Part 1 Documenatation:
Most of the documentation is in the code itself.

LOGIC USED:

I used ORB library to detect key points and based on the descriptors, I found the no. of matches between to images by
1) thresholding by distance less than 64(from David Lowe's paper on ORB and SIFT) [1]
2) then ratio to remove duplicate images again, the threshold used was 0.85(from David Lowe's paper again) [1]
[1] https://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf
3) I also used response value from orb function, returned in key points class variable where higher response equals higher chance of key point being a corner(a measure or cornerness). I take the top 65% of the data discarding the first 35% lowest values and the proceed to matching.

Basic K-means algorithm was used with an addition K-means++ which minimizes error generated due to random centroid selection in first iteration of K-means

Distance Metric used:

I first used the chamfer distance, but I improved on the accuracy by also involving symmetric matches, the function no_matches() which returns no. of match counts as well as end points of matches in the two images.

This data is used to create the distance metric matrix where for each image pair (i,j), the data obtained from no_matches() for i->j and j->i to find total matches either way(chamfer distance) and the no. of symmetric matches either ways, again. Now I noticed for some similar images(not exactly identical), the no of matches from i->j and j->i were nearly symmetric but the symmetric matches were really low, hence I added a condition where if the matches either ways is near as 80% of the other and the symmetric matches is less than 10, I increase the distance metric by a factor of 2 giving that image pair more weightage, also in cases where counts both ways weren't as near as less than 40%, then I punish the final image pair metric by a factor of 1/2.

PART 1 part 3:

Using the above implementation and the heuristics used in K-means, the accuracy on 93 images based on the formula(TP+TN/(n*n-1)) came out to be ~77%, where TP=60 and TN=6533 and n*n-1=8556.

Some tests:
```
python3 a2.py  part1 2 eiffel_18.jpg eiffel_19.jpg
bigben_3.jpg bigben_2.jpg bigben_12.jpg bigben_14.jpg oo.txt
```

```
Output: eiffel_18.jpg eiffel_19.jpg
bigben_3.jpg bigben_2.jpg bigben_12.jpg bigben_14.jpg
```

```
eiffel 18 and 19 are almost identical images and so are the group of
bigben images and the code clusters them perfectly, in perfectly
identical images, the implementation works quite well except in the
case of 93 images where sub images in a class are not exactly
identical or absolutely perfectly similar images, hence, a bit
```

wayward clustering.

Part 2:

Given any 4 correspondences in 2 images, we can easily find the transformation between the original image and the transformed image but for convenience and more accurate results the program is split into 4 sections or 4 different type of transformations that could have been applied.

1) Translation:

Translation is just moving the image in the x or y direction by some distance. Only one pair of points is required to find the transformation. The transformation matrix for translation is:

[1 0 tx]

[0 1 ty]

[0 0 1]

where tx, ty is the difference in x and y coordinates of the two images.

2) Euclidean:

Euclidean transformations consists of rotation or translations. Two pair of points are required to find the value of 4 unknowns in the transformation matrix, which is,

[cosθ -sinθ tx]

[-sinθ cosθ ty]

[0   0   1]

Where the 4 unknowns can be found by solving a series of linear equations as follows:

We can write the transformation matrix in the form,

[a -b c]

[b a d]

[0 0 1]

Writing in the form of homogeneous coordinates,

[[a -b c],[b a d],[0 0 1]] . [x,y,1]= [u,v,1]

Where x,y are the coordinates in the original image and u,v are the corresponding coordinates in the transformed image.

Multiplying the above eq. we get,

xa-yb+c=u

xb+ya+d=v

We get the resultant matrix form as,

[x, -y, 1, 0] [a,b,c,d]=[u,v]

[y, x , 0, 1]

For each point, we can solve these linear equations to get the values of 4 unknowns required to form the transformation matrix.

3) Affine:

Affine transformation consists of combination of linear transformations such as rotation, scaling, shear and translation. Using a set of 3 pairs of points we can find the 6 unknown variables required to formulate the transformation matrix. The transformation matrix for affine transformations is,

[[a, b, c],[d,e,f],[0,0,1]]

The transformation equation can be written as,

$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

Or

X=AU

Thus, we obtain A=U^{-1}X.

4)Projective:

Projective transformations are combinations of affine and projective warps. Using a set of 4 pairs of points we can find the 8 unknown variables required to formulate the transformation matrix. The transformation matrix for projective transformations is,

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

To solve the linear equations, we can use

$$\begin{bmatrix} u_0 & v_0 & 1 & 0 & 0 & 0 & -u_0x_0 & -v_0x_0 \\ u_1 & v_1 & 1 & 0 & 0 & 0 & -u_1x_1 & -v_1x_1 \\ u_2 & v_2 & 1 & 0 & 0 & 0 & -u_2x_2 & -v_2x_2 \\ u_3 & v_3 & 1 & 0 & 0 & 0 & -u_3x_3 & -v_3x_3 \\ 0 & 0 & 0 & u_0 & v_0 & 1 & -u_0y_0 & -v_0y_0 \\ 0 & 0 & 0 & u_1 & v_1 & 1 & -u_1y_1 & -v_1y_1 \\ 0 & 0 & 0 & u_2 & v_2 & 1 & -u_2y_2 & -v_2y_2 \\ 0 & 0 & 0 & u_3 & v_3 & 1 & -u_2y_2 & -v_3y_3 \end{bmatrix} \cdot A = X$$

Where, X = [x0 x1 x2 x3 y0 y1 y2 y3]^T are the known coordinates in the observed image and u_i,v_i are the coordinates of the source image.

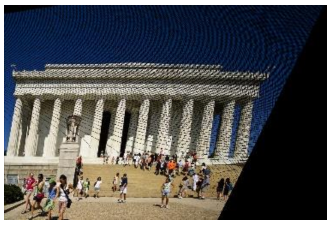
Referred from : http://graphics.cs.cmu.edu/courses/15-463/2008_fall/Papers/proj.pdf (Pg.3)

https://franklinta.com/2014/09/08/computing-css-matrix3d-transforms/

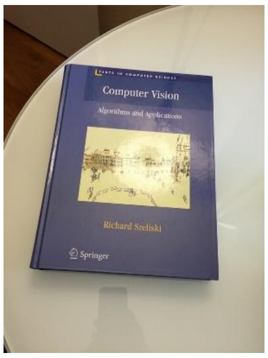
https://www.ldv.ei.tum.de/fileadmin/w00bfa/www/content_uploads/Vorlesung_3.2_SpatialTransformations.pdf (Pg.10)

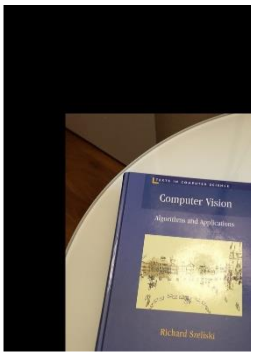
Result of the program:

1) Lincoln Image :

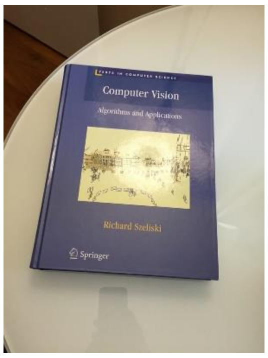


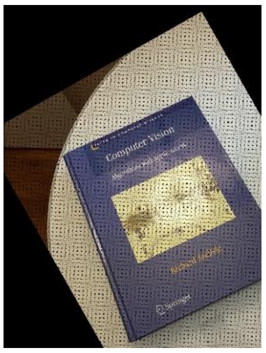
2)Translation transformation of book2.jpg:



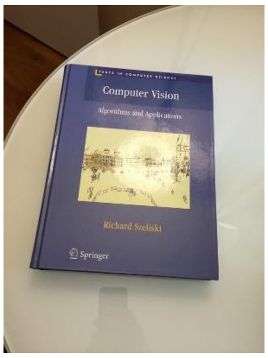


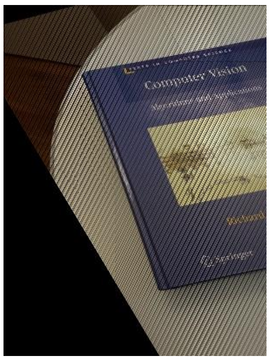
3)Euclidean transformation of book2.jpg:



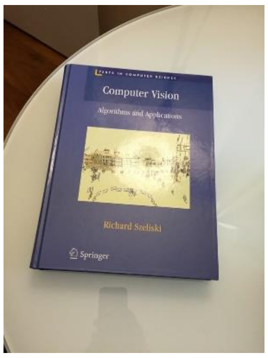


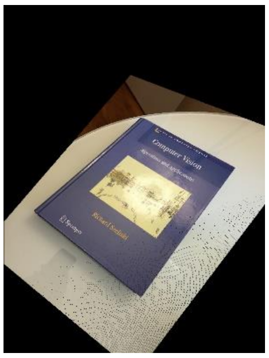
4)Affine transformation on book2.jpg:





5)Projective transformation on book2.jpg:





Report – Part: 3

Description:

Program for creating a panorama, the program takes three inputs, two images which will be stitched together and the name of resulting output image.

Run-time: 30secs – 1 min on personal computers

Algorithms/features from packages used:

1. ORB – Feature detector from opencv.
2. RANSAC

Implementation:

1. **Feature Detection:** We have used ORB – feature detector from opencv as indicated in the sample code. This gives us a list of keypoints and descriptors. In this case we are trying to detect around 1000 feature points.
2. **Feature Matching:**
   - We are calculating the Hammond distance between the features in order to match the detected features.
   - For this we have used a threshold of 10, so we say two descriptors/feature points match only if the Hammond distance between them is less than 10.
   - Next, we are storing the matched feature points in a list
3. **Calculating Transformation Matrix:**
   - In order to calculate the transformation matrix we are taking a set of four points from 1st image and another set of 4 corresponding matched points on 2nd image, and feeding them to an equation to calculate the transformation matrix.
   - The math behind solving this matrix equation was referenced from https://franklinta.com/2014/09/08/computing-css-matrix3d-transforms/
4. **RANSAC:**
   - At first we are calculating an initial hypothesis.
   - Next we are randomly selecting four pairs of matched keypoints from the list of matched keypoints which was formed earlier.
   - A new hypothesis is calculated based on these four points.
   - This new hypothesis is compared with initial hypothesis and if its similar then we vote for the initial hypothesis. Else if it's not similar we add this hypothesis as a new hypothesis.
   - Steps 2,3,4 are repeated for a certain number of times, in our case 80, however the number of iterations can be calculated using the formula: $1-(1-(1-e)^s)^N >= p$, as given in the slides by Prof. Crandell. However we do not know if we have the right samples, hence assuming number of iterations required to be 80.
   - Next, we find the hypothesis which has voted most number of times and use it further.
5. **Stitching:**
   - We are transforming the 2nd Image using the hypothesis given by RANSAC.
   - At last we are stitching the 1st and Image and the transformed 2nd image together.
   - The dimensions of the new image are calculated beforehand by finding the transformations of the corners of 2nd image and dimensions of 1st image.

Challenges and Design Decisions:

1. Finding out the transformation matrix was difficult, there were lot of trial and errors to find the right transformation matrix.
2. While matching the feature points between two images we are supposed to assume a minimum hamming distance between the feature points to call it a match. In ideal case this should be zero for perfect matches, however in practical this varies from 5 – 50(Atleast that's what we saw while testing).
3. For images in which the 2nd image is just a translation or rotation of 1st image we could set this threshold to 10 and find good enough number of matches, however for other transformations and depending on the clarity of the image this varies till 50 to get considerable amount of matches.
4. Setting this threshold high also results in many incorrect matches which again causes to detect wrong transformation matrix.
5. In RANSAC, we have set the number of iterations to 80, this could also vary depending on the number of good feature matches or the number of outliers we have. For some setting to 20 works, while for some even 120 doesn't give the right transformation matrix.
6. In RANSAC, while comparing two hypotheses, we are finding the element wise difference of hypotheses matrices, here as well we are assuming a threshold.
7. Calculating the dimensions of the new combined image also required a lot of debugging and trials.

Limitations:

1. The program might error out saying "numpy.linalg.linalg.LinAlgError: Singular matrix" sometimes, this is because during matrix calculation the randomly selected 4 the points sometimes have coordinates as multiples of another point and thus resulting matrix has a determinant 0 for which inverse cannot be calculated and hence equation cannot be solved. In this case please re-run with same inputs and it will work.
2. The program assumes that images to be stitched are given in order, i.e. in general, right part of the 1st image and left part of 2nd image have some or all overlapping features.
3. It works well for images where the 2nd image has features of 1st image which are translated or rotated in any direction or even small changes in scale.
4. For cases in which there is huge changes in scale or for drastic affine transformations, the program won't be able to find the feature matches correctly, or it might not be able to find the right transformation matrix because of fixed thresholds.
5. Since the thresholds are hardcoded, certain cases require tweaking.

References:

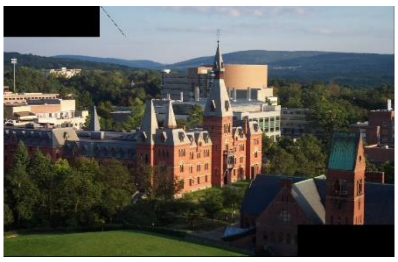
Solving matrix equations : https://franklinta.com/2014/09/08/computing-css-matrix3d-transforms/

Some Test Cases:

1. Taken from a2-Instructions PDF

1st part

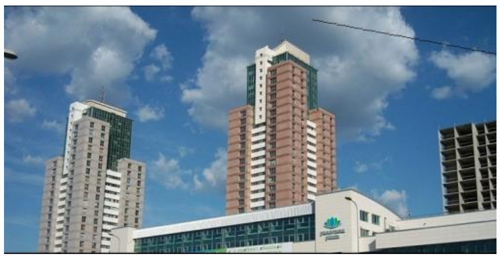
2nd part


combined

2. 2nd part in this was rotated 180 degrees

1st part

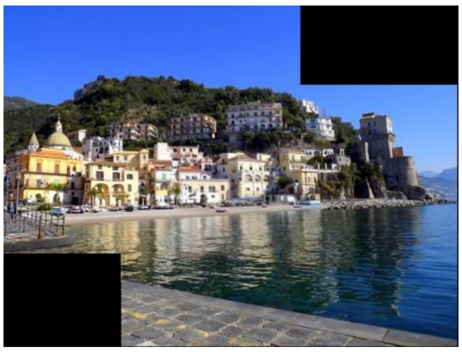
2nd part


combined

3. 2nd part was rotated 90 degrees

1st part

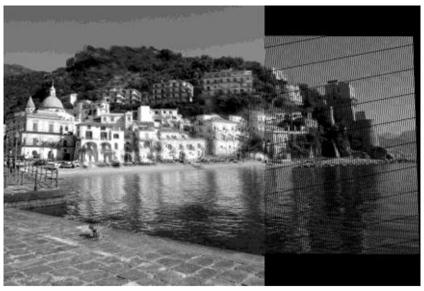
2nd part


combined

4. 2nd part has been scaled down along x-axis.
   This required changes in threshold to detect the right transformation matrix
   Also on transformation of 2nd image to match with 1st , caused degradation due to missing pixels.

1st part

2nd part (scaled)


3rd combined