Image Mosaicing

EECE5639 Computer Vision — Project 2
Fall Semester 2021

Cheung, Spenser & Liming, Tim
Dept. of Electrical and Computer Engineering
Northeastern University
Boston, MA, USA

***Abstract* — In this project we explored using a Harris corner detector to find corners in two images. We used those detected corners to find corresponding features in both images. By estimating a homography between the two images, we can warp one image into the second image's coordinate system to produce a mosaic that contains the union of all pixels in the two images.**

I. Introduction

To mosaic two images together, we can use corners to help. Using corners found by computing the Harris R function, we can perform non-max suppression to find a set of corner features in both images. With these sets, we can calculate normalized cross correlation values (NCC) and determine potential correspondences by choosing the highest NCC value. These correspondences can be used to estimate a homography to warp one image onto the other and create a mosaic.

In this project we will explore and compare detected features in two given images after we compute a Harris corner detector R function. Once correspondences are identified, we will use a calculated homography to create a mosaic of the two given images.

Code for our project can be found in our Github repository[1][2], linked in the Appendix.

II. Description of Algorithms

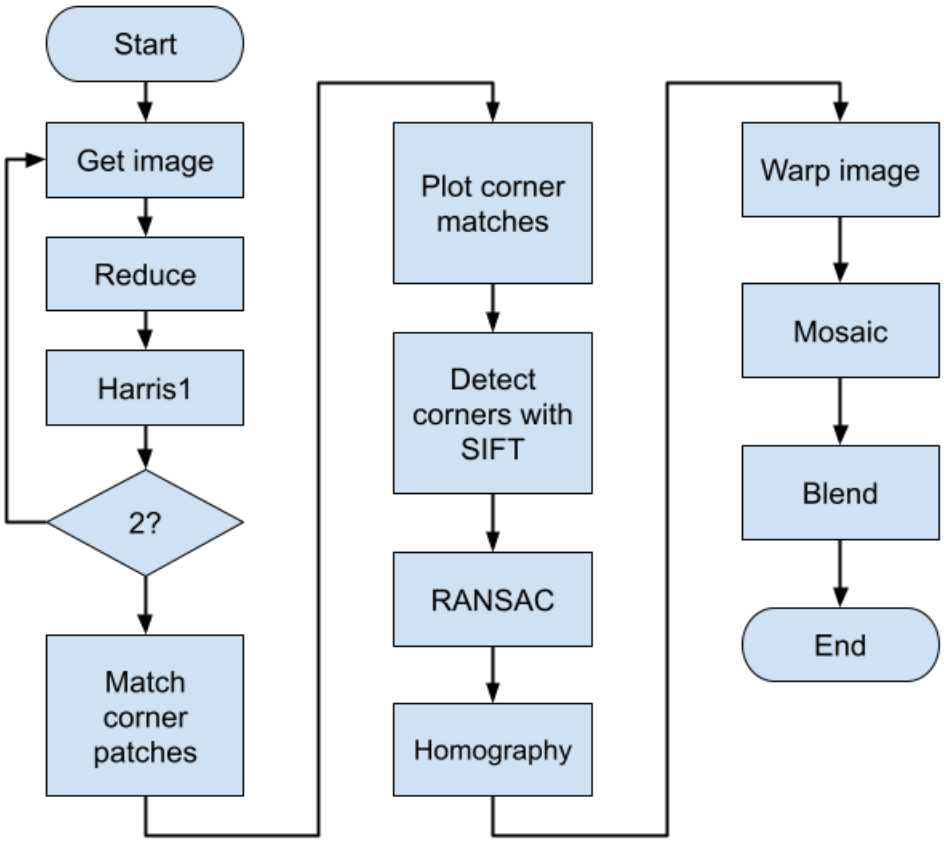
The first task of the project involved reading in a pair of images that are taken from the same location but have different perspectives and making them grayscale. We made use of the OpenCV python package `cv2` and used its `imread` function to read in an image and then used the `cvtColor` function to transform it to a grayscale image. Reading all the images in the folder and storing them in an array ahead of time allowed us to speed up the processing part of the program.

Once we had our images in an easy to use format, we used the `resize` function to reduce their size. The scaling factor used in this application was `0.5` for both dimensions.

After scaling down our images, we applied the Harris corner detector algorithm that we developed. The first step of our algorithm was to compute the image gradients in the x and y-directions using Sobel filters. The `cv2` package provides a helpful `Sobel` function that makes this easy.

We then used the image gradients to solve the matrix's determinant and trace values, and compute the Harris R function [3]:

$$det_M = (S^2X * S^2Y) - (S_{xy} * S_{xy}) \quad (1)$$



$$trace_M = S^2X + S^2Y \quad (2)$$

$$R = det - k * (trace)^2 \quad (3)$$

After getting the R values for the image, we threshold it with a threshold value equal to 1% of the maximum R value. We then create a window that parses the R matrix and suppresses any non-maximum R values. By doing this suppression, we are left with local maximums that can be used as corner features for comparison with another image.

Using the corner features of both images, we create patches and center them around the features. We then then calculate the normalized cross correlation between the patches:

$$NCC = \frac{1}{n}\sum_{x,y} \frac{1}{\sigma_f \sigma_t} f(x,y)t(x,y) \quad (4)$$

$$correspondences = max_{NCC}(features) \quad (5)$$

With equation 5, we are able to choose potential corner matches which we will label as correspondences. Since these correspondences are likely to contain noise from outliers, we run a SIFT detector and RANSAC to robustly get our homography estimate. The `cv2` function `findHomography` does this estimation well,

and allows us to easily get the homography matrix .

Using the homography matrix, we call the `cv2` function `warpPerspective` to apply a perspective transformation to our first image and warp it into the second image's coordinate system. The output from this is a mosaic.

However, the mosaic needs to have its pixels blended at the site where both images overlap. Our approach for blending was to use alpha blending. We created a Gaussian blurred version of the mosaic using the `cv2` function `GaussianBlur` with a kernel size of 3x3 and an std of `0`. Using this blurred image, we used the `cv2` function `addWeighted` with an alpha of `0.5` and a beta of `1-alpha`.

III. Experiments

Using the Harris algorithm that we developed as described in the previous section (`harris1`), we proceeded to compare the results of our algorithm with the results of a combination of built in functions that are part of the `cv2` python package.

This new function, called `harris2`, utilized the `cornerHarris` function to locate corners in the source image. We then dilated the resulting image to expand the areas for marking the corners. Using the same thresholding values as we used in our `harris1` function, we used the built-in threshold function to find our optimal values.

At this point, our `harris1` and `harris2` functions are essentially identical; what separates them is the accuracy of the different non max suppression algorithms. Instead of finding local maxima like in `harris1`, we first looked for centroids in our thresholded image using the `connectedComponentsWithStats` function.

With our centroids, we could further refine our corners with the `cornerSubPix` function. What this function does is iterate through the neighborhood of detected corners and refine to subpixel accuracy the location of the detected corner. Once completed, we can return our refined corners and the combination image with our detected corners.

Another experiment between the two different Harris functions involved varying the Harris Detector's k parameter. This parameter is an empirically determined constant in the range `[0.04, 0.06]` [4]. The k parameter determines the amount of corners detected. A larger k value reduces noise but can also miss some real corners (higher precision, but possibly less accuracy). A smaller k value detects more corners including what we would determine to be false positives (high accuracy, low precision). For our Harris detectors, we settled in the middle and used `k = 0.05` to get a good balance of detected corners.

IV. Observations

One of the first observations we noticed was that the `resize` function was initially cropping out portions of our image when we wanted to resize it. This was due to us accidentally specifying a `(0,0)` final image size. Swapping to `None` fixed our issues and we generated the reduced images (see Fig 1).

Figure 1: Reduced size images (0.5 scale)

Curious about the impact of image size on runtime, we ran our Harris functions with the images at their original resolution. Our `harris1` function took over 4 minutes to complete at full resolution compared to just 16 seconds at the reduced size. The speed of the built-in functions we used in our `harris2` function was apparent when we ran using the full resolution images. The full size images were processed by `harris2` in 0.567 seconds compared to just 0.32 seconds for the reduced images. Clearly we could optimize our own implementation much better.

We output the intermediate output of our corner detectors to visualize what is being detected. The intermediate output of our `harris1` function can be seen below in Figure 2.

Figure 2: `R function` Result (Image 1)

We can compare this to the intermediate output of our `harris2` function seen below in Figure 2-2, and see that there are considerably more false detections occurring in our `harris1` function, the output image looks much "noisier". The `harris2` result has many more detections in "clusters" around the corners. These results are before being put through the same thresholding function, so it is apparent that `harris2` is much more accurate even though both are using a harris k parameter of `0.05`.

Figure 2-2: `cornerHarris` Result before Thresholding (Image 1)

Similar to the results for image 1, we can see the intermediate outputs for image 2 from the `harris1` and `harris2` functions below in Figures 3 and 3-2.

Figure 3: `R function` Result (Image 2)

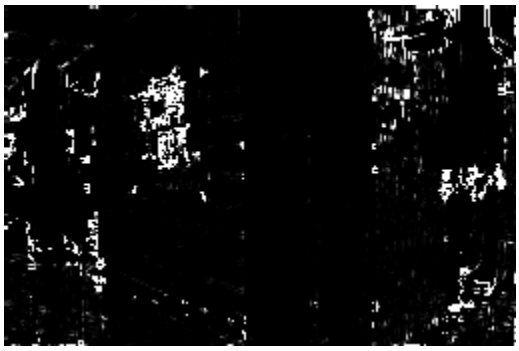


Figure 3-2: `cornerHarris` Result before Thresholding (Image 2)

These intermediate results are before the images are fed through the same thresholding function to down select the best matches for desirable corners. After thresholding, we then further refined our corner selections in `harris1` by searching for local maxima within our corner clusters. The results can be seen below in Figure 4. The top image represents the final corner outputs and the bottom image represents the input image combined with the corner mask. We can see the corner clusters line up with what we would consider corners.

Figure 4: Thresholded R (Corners) and Detected Corners on Input Image 1

Our `harris2` function produced similar results. The extra subpixel refining performed in `harris2` means that our corner clusters are much smaller (often just a single pixel). The results for image 1 from `harris2` can be seen below in Figure 4-2, with the top image showing the detected corners and the bottom image the combination of the input image and the corner mask.

Figure 4-2: Harris2 Function Results (Image 1)

Similar to the results for image 1, we can see the outputs from the `harris1` and `harris2` functions for image 2 below in Figures 5 and 5-2.

Figure 5: Thresholded R (Corners) and Detected Corners on Input Image 2

Figure 5-2: `Harris2` Function Results (Image 2)

Once we had our corners detected, we then needed to find matching corners in both image 1 and image 2. First, we determined the coordinates of our corners in each image. Then we took patches from the input images at those coordinates and calculated the normalized cross correlation of the patches. This was easily done with the `matchTemplate` function and specifying the `TM_CCORR_NORMED` flag. This was after attempting to do the matrix math and calculations ourselves and not quite getting the right values. When we found the highest correlation we stored the value and considered the patches to be matched. We then draw lines between the matching patches that have a normalized cross correlation greater than `0.9`. The matched corners between images 1 and 2 can be seen below in Figure 6.

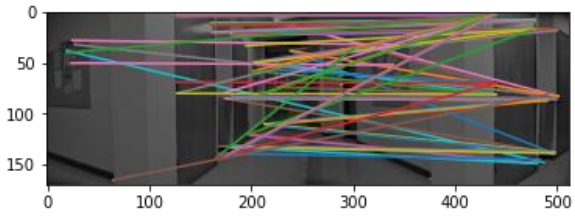


Figure 6: Detected Matches between Image 1 and Image 2

The matched corners are then run through our RANSAC algorithm implementation where we use a SIFT detector to find our corners and keypoints based on the supplied corner mask [5]. Once we find the key points, we can use the `findHomography` function with the `RANSAC` flag to calculate our homography matrix, seen below in Figure 8. This homography matrix is what we used to transform the perspective of our second input image to resemble that of our first input image. This transformation was performed using the `perspectiveTransform` function. The combination image of our matched correspondences, estimated homography, and our transformed image 2, can be seen below in Figure 7.

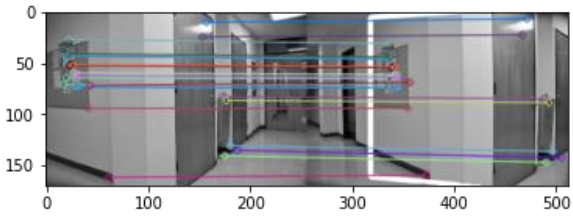


Figure 7: Estimated Homography and Correspondences

```
Homography Matrix:
[[ 7.81863631e-01 -1.62770452e-03  6.27965146e+01]
 [-6.30352132e-02  9.21918867e-01  6.49929576e+00]
 [-8.71796968e-04  3.13716547e-05  1.00000000e+00]]
```

Figure 8: Homography Matrix

Once we had our homography matrix, we were ready to warp the transformed image but first we had to determine the final size of our image. We decided on a simple doubling of the image width since both images were of the same dimension and that was the maximum size possible if there was zero overlap in our input images.

Using the `warpPerspective` function, we transformed our image 1 with our homography matrix and then matched it with the right position in the output image. With our warped image in place, we could then add the second image to our output image. The output of our combined image mosaic can be seen below in Figure 9.

Figure 9: Combined Image Mosaic

We left the output image in its original size but we could easily crop out the zero section and measure the size of that image to determine the final output image size.

The images already blend together pretty well when our warped image was added to the mosaic but we can still see a line underneath the second door from the right in Figure 9. This is caused by the reflection of the door in the tile being overlapped with the warped image.

Figure 10: Blended Mosaic

To reduce this artifacting, we first took a copy of our warped image and ran it through a gaussian filter to slightly blur the image. We then used this blurred image and added it to a copy of our original warped image using the `addWeighted` function. This had the effect of blending our final output image a bit although the artifacting can still be seen. Our final blended image mosaic can be seen above in Figure 10.

V. Conclusions

In this project we demonstrated the power of the Harris corner detector and how finding the right threshold can narrow down the total number of corners detected to a much more reasonable number with very few false positives. We discovered the impact of image size on program runtime and why reducing the total image size can be beneficial for corner detector accuracy. We compared and contrasted our own implementation of a corner detector with that of the built-in functions of the `cv2` python package.

Once we had a good set of corners that we believed to be accurate, we then searched the patches around those detected corners and compared their cross correlations. Patches with correlations close to 1 were considered to be the best matches. We used the `RANSAC` algorithm to sort the inliers from the outliers in our corner matches and then used those inliers to compute a homography.

The homography allowed us to warp an input image to match another image's perspective and create a mosaic of our input images. We finally blended the images to reduce any obvious artifacting.

VI. Appendix

[1] Project 2 Repository
https://github.com/scheung97/EECE5639_ComputerVision/tree/main/Project2

[2] Project 2 Repository
https://github.com/scheung97/EECE5639_ComputerVision/blob/main/p2.ipynb

[3] OpenCV » Harris Corner Detection
https://opencv24-python-

tutorials.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_features_harris/py_features_harris.html

[4] What does k mean in cornerHarris?
https://stackoverflow.com/questions/54720646/what-does-ksize-and-k-mean-in-cornerharris/54721585

[5] Matching + Homography to find Objects
https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_feature_homography/py_feature_homography.html