtain a sparse set of corner features in both images. A Harris corner detector is able to identify corner features by sliding a small window across the image and evaluating changes in intensity.

In our implementation of the Harris corner detector algorithm, we used a 3×3 window size to apply both the Sobel and Gaussian masks following the procedure in Algorithm 1. The response, R, was computed using an empirically determined constant k with a value of 0.04, which falls within the range of $k \in [0.04, 0.06]$. To reduce the number of detected corner points, we employed the use of Non-Maximum Suppression (NMS) on the computed R values. Subsequently, we applied a dataset-specific threshold to the suppressed values in order to remove any remaining noise in the response matrix. The threshold values were determined through trial and error to be 0.0025 and 0.01 for the hallway and office dataset respectively.

As depicted in Figures 4 and 5, we were able to accurately detect corner features in the images. That being said, there were some inaccuracies in our results in the form of false positives and false negatives. For instance, in Figures 4a and 4b a false positive was detected near the top right of the wooden door due to the reflection of the overhead lighting. Likewise, in Figures 5a and 5b, a corner was detected on the glass of the framed blue image due to the reflection of light from the window. Additionally, there were false negatives in both datasets. For example, the top left corner of the bulletin board went undetected in Figure 4a, and the top and bottom left corners of the framed blue picture were missed in Figure 5a.

(a) **(b)**

Figure 4. Hallway dataset - Feature Detection

(a) **(b)**

Figure 5. Office dataset - Feature Detection

4.2. Feature Matching

Using the outputs from the feature detection step, feature matching is performed in order to find the correspondences between the two images. Given a set of two corners from the two images, NCC is computed using image patches centered around each corner. Every corner detected in one image is compared to every corner detected in the other image in order to determine the best possible match, making this an $O(n^2)$ process. We also set a threshold such that the best possible matches have a high NCC score in an effort to eliminate incorrect matches. In our implementation of NCC, we opted for a patch size of 5 pixels.

The results produced by our feature-matching algorithm proved to be sufficient for our purposes. Our NCC function identified enough correspondences to be able to proceed with RANSAC for homography estimation, but not an abundance of extraneous matches that would be considered outliers as seen in Figures 6 and 7. The lines in the figures are drawn between corresponding points in the scene across the two images. High NCC scores indicate correct matches. Corresponding corners that had the best NCC score comparatively are included in the output correspondences to be used in future steps. High NCC values also provides a level of confidence in the quality of the matches being identified. Moreover, our correspondence visualization closely resembles that of the example provided in the project description.

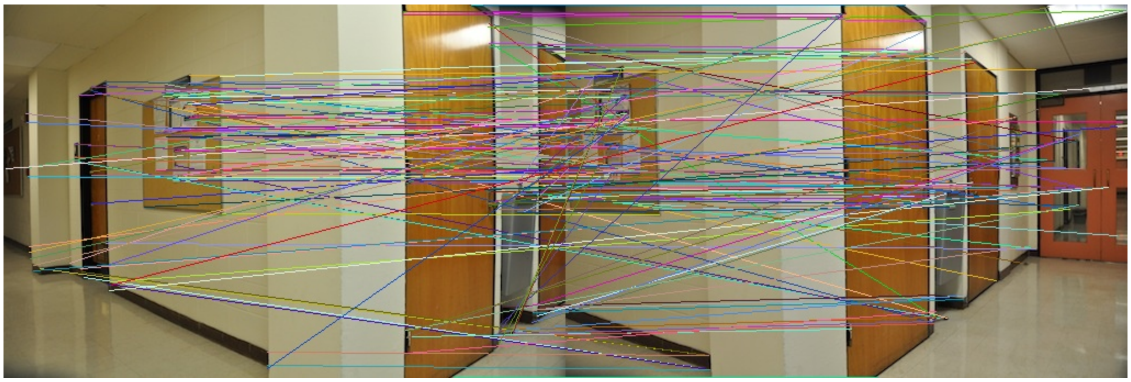


Figure 6. Hallway dataset - Feature Matching

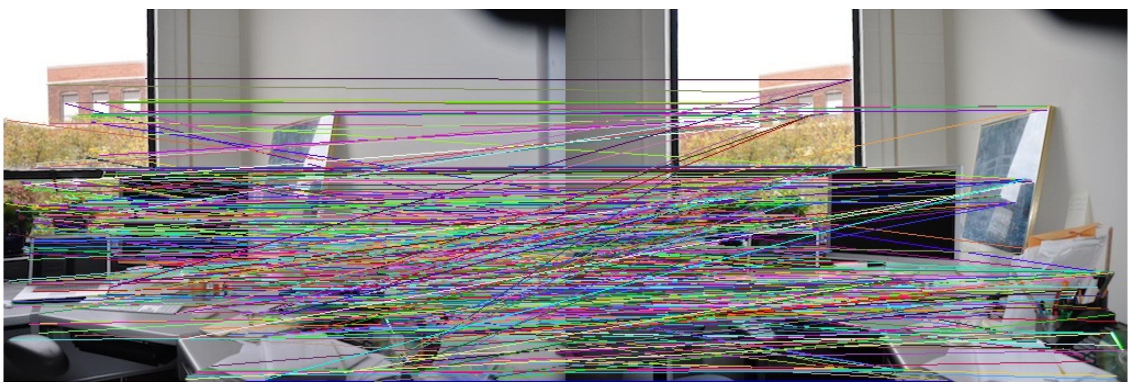


Figure 7. Office dataset - Feature Matching

4.3. Homography Estimation

With the correspondences identified from the feature matching process, we are now able to estimate the homography required to warp one image into the coordinate system of the second. Given that the correspondences we found are likely to have many errors/outliers which is exhibited in figures 5 and 6 by the various slopes of the lines, we perform RANSAC to robustly estimate the homography from the noisy correspondences. For an overview of the process, we use a random sample of 4 correspondences to estimate a homography and then determine the number of inliers produced after mapping all points using that homography. This iterative process is completed a certain number of times before computing a least-squares homography using all of the inliers from the largest set of inliers that was found. The RANSAC algorithm and parameters are described in greater detail in the algorithms section above in Algorithm 2.

In our implementation of RANSAC, we used a sample size of $N = 4$ points, a distance threshold of 5 pixels, and ran it over 72 iterations as per the highlighted row in Table 1 above. We were able to achieve positive results using these parameters as shown by visualizations in Figures 8 and 9. The lines in the figures are drawn between corresponding features in the scene across the two images. When compared to Figures 6 and 7 which display all of the correspondences that were found including errors and outliers, it is clear that the set of correspondences that our RANSAC determined to be inliers more accurately reflect corresponding features in the two images. This can be seen either through individual visual inspection of the lines, or more generally, the similar slopes of all the correspondence lines between the two images. This makes sense because the two images capture the same scene with a significant amount of overlapping features but taken at slightly different angles. Therefore, lines drawn between correctly matched features in the two images should all have the same slope.

Once RANSAC is complete, we have the largest set of inliers found across all of the iterations. Those inliers are used to compute a final homography that enables us to warp one image into the coordinate system of the other image. The final homography matrices found for each dataset can be seen below in Equations (29) and (30) These are the homographies used to warp one image onto the other in the final output frame for the corresponding dataset.

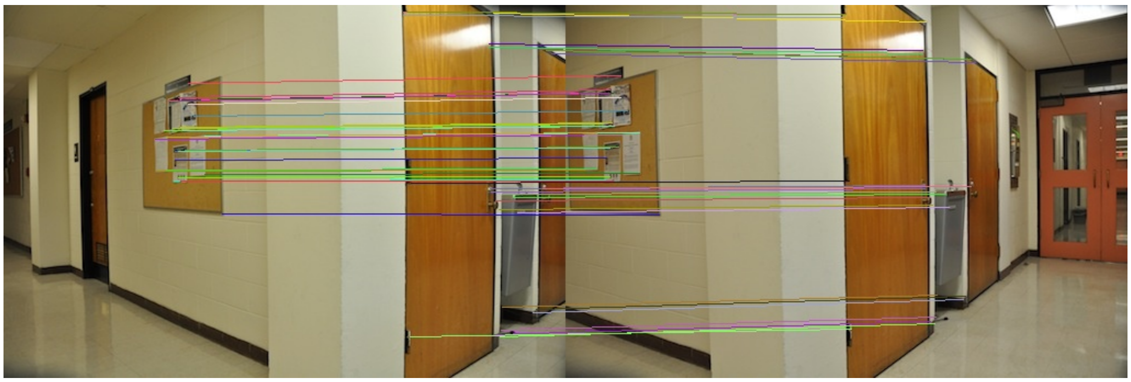


Figure 8. Hallway dataset - RANSAC Inliers

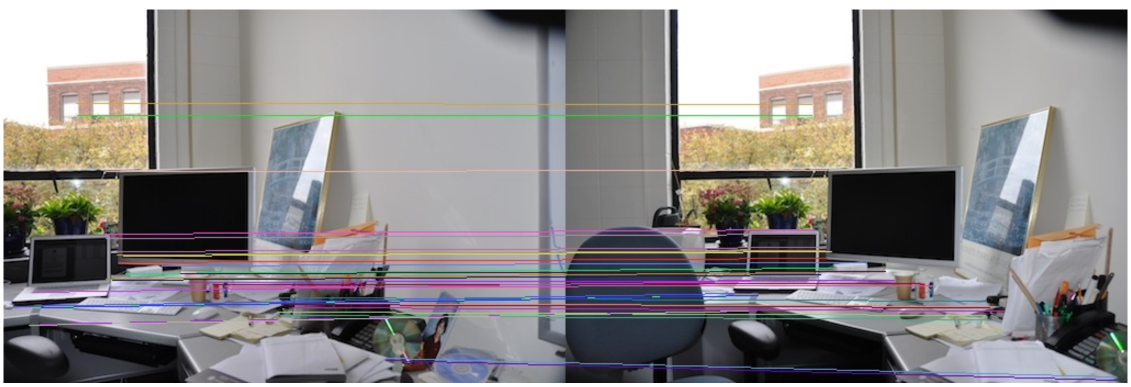


Figure 9. Office dataset - RANSAC Inliers

$$H_{\text{Hallway}} = \begin{bmatrix} 1.16410639 \times 10^{0} & 7.24666331 \times 10^{-2} & -2.59175335 \times 10^{1} \\ 1.30681317 \times 10^{-2} & 1.25005557 \times 10^{-2} & -1.57003065 \times 10^{2} \\ 2.45925764 \times 10^{5} & 4.84730417 \times 10^{-4} & 1.00000000 \times 10^{0} \end{bmatrix} \tag{29}$$

$$H_{\text{Office}} = \begin{bmatrix} 1.09010727 \times 10^{0} & -9.97310911 \times 10^{-2} & 5.51610087 \times 10^{0} \\ 1.88851954 \times 10^{-1} & 7.99619183 \times 10^{-1} & 1.35065914 \times 10^{2} \\ 9.03209811 \times 10^{-4} & -9.85230262 \times 10^{-4} & 1.00000000 \times 10^{0} \end{bmatrix} \tag{30}$$

4.4. Warping and Blending

Using the homography matrix estimated previously with RANSAC, we are now able to map points from the coordinate system of one image to the coordinate system of another. This is required for the final step of warping one image onto the other and blending the overlapping pixels together to create a single image that shows the union of all pixels from both images.

In order to accomplish this, we needed to create a final output image that was large enough to contain the composition of the two images. This is primarily a concern for the width of the output image given the nature of the input images being taken at similar vertical angles. With that in mind, we are effectively creating a panorama, so we determined a sufficient width of the final image to be the sum of the widths of the two input images. We proceeded to insert the image not being warped into the output frame and then warped in the second image. As shown in Section 4.4, we were then able to successfully produce a mosaic of the two inputs images for each dataset.

Figure 10. Hallway dataset - Output Mosaic

Figure 11. Office dataset - Output Mosaic

4.5. Extra Credit

The extra credit portion of this project involves warping one image into a user-selected region in a second image. To accomplish this, we consider the corner points of the image being embedded to be the corners of the image. To find the corners of the region where the image is being embedded, we take user input in the form of mouse clicks on the target image in clockwise order starting with the top left corner. Using the corner coordinates from both images, we can calculate a homography matrix to map points from one image coordinate system to the other. The computed homography is then used to warp the image we want to embed into the base image, creating a single composite image.

We managed to accomplish this task as shown below in Figure 12 where we successfully embedded an image onto the bulletin board in the hallway scene. We selected this region of the image by using mouse clicks to pick the corners of our desired frame in the order that was previously mentioned above. The locations of the mouse clicks are represented by the red squares centered around the click coordinates. The order of the corner selection is significant here because it influences the orientation of the image by defining the coordinate system of the region we are embedding the image into. For example, if the corners of the target region were selected clockwise starting from the bottom right corner instead, the resulting embedded image would be upside down.

Figure 12. Extra Credit - Output Example

5. Conclusion

In conclusion, the project "Image Mosaicing" demonstrates a practical application of image processing techniques to create a mosaic using two images of the same scene. The project defines a general procedure of feature detection, feature matching, homography estimation, and warping/blending. Implementing and performing this process can be used to find corner features in the images to then identify correspondences that are crucial for image alignment to ultimately produce a single composite image depicting the scene in the images.

The performance of the image mosaicing algorithm is dependent on several factors and can influence the decision as to what techniques to use for the various steps. For example, in scenes that vary in illumination intensity, NCC is a good choice for feature matching because normalization mitigates the impact of the different illumination intensities on the correlation results. Furthermore, the choice of parameters for each technique also plays a significant role. The project provides resources that describe how to calculate and fine-tune the parameters for some of these techniques, such as RANSAC for homography estimation.

Overall, this project demonstrates how fundamental image processing techniques can be powerful tools for image mosaicing. By applying these techniques to a set of images depicting a single scene, it is possible to extract useful information about corresponding points in the images and create a seamless composite image.

Appendix - Code

Checkout our source code in GitHub (https://github.com/Sahas-Ananth/CVP2)

A. Source Code

```python
import os
from random import choices, randint

import cv2
import numpy as np


class Panorama:
    def __init__(self, path: str) -> None:
        """Initialize the Panorama class.

        Args:
            path (str): Path to the folder containing the images.
        """
        if path == "DanaHallWay1":
            self.harris_thresh = 0.0025
        elif path == "DanaOffice":
            self.harris_thresh = 0.01

    def load_images(self, path: str) -> np.ndarray:
        """Load images from a path.

        Args:
            path (string): Path to the folder containing the images.

        Returns:
            np.ndarray: Color images.
            np.ndarray: Grayscale images.
        """
        files = [
            os.path.join(path, f)
            for f in sorted(os.listdir(path))
            # if f.endswith(".JPG")
        ]
        return np.array([cv2.imread(f) for f in files]), np.array(
            [cv2.imread(f, 0) for f in files]
        )

    def non_maximum_suppresion(
        self, image: np.ndarray, window_size: int = 5
    ) -> np.ndarray:
        """Apply non-maximum suppression to a given image

        Args:
            image (np.ndarray): Image to perform non-max suppression on the.
            window_size (int, optional): The window size around each pixel.
        Returns:
            suppressed (np.ndarray): Resulting image after non-maximum suppression.
        """
        suppressed = image.copy()
        global_min = image.min()
        p = window_size // 2
```

```python

        width, height = suppressed.shape

        for i in range(width):
            x1 = max(0, i - p)
            x2 = min(width, i + p)
            for j in range(height):
                # Bounds for window around pixel at (i, j)
                y1 = max(0, j - p)
                y2 = min(height, j + p)

                # Set pixel value to the global min to exclude it from max
                value = suppressed[i, j]
                suppressed[i, j] = global_min

                # If pixel has the maximum value in the window then use its value
                local_max = suppressed[x1:x2, y1:y2].max()
                if value > local_max:
                    suppressed[i, j] = value

                # Else keep it is global minimum
        return suppressed

    def harris_corner_detector(
        self, image: np.ndarray, k: float = 0.04, window_size: int = 3
    ) -> np.ndarray:
        """Given an image, it finds the corners using the Harris Corner Detector.

        Args:
            image (np.ndarray): Grayscale image to find the corners.
            k (float, optional): Empirically determined constant in range 0.04 <= x <= 0.06. Defaults to 0.04.
            window_size (int, optional): The search window size for operations such as Gaussian Blur, Sobel Mask etc.. Defaults to 3.
        Returns:
            corners (list): Corners in the image in the form [(x, y)].
        """
        # Gaussian Blur over the image
        image = cv2.GaussianBlur(image, (window_size, window_size), 0)

        # Compute the gradients
        Ix = cv2.Sobel(image, cv2.CV_64F, dx=0, dy=1, ksize=window_size)
        Iy = cv2.Sobel(image, cv2.CV_64F, dx=1, dy=0, ksize=window_size)

        # Compute the products of the gradients
        IxIy = Ix * Iy
        Ix2 = Ix**2
        Iy2 = Iy**2

        # Compute the sums of the products of the gradients
        S2x = cv2.GaussianBlur(Ix2, (window_size, window_size), 0)
        S2y = cv2.GaussianBlur(Iy2, (window_size, window_size), 0)
        Sxy = cv2.GaussianBlur(IxIy, (window_size, window_size), 0)

        # Harris Corner Response
        det = (S2x * S2y) - (Sxy**2)
        trace = S2x + S2y

```

```
        R = det - k * (trace**2)

        # Normalize
        R /= R.max()

        # Non-max suppression
        R = self.non_maximum_suppresion(R)

        # Thresholding
        R[R > self.harris_thresh] = 255

        corners = np.where(R == 255)
        corners = list(zip(corners[0], corners[1]))
        return corners

    def normalized_cross_correlation(
        self,
        image1: np.ndarray,
        image2: np.ndarray,
        corners1: list,
        corners2: list,
        window_size: int = 5,
    ) -> dict:
        """Given two images and their corners, it computes the normalized cross correlation
        ↪ between the two images and returns the correspondences.

        Args:
            image1 (np.ndarray): Color image.
            image2 (np.ndarray): Color image.
            corners1 (list): It is a list of tuples (x, y) where x and y are the coordinates
↪   of the corners in image1.
            corners2 (list): It is a list of tuples (x, y) where x and y are the coordinates
↪   of the corners in image2.
            window_size (int, optional): Window Size. Defaults to 5.

        Returns:
            Correspondences (dict): Correspondences between the two images in the form
↪   {corner1 (x, y): corner2 (x, y)}
        """
        pad = window_size // 2

        # Pad gray images
        image1_pad = np.pad(image1, pad, "constant", constant_values=0)
        image2_pad = np.pad(image2, pad, "constant", constant_values=0)

        correspondences = {}
        for i, corner1 in enumerate(corners1):
            # Image patch around corner 1
            x1 = max(corner1[0] - pad, 0)
            x2 = max(corner1[0] + pad + 1, window_size)
            y1 = max(corner1[1] - pad, 0)
            y2 = max(corner1[1] + pad + 1, window_size)
            patch1 = image1_pad[
                x1:x2,
                y1:y2,
            ]

            max_ncc = -1
```

```python
            best_corner = None
            for j, corner2 in enumerate(corners2):
                print(f"i={i + 1}/{len(corners1)} j={j + 1}/{len(corners2)}", end="\r")

                # Image patch around corner 2
                x1 = max(corner2[0] - pad, 0)
                x2 = max(corner2[0] + pad + 1, window_size)
                y1 = max(corner2[1] - pad, 0)
                y2 = max(corner2[1] + pad + 1, window_size)
                patch2 = image2_pad[
                    x1:x2,
                    y1:y2,
                ]
                # Calculate NCC using image patches
                patch1_hat = patch1 / np.linalg.norm(patch1)
                patch2_hat = patch2 / np.linalg.norm(patch2)
                ncc = np.sum(patch1_hat * patch2_hat)

                # If this NCC is the new max, store it and the coords of the corner
                if ncc > max_ncc:
                    max_ncc = ncc
                    best_corner = corner2

                # Store correspondence with highest NCC
                correspondences[corner1] = best_corner
        return correspondences

    def homography(self, points1: np.ndarray, points2: np.ndarray) -> np.ndarray:
        """Given two sets of points, it computes the homography matrix.

        Args:
            points1 (np.ndarray): Points in the form [[x1, y1], [x2, y2], ...] of shape (4 x 2).
            points2 (np.ndarray): Points in the form [[x1, y1], [x2, y2], ...] of shape (4 x 2).

        Returns:
            h_mat (np.ndarray): Homography matrix of shape (3 x 3).
        """
        H = np.zeros((points1.shape[0] * 2, 9))
        for i, ((x1, y1), (x2, y2)) in enumerate(zip(points1, points2)):
            # print(f"{i}, (({x1}, {y1}), ({x2}, {y2}))")
            H[2 * i] = [x1, y1, 1, 0, 0, 0, -x2 * x1, -x2 * y1, -x2]
            H[2 * i + 1] = [0, 0, 0, x1, y1, 1, -y2 * x1, -y2 * y1, -y2]
        _, _, V = np.linalg.svd(H.T @ H)
        h = V[-1]
        h_mat = h.reshape(3, 3)
        h_mat = h_mat / h_mat[2, 2]
        return h_mat

    def ransac(
        self, correspondences: dict, threshold: int = 5, k: int = 72, N: int = 4
    ) -> np.ndarray:
        """Performs RANSAC to find the best homography matrix, given the correspondences. This is done using the largest set of inliers.

        Args:
```

```
            correspondences (dict): Correspondences between the two images in the form {corner1 (x, y): corner2 (x, y)}
            threshold (int, optional): The distance threshold. Defaults to 5.
            k (int, optional): No of iterations. Defaults to 72.
            N (int, optional): Sample size. Defaults to 4.

        Returns:
            H (np.ndarray): Homography matrix.
            inliers (dict): Inliers in the form {corner1 (x, y): corner2 (x, y)}.
            outliers (dict): Outliers in the form {corner1 (x, y): corner2 (x, y)}.
        """
        H = None
        inliers = {}
        outliers = {}
        max_inliers = 0

        for _ in range(k):
            set_inliers = {}
            set_outliers = {}

            num_inliers = 0
            num_outliers = 0

            # Sample 4 points from the correspondences
            points1 = choices(list(correspondences.keys()), k=4)
            points2 = [tuple(correspondences.get(p)) for p in points1]

            # Compute homography matrix using these points
            h = self.homography(np.asarray(points1), np.asarray(points2))

            for corner1, corner2 in correspondences.items():
                # Estimate point using homography
                pt1 = np.array([corner1[0], corner1[1], 1])
                pt2 = np.array([corner2[0], corner2[1], 1])
                res = np.dot(h, pt1)
                # res = np.matmul(h, pt1)
                res = (res[:2] / res[2]).astype(int)
                dist = np.linalg.norm(res - corner2)

                # Check if outlier
                if dist > threshold:
                    set_outliers[tuple(corner1)] = tuple(corner2)
                    num_outliers += 1
                    continue

                # Store inlier
                set_inliers[tuple(corner1)] = tuple(corner2)
                num_inliers += 1

            # Check if this homography produced the new largest set of inliers
            if num_inliers > max_inliers:
                max_inliers = num_inliers
                inliers = set_inliers
                outliers = set_outliers
                H = h

        return H, inliers, outliers

```

```
    def create_panorama(
        self, image1: np.ndarray, image2: np.ndarray, H: np.ndarray
    ) -> np.ndarray:
        """Creates a panorama image given two images and the homography matrix.

        Args:
            image1 (np.ndarray): Color image.
            image2 (np.ndarray): Color image.
            H (np.ndarray): Homography matrix, H (3x3).

        Returns:
            Final (np.ndarray): Final panorama image.
        """
        copy1, copy2 = image1.copy(), image2.copy()
        stitcher = cv2.Stitcher().create()
        final_size = (copy1.shape[1] + copy2.shape[1], copy2.shape[0])
        final = cv2.warpPerspective(copy2, H, final_size)
        _, final = stitcher.stitch((copy1, copy2))
        return final

    def embed_image(
        self, embed_image: np.ndarray, base_image: np.ndarray
    ) -> np.ndarray:
        """Embeds an image into a given base image.

        Args:
            embed_image (np.ndarray): The image being embedded in the base image.
            base_image (np.ndarray): The base image being embedded into.
        Returns:
            composite_image (np.ndarray): The resulting composite image containing the embed.
        """
        # Create a copy of the base image
        base_image = base_image.copy()

        # Get the desired region to embed image on reom mouse events.
        region = self.get_user_region(base_image)
        region = np.asarray(region)

        base_width, base_height, _ = base_image.shape
        embed_width, embed_height, _ = embed_image.shape

        # Clockwise starting from top left corner
        embed_image_corners = np.array(
            [(0, 0), (embed_width, 0), (embed_width, embed_height), (0, embed_height)]
        )

        # Calculate homography matrix
        H = self.homography(embed_image_corners, region)

        # Warp the image into the user selected region using the homography matrix
        embed_image_isolated = cv2.warpPerspective(
            embed_image, H, (base_height, base_width)
        )

        # Create a mask of the user selected region
        fill_mask = np.zeros(base_image.shape).astype("uint8")
        cv2.fillConvexPoly(fill_mask, region, (255, 255, 255))

```

```python
        # Apply mask of user selected region to the base image
        base_image_with_region = cv2.bitwise_and(base_image, cv2.bitwise_not(fill_mask))

        # Add the base image and the isolated embed image
        composite_image = base_image_with_region + embed_image_isolated

        return composite_image

    def mouse_callback(
        self, event: int, x: int, y: int, flags: int, param: dict
    ) -> None:
        """Callback function for mouse events used when obtaining the user selected region for embedding an image.

        Args:
            event (int): The type of mouse click event.
            x (int): The x pixel coordinate of the mouse in base image.
            y (int): The y pixel coordinate of the mouse in base image.
            flags (int): Event flags.
            param (dict): Any parameters that are passed in.
        """

        # Handle when the mouse is clicked
        if event == cv2.EVENT_LBUTTONDOWN:
            # Get image and corners by reference
            base_image = param["image"]
            corners = param["corners"]
            corners.append([x, y])
            # Draw square centered around click
            p = 5
            start_point = (x - p, y - p)  # Top left corner
            end_point = (x + p, y + p)  # Bottom right corner
            color = (0, 0, 255)
            thickness = 2
            cv2.rectangle(base_image, start_point, end_point, color, thickness)

    def get_user_region(self, base_image: np.ndarray) -> list:
        """Get the corners of the region to embed the image into from the user using mouse click on the base image.

        Args:
            base_image (np.ndarray): The base image being embedded onto.
        Returns:
            corners (list): the pixel coordinates of the corners of the user selected region in the form [(x, y)]
        """
        corners = []
        param = {
            "image": base_image,
            "corners": corners,
        }

        window_name = "Embed Image"
        cv2.imshow(window_name, base_image)
        cv2.setMouseCallback(window_name, self.mouse_callback, param)

        while True:
            cv2.imshow(window_name, base_image)
```

```python
            if cv2.waitKey(1) & 0xFF == ord("q") or len(corners) == 4:
                cv2.destroyWindow(window_name)
                break

        return corners

    def draw_lines(
        self, image1: np.ndarray, image2: np.ndarray, points: dict, col1: tuple = None
    ) -> None:
        """Draws lines between the points in the two images.

        Args:
            image1 (np.ndarray): Color image.
            image2 (np.ndarray): Color image.
            points (dict): dict of points in the form {corner1 (x, y): corner2 (x, y)}.
        Return:
            vis (np.ndarray): Color image with lines.
        """
        # Concatenate the two images
        vis = np.concatenate((image1, image2), axis=1)

        # Draw lines between correlated corners
        for pt1, pt2 in points.items():
            col = col1 if col1 else (randint(0, 255), randint(0, 255), randint(0, 255))
            start_y, start_x = pt1
            end_y, end_x = pt2
            end_x = int(end_x + vis.shape[1] / 2)
            end = (end_x, end_y)
            start = (start_x, start_y)
            cv2.line(vis, start, end, col, 1)

        return vis

    def draw_corners(self, image: np.ndarray, corners: list) -> None:
        """Draw corners in the image.

        Args:
            image (np.ndarray): Color image.
            corners (list): List of points in the form [(x, y)].
        Returns:
            vis (np.ndarray): Color image with corners drawn.
        """
        vis = image.copy()
        # Draw lines between correlated corners
        for corner in corners:
            cv2.circle(vis, (corner[1], corner[0]), 3, (0, 0, 255), 1)
        return vis

    def show_image(self, images: list, titles: list) -> None:
        """Shows a list of images along with their titles.

        Args:
            images (list): A list of all the images to be shown.
            titles (list): A list of all the titles of the images.
        """
        for image, title in zip(images, titles):
            cv2.imshow(title, image)
        if cv2.waitKey(0) & 0xFF == ord("q"):
```

```python
            cv2.destroyAllWindows()

    def save_image(self, images: list, fnames: list) -> None:
        """Saves a list of images with their corresponding filename

        Args:
            images (list): A list of all images to be saved
            fnames (list): A list of all the file names of the images
        """
        for image, fname in zip(images, fnames):
            cv2.imwrite(f"results/{fname}.jpg", image)


def main():
    """Main function to run the Panorama class."""
    # The directory to use two images from to create a mosaic
    DIR = "DanaHallWay1"
    # DIR = "DanaOffice"
    pano = Panorama(DIR)

    # Load images
    col, gray = pano.load_images(DIR)
    image1, image2 = col[0], col[1]
    gray1, gray2 = gray[0], gray[1]

    # Apply Harris corner detector
    corners1 = pano.harris_corner_detector(gray1)
    corners2 = pano.harris_corner_detector(gray2)

    # Find correspondences using NCC
    correspondences = pano.normalized_cross_correlation(
        gray1, gray2, corners1, corners2
    )

    # Use RANSAC to estimate homography matrix and find inliers
    H, inliers, outliers = pano.ransac(correspondences)

    # Warp images
    output = pano.create_panorama(image1, image2, H)

    # Create visualiations of results
    corners1_vis = pano.draw_corners(image1, corners1)
    corners2_vis = pano.draw_corners(image2, corners2)
    correspondences_vis = pano.draw_lines(image1, image2, correspondences)
    inliers_vis = pano.draw_lines(image1, image2, inliers)
    outliers_vis = pano.draw_lines(image1, image2, outliers)

    # Display results
    print(H)
    pano.show_image(
        [
            image1,
            image2,
            corners1_vis,
            corners2_vis,
            correspondences_vis,
            inliers_vis,
            outliers_vis,
```

```python
                output,
            ],
            [
                "Input 1",
                "Input 2",
                "corners1",
                "corners2",
                "correspondences",
                "inliers",
                "outliers",
                "Output",
            ],
        )

    # Save results
    pano.save_image(
        [
            image1,
            image2,
            corners1_vis,
            corners2_vis,
            correspondences_vis,
            inliers_vis,
            outliers_vis,
            output,
        ],
        [
            f"{DIR}_input1",
            f"{DIR}_input2",
            f"{DIR}_corners1",
            f"{DIR}_corners2",
            f"{DIR}_correspondences",
            f"{DIR}_inliers",
            f"{DIR}_outliers",
            f"{DIR}_output",
        ],
    )


def extra_credit():
    """Extra credit function that embeds an image into another image."""
    # The directory to pull an image from to use as the base image
    DIR = "DanaHallWay1"
    # DIR = "DanaOffice"

    pano = Panorama(DIR)

    # Load images
    col, _ = pano.load_images(DIR)
    embed_images, _ = pano.load_images("ec")

    embed_image = embed_images[0]
    # embed_image2 = embed_images[1]
    base_image = col[0]

    # Warp an image into a region in the second image
    output = pano.embed_image(embed_image, base_image)

```

```python
    # Display results
    pano.show_image([output], ["ExtraCredit_output"])

    # Save Results
    pano.save_image([output], ["ExtraCredit_output"])


if __name__ == "__main__":
    main()
    extra_credit()
```