PART-2 Manual Mosaicing

Q.1 How many point-correspondences did you choose? Which OpenCV library function did you use to find the transformation in (2)? and why?

Ans: Since we need to find a transformation matrix that transforms **I1** to the perspective of **I2**. We need at least 4 point-correspondences for each input pairs to have 3x3 matrix which will do the desired transformation. But choosing only 4 points doesn't give desired/proper/good result, so we deduced and also experienced that choosing more that 4 points (maybe 6 or 7) often gives good result.

Q.2 Consider a case in which we keep I2 as the source image. List the ways or properties in which output after mosaicing will change and, the ways in which it will remain the same.

Ans: In our case we are keeping **I1** as source image, overlapping is being done on the right side of **I2** and left side of **I1** as seen from the observer. In case we do the opposite as asked in the question, output will change when

- Overlapping was done on the same side as it was for the other case.
- **I2** still holds the maximum of view s compare to **I1**
- Output will not change if overlapping side is chosen properly and **I1** holds the maximum of view.

Q.3 Consider two 1-D images J1 and J2 of size 100 pixels each. Both share some common points (correspondences). J1 has correspondences in range of [10,20] and J2 has correspondences in the range of [85,95]. Will the size of the output image (after mosaicing J1 and J1) change if we keep J1 as source image vs if we keep J2 as the source image. Explain.

Ans: Points which are common will overlap and extra parts in both the images will take extra space, so it doesn't matter as overlap will happen one way or another ([10,20] over [85, 95] OR [85,95] over [10,20]). Hence, size will not change whether we take **J1** or J2 as source image.

Q.4 Explain your choice of the affine transform.

Ans: For the normalization of points, **'Hartley Algorithm'** is used (Ref: Wojciech Chojnacki, Michael J. Brooks, Anton van den Hengel, Darren Gawley, "Revisiting Hartley's Normalized Eight-Point Algorithm," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 25, no. 9, pp. 1172-1177, Sept., 2003). Affine transform used here is **'Scaling and translation'.** These affine transform is basically used to take points X to a new set of points tilde(X) such that the centroid of the points tilde(X) is the coordinate origin (0, 0) , and their average distance from the origin is sqrt(2).

Q.5 Explain the relationship between H and Hn.

Ans: As per the reference mentioned above relationship between Hn and H is given by, **H = T'^(-1) * Hn * T**. T' and T are normalization matrix (Ref. Hartley Algorithm).

Q.6 Design an error metric and report the percentage error while using normalization and not using normalization.

Ans: We have used two error metric for this task, one is RMSE (Root mean square errror) and the other one is 'Symmetric Transfer Error' $d_{transfer}(\mathbf{x},\mathbf{y})^2 = d(\mathbf{x}, H^{-1}\mathbf{y})^2 + d(\mathbf{x}, H\mathbf{y})^2$.

RMSE reported error between both images: 9.2

Symmetric Error Unnormalized: 14.494

Symmetric Error Normalized: 8.448

Percentage error: |14.494-8.448| / 14.494 * 100 = 41.71%

Part-3 Auto mosaicing

3.1 RanSaC subroutine

1. The number of iterations N is given by,
   $N = \log(1-p)/\log(1-w^s)$
   For non-adaptive mode, w=0.5, p=0.99 given
   For different values of s, we get N proportional to it,
   Example,
   s=4 → N=72
   s=5 → N=146
   s=6 → N=293
   We took the ceil of value N in code.
   The model does a good job of inlier detection even with s=4 (which is the least number of points required to fit the model). We recommend s=4 as it is faster and increasing s can also adversely affect the error and increase it.
2. The parameter T is given by,
   T = w*n
   Where, w → probability of a point being an inlier
   n → Total number of points
   Example, in code n is close the the below given value
   n = 52, w = 0.5
   T = 26
   a. As, this value T could be very large if n is large and we start with w=0.5.

b. For this, we have introduce a dampening factor k=0.99, that lowers T every time consensus set size < T
c. T is used in all the iterations not permitting a consensus set of size less than T. The algorithm says, we should stop when consensus set size is greater than T threshold. But, to find the best possible consensus set, we continue for all N iterations. Hence, the number of times this parameter was used would be N (100%).
d. If we stop when the first consensus set size > T, implies we stop as soon as the ratio of inliers to total points are more than w (but definitely not the best consensus set in majority of the cases).
e. One can also stop checking the condition consensus set size > T after the very first time it is satisfied, as the best consensus set will only change when the new consensus set will be of size greater than the current best (which definitely has size > T). This will change the percentage of times T was used, and will be the **ratio of number of iterations for inlier percentage to be greater than w to N.**
f. The dampening was done on T in 1.36 % of the iterations when s=4. (we found the dampening saturates after a particular value of T) (This metric is reported for auto.py –normalize 0)
g. Both T and N change adaptively when adaptive ransac is used. But, above holds for the non-adaptive version as asked.

3. Parameter t is used to classify a point into inlier and outlier
   a. If distance of a point < t then, the point is an inlier
   b. We have set, t = sqrt(5.99)*threshold, the root of 5.99 is due to points being 2D.
   c. As explained in the slides, threshold parameter is standard deviation of gaussian measurement error, probability α the point is an inlier = 0.95. If we use the square of the distance to the line (say d^2) as the random variable, then the distribution is a chi-squared distribution with degree of freedom =2 (that is the codimension, for a point = 2). An inlier will be rejection no more than 5% of the time
   d. The square of the distance is the sum of squared x and y measurement errors. (basically euclidean, we have also made provision for symmetric distance in the code)
      i. Euclidean distance error $d(\mathbf{x},\mathbf{y})^2 = \sum_i (x_i - y_i)^2$
      ii. Symmetric transfer error $d_{transfer}(\mathbf{x},\mathbf{y})^2 = d(\mathbf{x}, H^{-1}\mathbf{y})^2 + d(\mathbf{x}, H\mathbf{y})^2$
      H is homography model fit on sample points
   e. Hence, $d^2 < t^2 = 5.99\sigma^2$, here σ is threshold parameter.

4. The largest consensus set size will be the maximum number of inliers. An acceptable consensus set size is any size > T. Being ambitious, one could iteratively keep checking for N iterations and keep updating consensus set if it's size is greater than or equal to (pick one with lower standard deviation) current best consensus set size.

3.2 Mosaicing Implementation

Tweaking of threshold parameter:
We found for normalized points it is generally the case that a lower threshold (like 0.1) will also do a good job but for unnormalized points higher threshold works the better. (decreasing it reduces the number of inliers as expected and vice versa). (below table if for auto.py campus images). We have set threshold=0.5 for unnormalized images, and threshold=0.01 for normalized images.

Normalize	Threshold	Total features points	Inliers found	Points Pruned(%)
0	0.2	52	36	0.3077
0	0.5	52	47	0.0962
0	1	52	49	0.0577
0	2	52	52	0.0
1	0.005	52	46	0.1154
1	0.01	52	48	0.0769
1	0.1	52	52	0.0

Comparison with stitcher API:
Observations:

- No intensity differences between the images after stitching with Stitcher API
- The order in which images stitched does not matter with Stitcher API
- We can’t say in which order the images are stitched by using Stitcher API
- With Stitcher API the image looks smoother
- Images attached below for reference

Fig. Stitcher API Campus

Fig. Stitcher API Lake(Own dataset images)

Fig. No norm campus with manual.py

Fig. No norm Lake with manual.py

Fig. Norm campus with manual.py

Fig. Norm Lake with manual.py

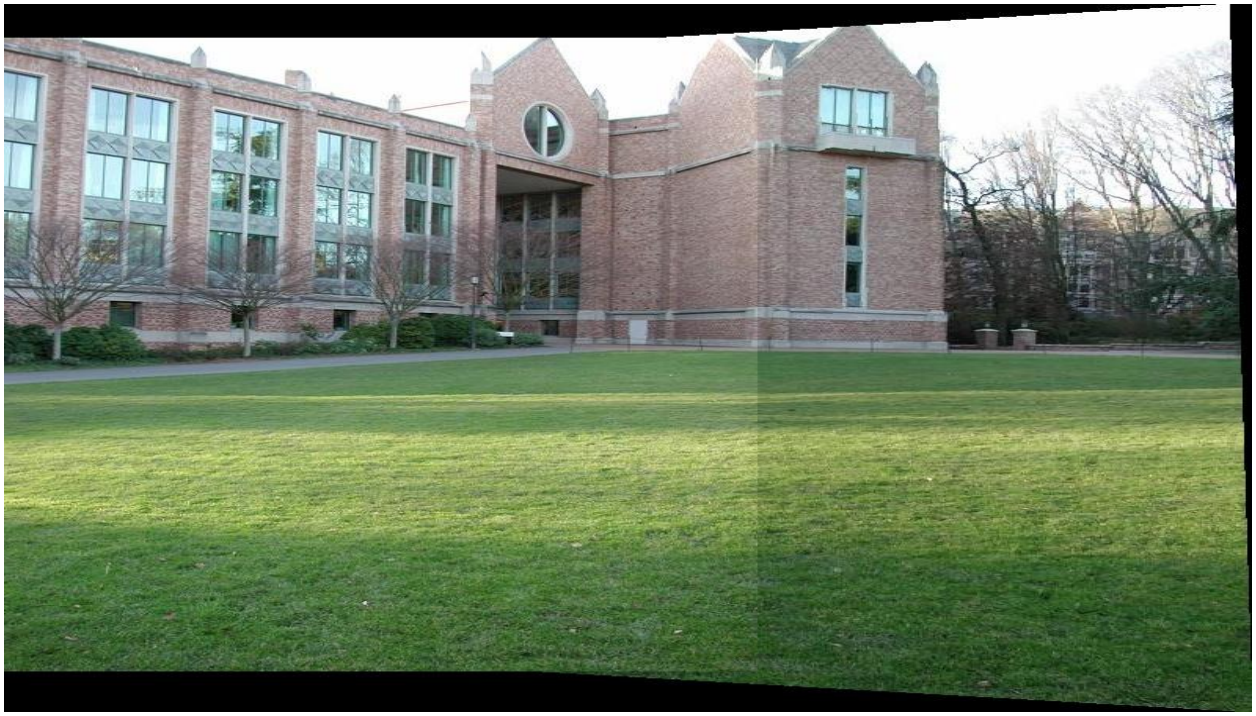

Fig. Auto.py campus

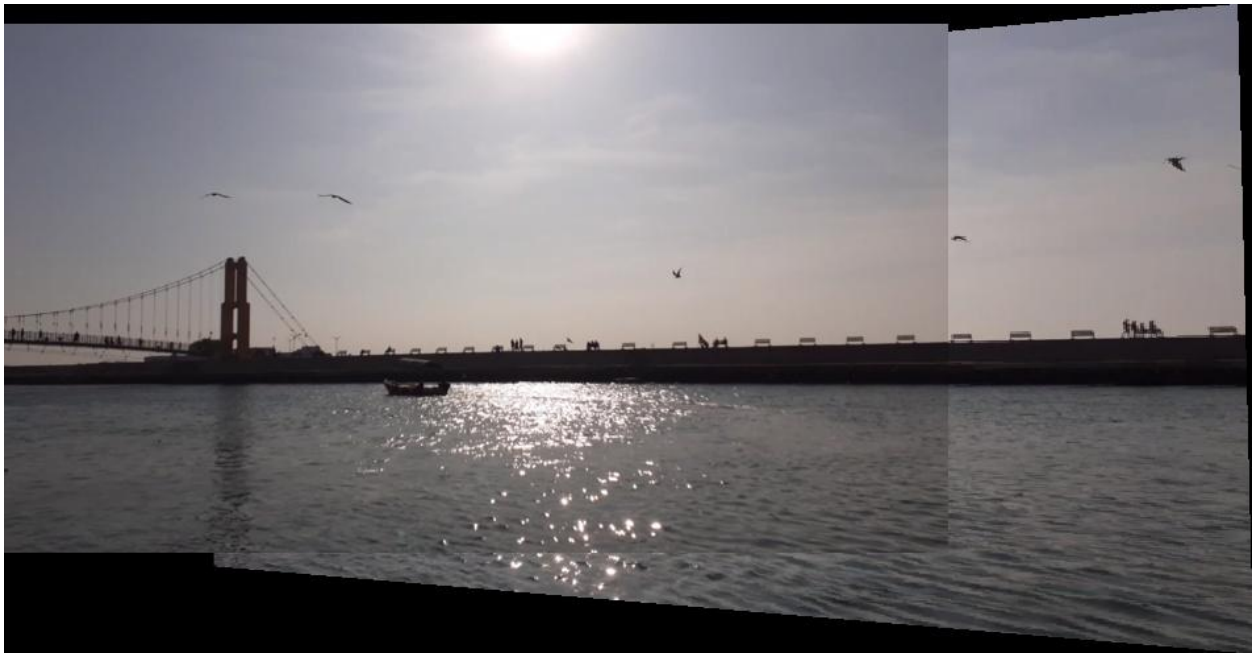

Fig. Auto.py Lake

Q.1 Report the number of feature points used for campus and the percentage of pruning due to your method:
Top 10% of features found using SIFT as detector and descriptor are used. Matcher used is brute force using L2 norm and crosscheck=true. The Ransac threshold is set the same as mentioned above. (threshold=0.5 for unnormalized images, threshold=0.01 for normalized images.)

Version	Normalize	Total features points	Inliers found	Points Pruned(%)
Custom (threshold=0.5)	0	52	47	0.0962
Custom (threshold=0.01)	1	52	48	0.0769
Inbuilt (threshold=0.5)	0	52	29	0.4423
Inbuilt (threshold=0.01)	1	52	41	0.2115
Inbuilt (threshold=1)	0	52	45	0.1346
Inbuilt (threshold=0.05)	1	52	50	0.0385

The distance measure of inbuilt ransac seems to be more sensitive (and penalizing) to the threshold parameter than custom ransac. Higher threshold was needed for good results in inbuilt ransac compared to custom ransac.

Q.2 Consider the personalized (“group”) dataset you used earlier for the manual case where you handtweaked the correspondences. Does auto-mosaic work with this dataset? Can you think of a dataset that breaks auto-mosaic but will (obviously) work with the manual case.
The personalized dataset worked fine on auto mosaicing but, when the features detected are not good enough, basically faulty detector, descriptor pair then, auto-mosaicing will fail. Also, when the images contain symmetric and similar structures like pillars in Taj Mahal automatic point correspondences fail quite often. That is when, manual method should work.

Part-4 Let's generalize

Q.1 Explain the method you followed in accomplishing this task.

We have stitched the images one by one. Starting from the reference image we stitch an image to it that has a significant amount of keypoint matched as inliers. Basically the idea is to stitch two images together that are similar to each other. This will avoid any issue as there will always be an overlap between two images being stitched together. For stitching two images together, we first find the Homography matrix H from $image_1$ to $image_2$ and then transform the $image_1$ using warpperspective. We have to manipulate the image size so that the whole image is visible. It is to be noted that H can generate values that might be very small or even negative. To display the image we need to translate the current minimum (x,y) to origin. Hence now along with homography we have to do translation as well. Pre-multiplying H with translation matrix will give us the resultant transformation.
After transforming $image_1$, we just have to pick all points in $image_2$ (that are not absolute zero) and merge it with transformed $image_1$. The size of transformed $image_1$ is set to the difference of maximum and minimum possible values of (x,y). This makes sure that the whole image will be visible. The pixel locations for $image_2$ will be from minimum to the length of $image_2$ for that particular axis (x & y or height & width).
Here it is assumed that image order is given.

Q.2 Show the results for the given datasets as well as on your own datasets (of 5 images). Mention the results when your code is working and when it is not working for some reference images or some datasets (if that's the case).

Code worked for the real dataset for any image being the reference image. The order of stitching is set to bottom-up based on inlier ratio as similarity between images. The only difference is the inlier ratio threshold which classifies whether two images are related or not had to be lowered for real image setting, as fewer inliers were found in real images. Also, normalization helps for faster code, as sometimes things get stuck without normalization.

Q.3 Do you get the result if you choose any image to be the reference image? If no, what issues are you facing? Give your opinion on the reason behind these issues.
Ans: For stitching of 5 images, the code doesn't break for any reference image. But, as we move to more images(7) the code fails for some cases. On investigating, it was found that jagged lines after warping were also considered as features, which inevitably caused issues in homography. This was specifically the case when normalized. Also, without normalization, the H matrix becomes very large and slows down everything. If the reference image was used which was not in the center of the whole stitched image, this also caused issues. As we continuously stitch it gets sidelined and becomes smaller and smaller, causing more errors in finding homography.

Q.4 Here, we assumed that you know the sequence in which you have to stitch the images. How would you modify your approach if you don't know the sequence? Describe the approach briefly.
We have made provision for determining the stitching order by making an inlier similarity-based graph (an undirected adjacency list like structure with most similar image corresponding to a particular image is at the beginning of the list for that particular image) and then applying BFS (Breadth first Search) on the graph. The BFS tree will be used to determine the stitching order in case no order (--order parameter is given see code) is given. So, the use of BFS is basically to determine which two images should be stitched together first and so on, making clusters of different stitched images.

For stitching, we use the BFS tree in bottom-up fashion. After generating the graph, we apply BFS and get the BFS tree. Starting from leaf nodes we stitch all the leaves with their parents as reference. After all the leaves are stitched with their parents, now these parents become new leaves and this will propagate upwards from bottom till the root. And the root will be our complete stitched image.

Q.5 How does the functionality you are providing (or could provide) differ from the Stitcher API.

Figure with general.py

Figure with StitcherAPI

Observations:

- No intensity differences between the images after stitching
- The order in which images stitched does not matter with Stitcher API
- We can't say in which order the images are stitched
- With Stitcher API the image looks smoother

Q.6 List items that you think need fixing. For example, the mosaicing results show a "seam". What techniques can be used to remove the seam?

a. The intensity across the images are different. For that we have to do intensity smoothing across all images to get a continuous image look.
b. The edges of the images after warping are jagged. Blurring and smoothing (anti-aliasing) is needed.
c. Seam is basically like the vacant gap (with low energy) that can be reduced so that images can fit or be shrunken realistically. Seam carving technique with different energy functions like gradient magnitude, entropy, visual saliency, eye-gaze movement, etc. can be used. (source)