CS4277/CS5477 Assignment 2: Metric Rectification and Robust Homography

1. Overview

In this assignment, you will implement two approaches for metric rectification and RANSAC algorithm for robust homography estimation.

References: Lecture 3 & 4

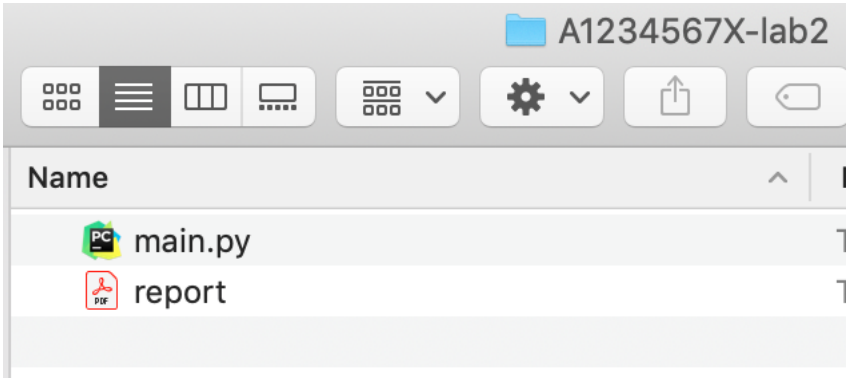
Honour Code. This coding assignment constitutes **10%** of your final grade in CS4277/CS5477. Note that plagiarism will not be condoned! You may discuss with your classmates and check the internet for references, but you MUST NOT submit code/report that is copied directly from other sources!

2. Submission Instructions

Items to be submitted:

- **Source code (main.py).** This is where you fill in all your code.
- **Report (report.pdf).** This should describe your implementation and be no more than one page.

Please clearly indicate your name and student number (the one that looks like A1234567X) in the report as well as the top of your source code. Zip the two files together and name it in the following format: **A1234567X_lab2.zip** (replace with your student number). Opening the zip file should show:



Submit your assignment by **16 February 2021, 2359HRS** to LumiNUS. 25% of the total score will be deducted for each day of late submission.

3. Getting Started

This assignment as well as the subsequent ones require Python 3.6, or later. You need certain python packages, which can be installed using the following command:

```
pip install -r requirements.txt
```

If you have any issues with the installation, please post them in the forum, so that other students or the instructors can help accordingly.

4. PRELIMINARY: FUN WITH HOMOGRAPHIES

4.1 Transformation using provided homography matrix

In this part, to ensure you understand the homography transformation, you will implement the function that enables transformation on 2D points given the homography matrix.

Implement the following function: transform_homography( )

- Prohibited Function: cv2.perspectiveTransform( )

4.2 Image Wraping Using Homography

In this part, you will implement the image wraping function. Given an image I_{src} and a homography matrix H, the transformed image I_{dst} can be obtained by relating corresponding coordinates in two images as:

$$x_{dst} = Hx_{src} ,$$

where x_{src} and x_{dst} are pixel coordinates of I_{src} and I_{dst} respectively.

The objective is to wrap the source image I_{src} onto the I_{dst} by providing the homography matrix H. To solve this problem, there are two ways: 1) one obvious way is doing forward-interpolation as the equation shows. For each pixel in I_{src}, you can compute its corresponding location in I_{dst} and then copy the pixel value over. However, this will result in holes in the image. 2) A better way to circumvent the hole problem is backward-interpolation, i.e. doing it in the reverse direction by computing the corresponding coordinates in I_{src} for every pixel in I_{dst}.

You will implement the second backward-interpolation method. Steps can be followed as:

1. First create a coordinate matrix $M \in R^{H_{dst}W_{dst}\times 2}$ which contains coordinates of all pixels in I_{dst}, i.e,

$$\begin{bmatrix} 0 & 0 & 0 & \dots & W_{dst}-1 & W_{dst}-1 \\ 0 & 1 & 2 & \dots & H_{dst}-2 & H_{dst}-1 \end{bmatrix}^T ,$$

   where H_{dst} and W_{dst} are the width and height of I_{dst}.

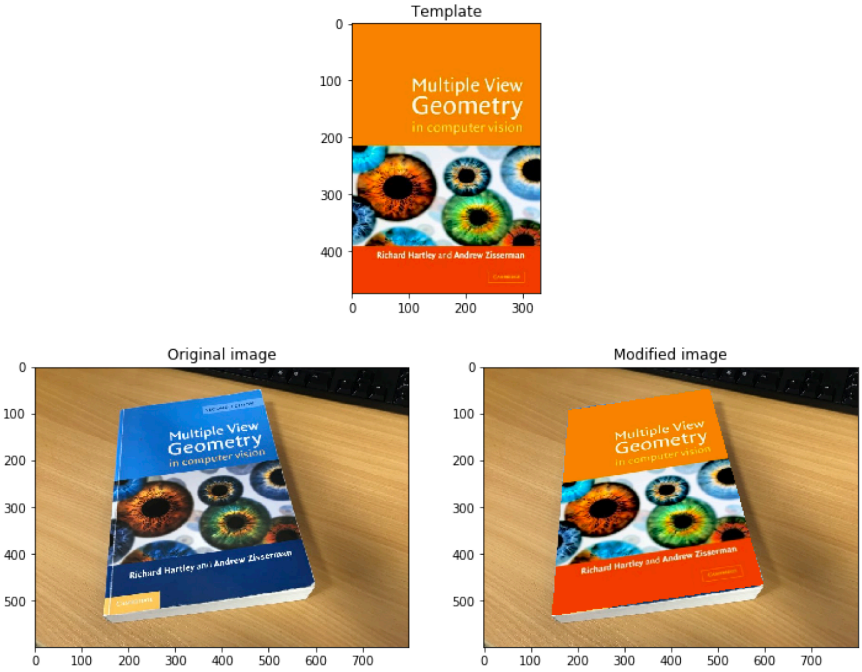
2. Use the function transform_homography( ) you implemented in previous section to compute the coordinated in I_{src}.
3. For every pixel in destination image I_{dst}, fill its value with the corresponding pixel in source image I_{src}. You can use interpolation method to handle non-integer source pixel coordinates.

Implement the following function: wrap_image( )

- Prohibited Function: cv2.wrapPerspective( )
- You may use the following functions: cv2.remap( ), np.meshgrid( ), transform_homography( )

If you use cv2.remap, you might find the borderMode value of cv2.BODER_TRANSPARENT useful. Also consider using bilinear interpolation in cv2.remap.

Your will be provided with a test example by replacing the book cover with a template image. If all functions are implemented correctly, you will get results like:



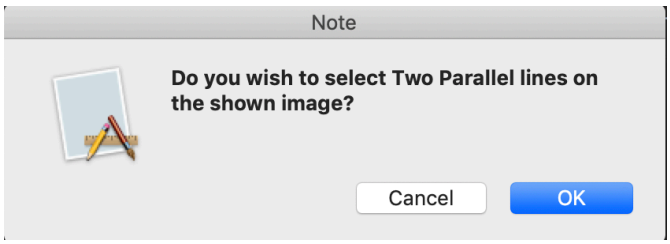
5. PART 1: METRIC RECTIFICATION

In this part, you will implement two approachs for the rectification of plannar surfaces imaged under perspective projection. As discussed in the lectures, metric structure recovery can be stratified by determining the affine property first and then metric property. Affine property recovery (parallelism, ratio of lengths) requires identifying the line at infinity I_∞ to remove projective distortion. Then, metric property recovery removes the affine distortion by specifying the image of the circular points. Finally, the remaining distortion is a similarity. Another one-step metric rectification approach directly uses five orthogonal line pairs to generate linear constraints on the rectification parameters.

Now let's have fun with an interactive interface, which enables you to choose line constraints on your own to perform metric rectification. However, as precision errors usually exists when selecting points on the image, the final rectified image will not be exactly the same as those shown for demonstration. And different constraints selected each time will also result in slight differences. Therefore, small precision errors can be tolerated in your final rectified results shown in the report. Nonetheless, you are recommended to select points more precisely to ensure your implemented functions work well.

Steps to use interface:

1. When you run test example, a pop-up window will be shown to confirm whether you need to choose line constraints.



2. If you wish to select points, click "OK" button and do your selection on the shown image. If 'Cancel' is clicked, current constraints will be read from file ("data/*.csv") directly for rectification.

3. When finishing your selection on the image, press button "q" to exit the user interface.

5.1 Method 1: Two-Step Stratification

Consider points on the image plane x are related to points on the world plane x' as $x' = Hx$, where x and x' are homogeneous 3-vectors. The projective transformation H can be decomposed into a concatenation of three matrices H_S, H_A and H_P, which represent similarity, affine and pure projective transformation respectively:

$$H = H_S H_A H_P$$

Stage 1: Removing Projective Distortion.

To determine H_P, the line at infinity $I_\infty = [0,0,1]^T$ is required to be identified. As parallel lines on the world plane intersect at the vanishing points in the image and the vanishing points lie on I_∞, two or more vanishing points determine I_∞ and thus H_P which maps line at infinity I_∞ to finite line $I = [l_1, l_2, l_3]^T$ (i.e., $I = H_P^{-T} I_\infty$).

Implement the following function: compute_affine_rectification ( )

- You may use the following functions: np.cross( ), wrap_image( )

Stage 2: Removing Affine Distortion

After stage1, now we get the affinely rectified image X_a, and the objective is to find the affine transformation H_A. Recall the lectures, angles on the projective plane is invariant to projective transformation once the dual degenerate conic C_∞^* is identified. Suppose a pair of orthogonal lines l and m in the world plane correspond to lines l' and m' in affinely rectified image plane (i.e. $l' = {H_A}^{-T} l, m' = {H_A}^{-T} m, C_\infty'^* = H_A C_\infty^* H_A^T$), we can get:

$$l'^T C_\infty'^* m' = l C_\infty^* m = 0$$

Then we require two constraints to specify the 2 degrees of freedom of circular points in order to determine a metric rectification, which may be obtained from two pairs of orthogonal lines and written as :

$$(l'_1 m'_1, l'_1 m'_2 + l'_2 m'_1, l'_2 m'_2)s = 0$$

Thus s, and hence K, is obtained up to scale by Cholesky decomposition.

Implement the following function: compute_metric_rectification_step2 ( )

- You may use the following functions: np.linalg.cholesky(), np.linalg.svd( ), np.linalg.det(), wrap_image( )

5.2 Method 2: One-Step Using C_∞^*

The previous stratification method use the properties of C_∞^*only to remove for affine distortion. However, starting from the original perspective image of the plane, C_∞^* can also be used in a direct way to remove distortion up to similarity. This can be achieved by choosing five orthogonal line pairs to linearly constrain elements of C_∞^*. C_∞^* (hence $H_A H_P$) can thus be obtained by Singular Value Decomposition (SVD).

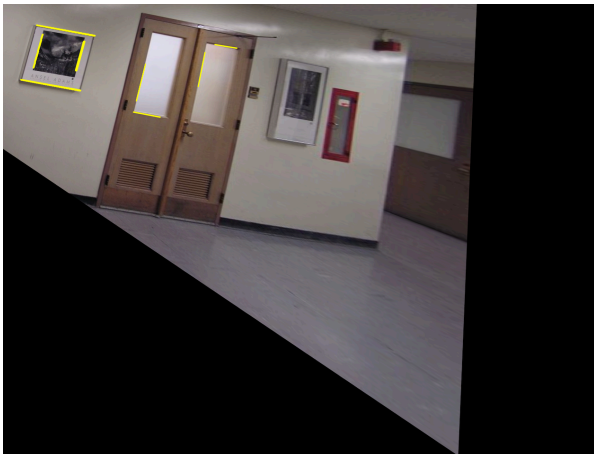
Implement the following function: compute_metric_rectification_onestep ( )

- You may use the following functions: np.linalg.cholesky(), np.linalg.svd( ), np.linalg.det(), wrap_image( )

If you implement above functions correctly, the results of provided constraints on image ("data/inputs/door01.jpg") can be shown like:

(a) Original Image

(b) Affinely Rectified Image (Stratification)



(c) Image after Metric Rectification (Stratification)

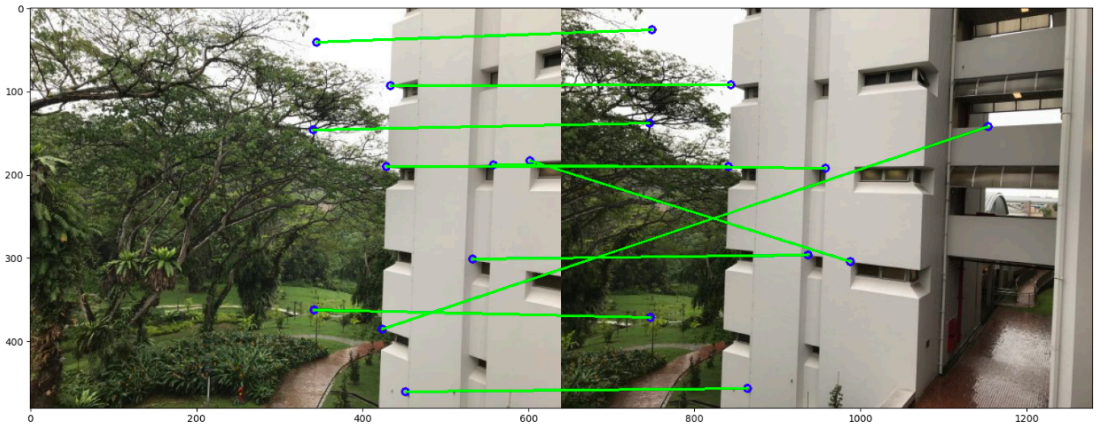
(d) Image after Metric Rectification (One-step)

6. PART 2: ROBUST HOMOGRAPHY

6.1 Robust Homography Estimation Using RANSAC

In this part, you will implement robust homography estimation using RANSAC algorithm in the presence of outliers.

1. First, let's see how outliers influence the homography estimation. Consider an image stitching example containing 10 provided correspondences of which 2 are wrong.



2. When just using the correct correspondences by removing outliers, it yields correct result:

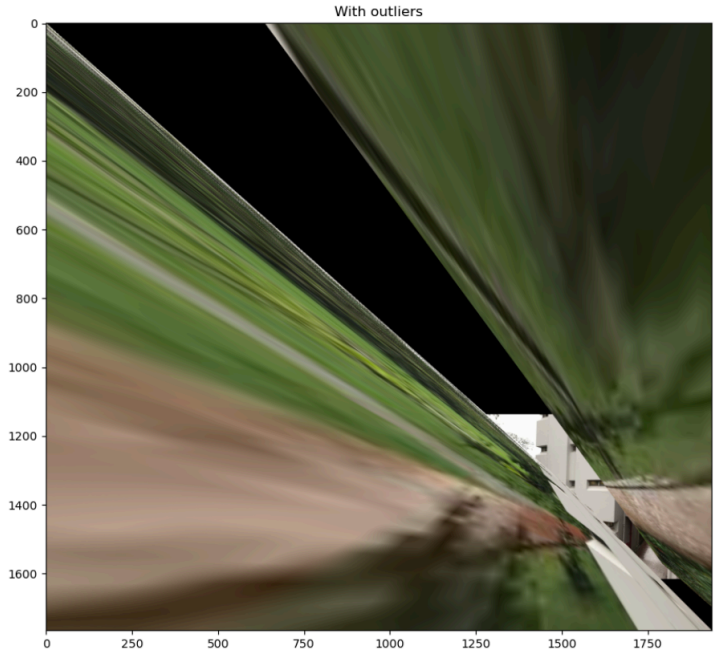
Computed clean homography matrix:

[[1.73813378e+00 9.49923000e-03 -4.46694994e+02]

[2.74543836e-01 1.51470187e+00 -1.21372176e+02]

[1.16001717e-03 2.07329780e-05 1.00000000e+00]]

3. However, if we include the two outliers, notice that the alignment fails:



The de-facto algorithm to solve such misalignment caused by outliers is RANdom Sample Consensus (RANSAC). You will implement the robust homography estimation to handle outliers.

As lecture shows, RANSAC requires an error function, i.e. shortest point to line distance, to compute which correspondences are outliers. Steps are as follows:

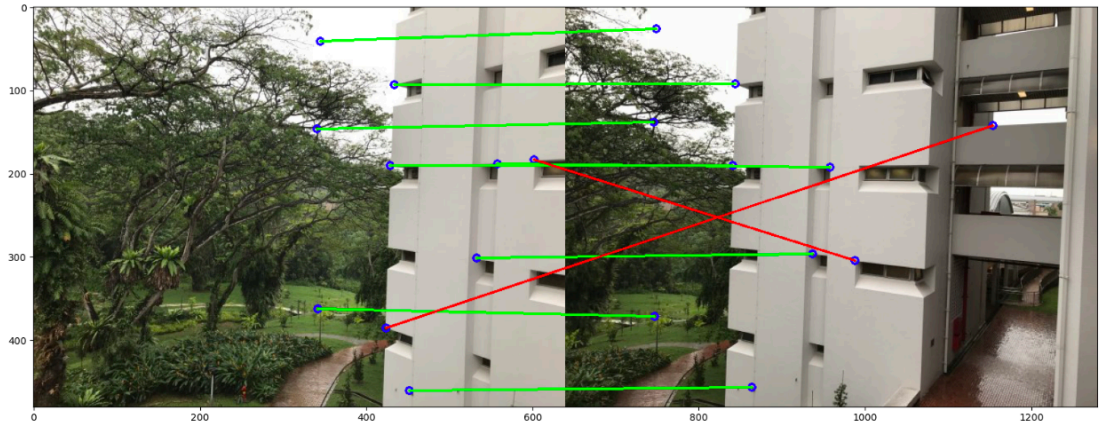
1. Implement the symmetric transfer error distance measure as described in the lectures:
$$d(x, x'; H) = \|x - H^{-1}x'\|^2 + \|x' - Hx\|^2$$

Implement the following function: compute_homography_error( )
- Prohibited Functions: cv2.findHomography( )
- You may use the following functions: np.linalg.inv(), and transform_homography( ) from the previous sections.

2. Implement the RANSAC procedure covered in the lectures, using the error function you just implemented:
(*Be sure to re-estimate the homography transformation using all inliers at the end of the RANSAC procedure*.)

Implement the following function: compute_homography_ransac( )
- Prohibited Functions: cv2.findHomography( )
- You may use the following functions: compute_homography( ) from the provided helper.py

If you implement above functions correctly, the images should be correctly aligned as that using only inliers. The RANSAC algorithm detects outliers (drawn in red) and excludes them from the homography computation. The computed homography matrix should be the same as the previous sections:



Computed homography matrix with RANSAC:

[[1.73813378e+00 9.49923000e-03 -4.46694994e+02]

[2.74543836e-01 1.51470187e+00 -1.21372176e+02]

[1.16001717e-03 2.07329780e-05 1.00000000e+00]]

At this point, we now can perform automatic alignment of the image pair. Recall in the lectures that we can detect keypoints and match them between the two images. The matches generally will contain outliers but your robust homography computation can handle those and still output the correct homography.

7. APPENDIX

SVD(Singular Value Decomposition)： The singular value decomposition is one of the most useful matrix decompositions, particularly for numerical computations.

1) Given a square matrix A, the SVD is a factorization of A as $A = UDV^T$, where U and V are orthogonal matrices, and D is a diagonal matrix with non-negative entries. The decomposition may be carried out in such a way that the diagonal entries of D are in descending order.

2) For non-square matrix $A \in R^{m\times n}(m \geq n)$, in this case, U is an $m \times n$ matrix with orthogonal columns, D is an $n \times n$ diagonal matrix and V is an $n \times n$ orthogonal matrix. The fact that U has orthogonal columns means that $U^T U = I_{n\times n}$. Furthermore U has the norm-preserving property that $\|Ux\| = \|x\|$ for any vector x. One the other hand, $U^T U$ is in general not the identity unless $m = n$.