Assignment 1:

Homography & Panorama

Ben Solomon 204454615

Din Carmon 209325026

1. While an affine transformation is used for scaling, skewing and rotation, a projective transformation is used to express how the perceived objects coordinates change when the view point of the observer changes.
   A projective transformation of coordinates (x,y) in one vector space to (x',y') on another vector space, can be expressed by the following equation:
   $$(x', y', 1)^T \cong H * (x, y, 1)$$
   Where H is a 3x3 matrix, and the ≅ sign represents an equality up to scale.
   Let's denote coordinates, and their projections with a subindex i.
   Thus, we can write the following equations for each pair:
   $$x_i' = C_i(H_{00}x_i + H_{01}y_i + H_{02})$$
   $$y_i' = C_i(H_{10}x_i + H_{11}y_i + H_{12})$$
   $$1 = C_i(H_{20}x_i + H_{21}y_i + H_{22})$$
   Where C_i holds a degree of freedom since we only demand equality up to scale. Therefore, each pair (and their projection) of coordinates only sum up to 2 constraints on the values of H which can be expressed with the following equations:
   $$\frac{x_i'}{1} = \frac{H_{00}x_i + H_{01}y_i + H_{02}}{H_{20}x_i + H_{21}y_i + H_{22}} \quad => \quad H_{00}x_i + H_{01}y_i + H_{02} - H_{20}x_i'x_i - H_{21}x_i'y_i - H_{22}x_i' = 0$$
   $$\frac{y_i'}{1} = \frac{H_{10}x_i + H_{11}y_i + H_{12}}{H_{20}x_i + H_{21}y_i + H_{22}} \quad => \quad H_{10}x_i + H_{11}y_i + H_{12} - H_{20}y_i'x_i - H_{21}y_i'y_i - H_{22}y_i' = 0$$
   Suppose we have n pairs of coordinates. Then we can express the equations as follows:
   $$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1'x_1 & -x_1'y_1 & -x_1' \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y_1' & -y_1'y_1 & -y_1' \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n & y_n & 1 & 0 & 0 & 0 & -x_1'x_1 & -x_1'y_1 & -x_1' \\ 0 & 0 & 0 & x_n & y_n & 1 & -y_1' & -y_1'y_1 & -y_1' \end{bmatrix} * \begin{bmatrix} H_{00} \\ H_{01} \\ H_{02} \\ H_{10} \\ H_{11} \\ H_{12} \\ H_{20} \\ H_{21} \\ H_{22} \end{bmatrix} = \bar{0}$$
   We denote this matrix as A – a matrix of size 2n x 9, and get a matrix of the form $Ax = b$ where the solution for x, describe H.
   Since the source equality is up to scale, there are actually 8 degrees of freedom for H. Thus only 4 pairs of coordinates are sufficient to find H (We assume that A is not defective).
   If n is bigger than 4, equality may not be possible (due to some error in the projective input). In such case, we strive to minimize $|b|$.
   One can notice that:
   $$|b| = \sqrt{b^T b} = \sqrt{(Ax)^T(Ax)} = \sqrt{x^T(A^T A)x}$$
   We denote $M = A^T A$ – a PSD matrix of size 9x9. Thus, our objective is to find x such that $x^T Mx$ is minimized.
   Assuming M has 9 linearly independent eigenvectors, we can define P_M – a matrix whose columns are the linearly independent eigenvectors of M, and D_M – a diagonal matrix whose diagonal elements are the corresponding eigenvalues of M.
   So:
   $$x^T Mx = x^T P_M D_M P_M^{-1} x$$

Each vector x can be described as a weighted sum of the eigenvectors of M. From here, it is trivial to see that the desired x that minimizes $|b|$ is the eigenvector corresponding to the smallest eigenvalue of M (which is necessarily not negative since M is a PSD matrix).

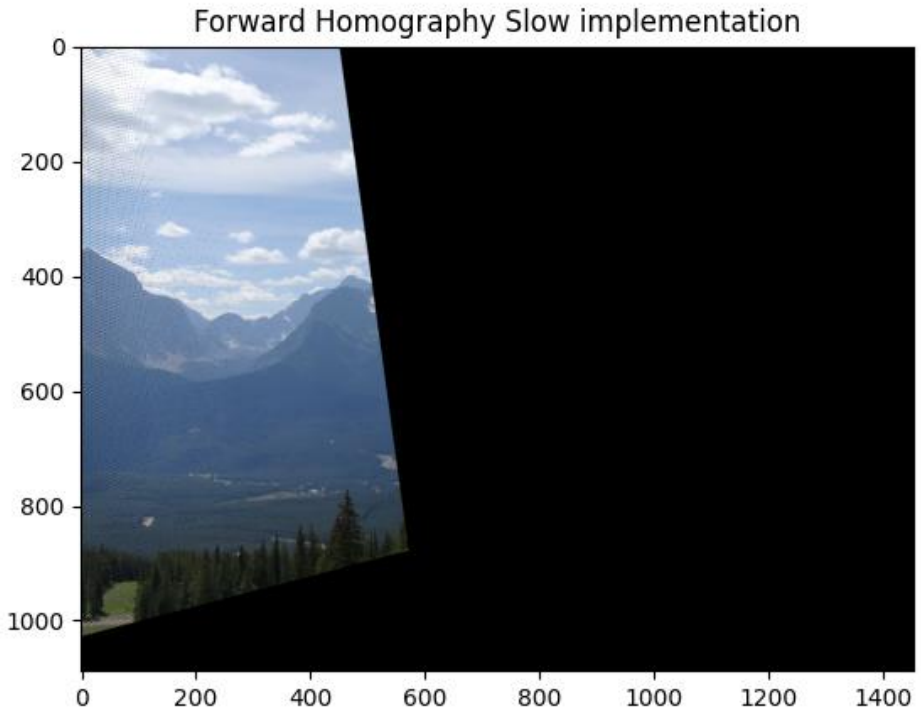
2. The function compute_homography_naive is implemented and can be viewed in the python src files.
3. After loading matches_perfect.mat and using the function implemented under section 2, we found the following matrix:

$$H = \begin{bmatrix} 1.12313787\text{e}-03 & 1.64757637\text{e}-04 & -9.99919589\text{e}-01 \\ 1.05114986\text{e}-05 & 1.05462486\text{e}-03 & -1.25618938\text{e}-02 \\ 2.96940477\text{e}-07 & 4.35704606\text{e}-08 & 7.82908438\text{e}-04 \end{bmatrix}$$

We remember that H is defined up to scale. Thus, a common approach is to represent H where $H_{22} = 1$. In such case:

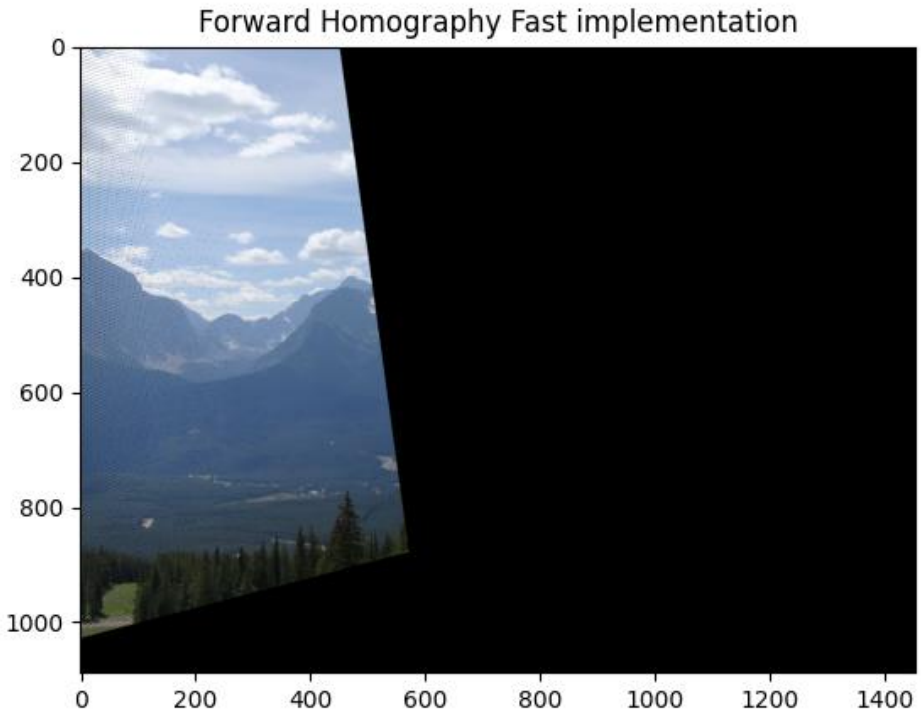
$$H = \begin{bmatrix} 1.43457116\text{e}+00 & 2.10443047\text{e}-01 & -1.27718586\text{e}+03 \\ 1.34262170\text{e}-02 & 1.34706028\text{e}+00 & -1.60451634\text{e}+01 \\ 3.79278677\text{e}-04 & 5.56520514\text{e}-05 & 1.00000000\text{e}+00 \end{bmatrix}$$

4. Below is the image generated using compute_forward_homography_slow function:



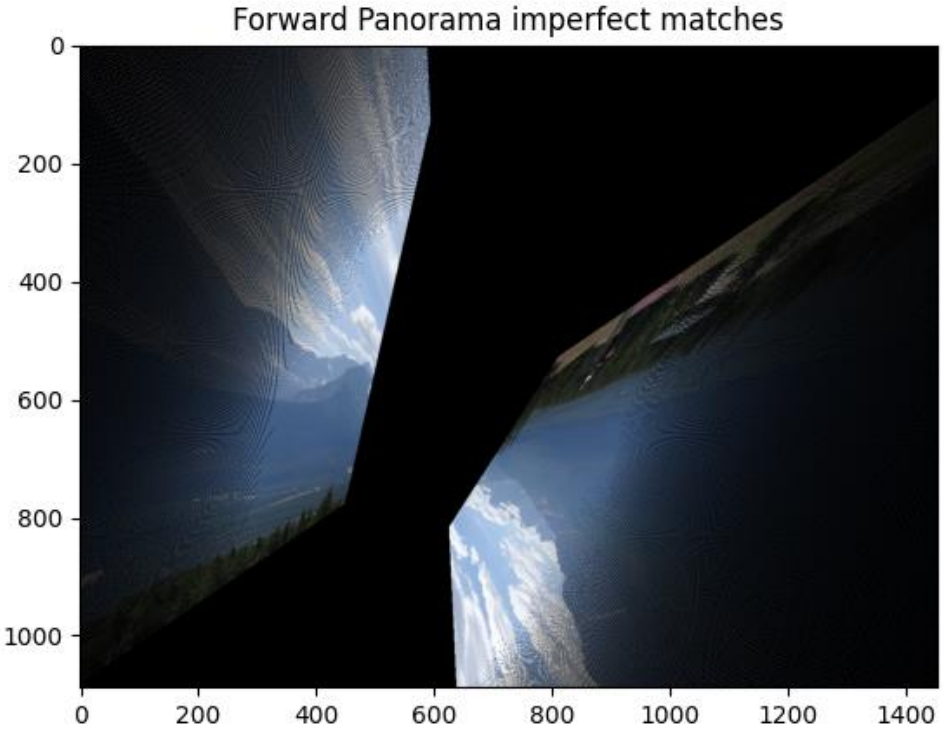
It is clear to see that the projected image is only the part of the src image which is in the coordinates domain of the target image.

5. Below is the image generated using compute_forward_homography_fast function:



The output appears to be the same as of section 4, as expected.

6. Forward mapping has 3 main problems:
   - Upon projecting a coordinate to the destination image coordinate space, the pixel coordinate may not necessarily be an integer, but rather in between the coordinates in the destination space of pixels. In such case, the forward mapping rounds up the coordinate location. This brings some distortion in the projection.
   - The added process of rounding up the coordinate location + the basic method the projection is done by, yields a situation where some coordinates in the destination image do not have a source coordinate which was projected to it. This yields in black spaces in between projection (As can be viewed in the images at section 4, 5).
   - The additional step of rounding up the coordinate location means that some coordinates are mapped to the same location in the destination image. Thus, the later projection overrides the first one.
7. Below is the image generated using compute_forward_homography_fast function, where we used the imperfect matches instead of the perfect ones:



As we can see the addition of imperfect matches (outliers), yields in a full corruption of the homography matrix found. The problems of section 6 still appear (However it is not the main problem of the transformation now).

8. The function test_homography is implemented and can be viewed in the python src files.
9. The function meet_the_model_points is implemented and can be viewed in the python src files.
10. To calculate the required number of iterations in RANSAC, we use the formula: $k = \left\lceil \frac{\log(1-p)}{\log(1-w^n)} \right\rceil + 1$ (+1 is added to handle the case where w=1). Given in the problem, $w = 0.8$ (the fraction of inliers), and from the homography requirements, we have $n = 4$ (minimum number of points needed to recover the model).

    For a confidence level of $p = 0.9$, we calculate $k = \left\lceil \frac{\log(1-0.9)}{\log(1-0.8^4)} \right\rceil + 1 = \lceil 4.37 \rceil + 1 =$ 6 iterations.

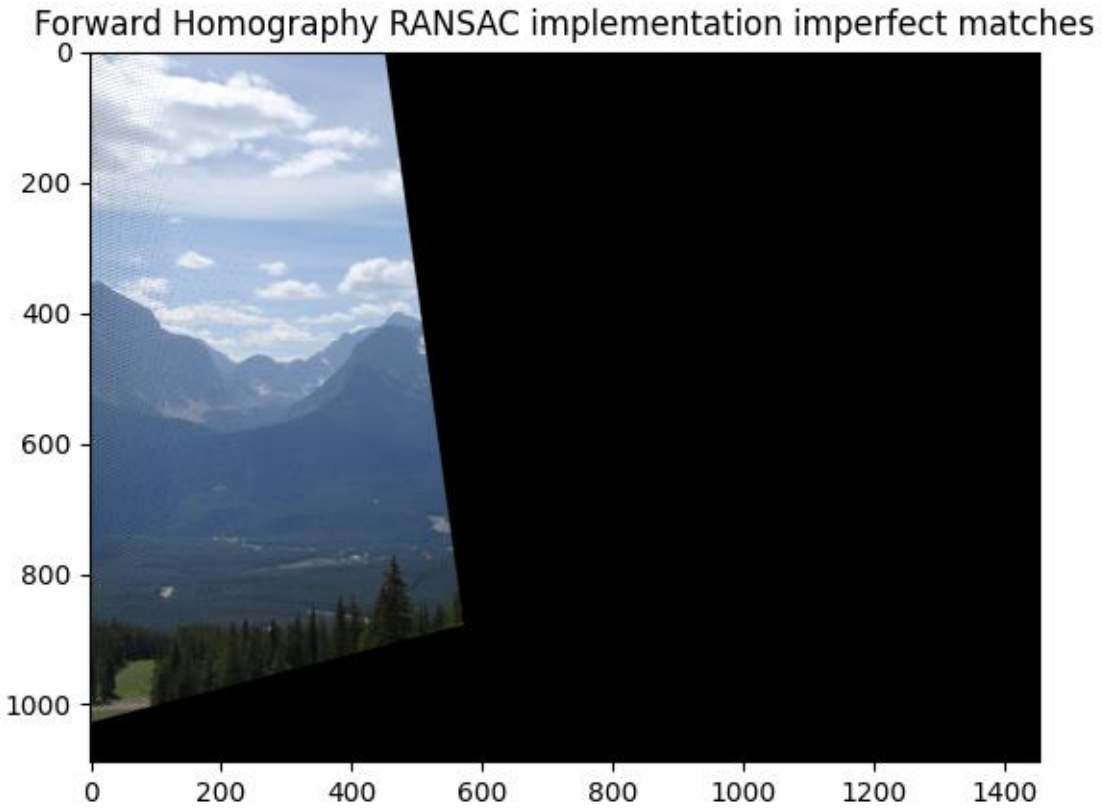
    For a confidence level of $p = 0.99$, we calculate $k = \left\lceil \frac{\log(1-0.99)}{\log(1-0.8^4)} \right\rceil + 1 = \lceil 8.74 \rceil + 1 =$ 10 iterations.

    The number of iterations to cover all options is the total possible combinations of 4 points from a set of 30 (without repetition and order not being important), which is the binomial coefficient $\binom{n}{k}$. In our case, n=30, k=4, hence we have $\binom{30}{4} = \frac{30!}{4!(30-4)!} = 27405$ possible combinations.
11. The function compute_homography is implemented and can be viewed in the python src files.

12. After loading matches.mat and using the RANSAC function implemented under section 11, we found the following matrix:

$$H = \begin{bmatrix} 1.12313787\text{e}-03 & 1.64757637\text{e}-04 & -9.99919589\text{e}-01 \\ 1.05114986\text{e}-05 & 1.05462486\text{e}-03 & -1.25618938\text{e}-02 \\ 2.96940477\text{e}-07 & 4.35704606\text{e}-08 & 7.82908438\text{e}-04 \end{bmatrix}$$

The source image after projective transform using forward mapping and homography computed by the RANSAC function:



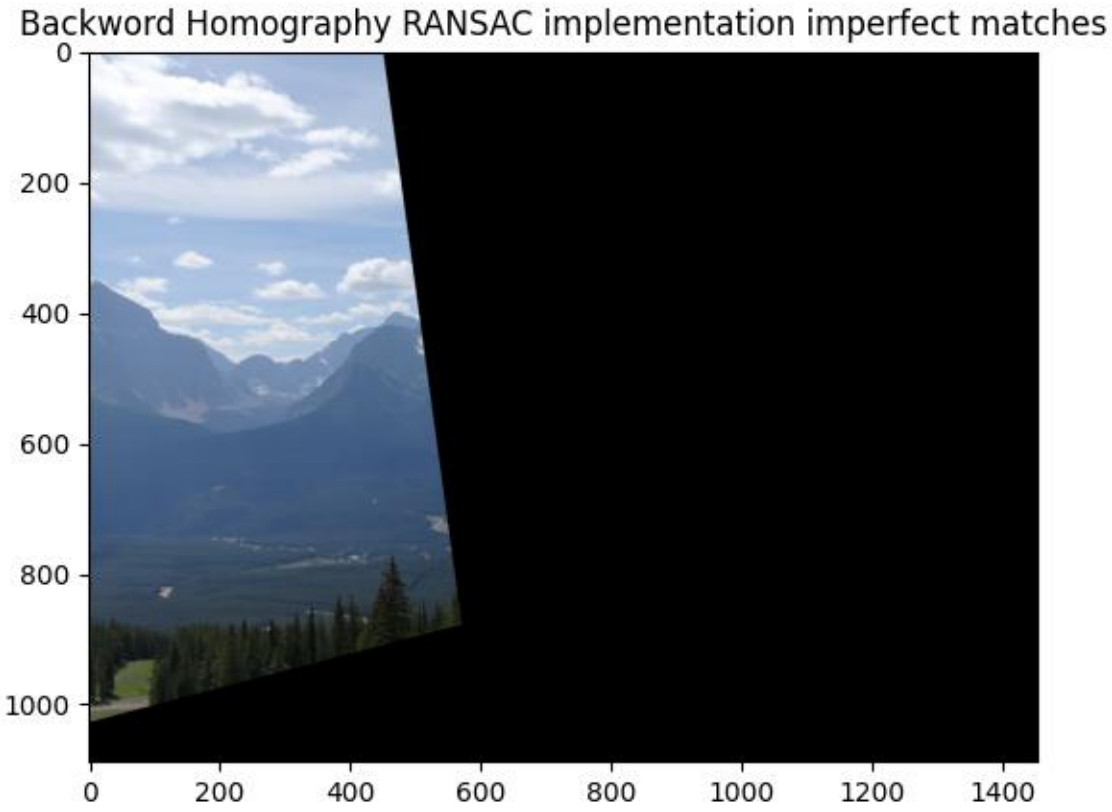
The computed RANSAC homography coefficients from imperfect matches closely resemble those obtained using the native homography on perfect matches. Consequently, the output image produced by the projective transform with forward mapping is similar to the image from Section 5.

Using the test_homography function from Section 8, we evaluated the naive homography model on the imperfect matches from Section 7. This resulted in a fit percentage of 16% and a mean squared error (MSE) of 456.310. In contrast, applying RANSAC homography on imperfect matches achieved a significantly better fit percentage of 80% and a much lower MSE of 4.6420.

These results make sense because RANSAC is designed to handle imperfect data by iteratively selecting inliers and minimizing the impact of outliers. In contrast, the naive homography approach does not account for outliers, which leads to poor performance (lower fit percentage and higher

MSE) when matches are imperfect. RANSAC's ability to isolate inliers results in a more accurate homography transformation, reflected in the significantly higher fit percentage and lower MSE.

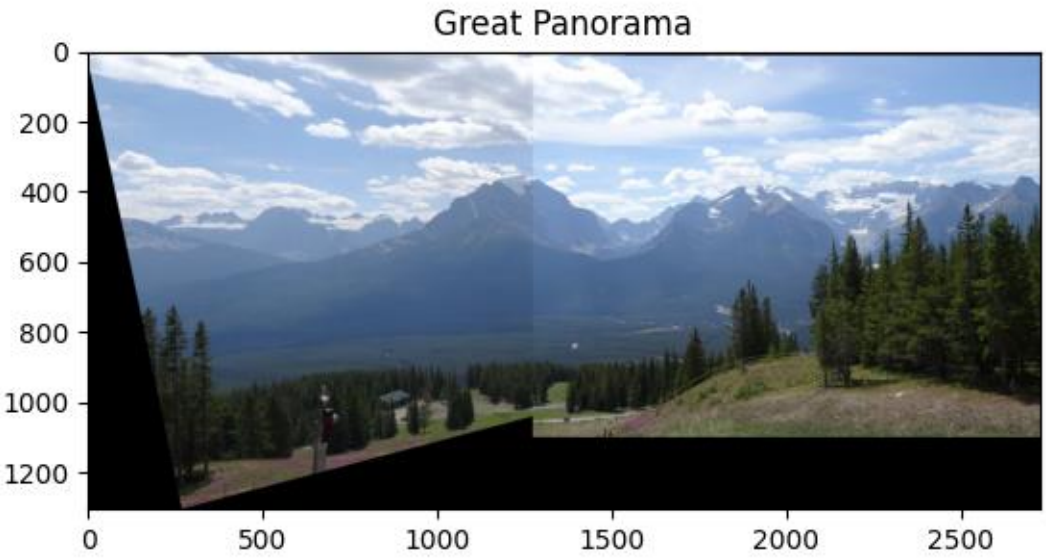
13. The source image after projective transform using backward mapping and homography computed by the RANSAC function:



As expected, the black spaces in between projection caused by the forward mapping method that appears in Section 12, have been removed.

14. The function add_translation_to_backward_homography is implemented and can be viewed in the python src files.
15. The function panorama is implemented and can be viewed in the python src files.

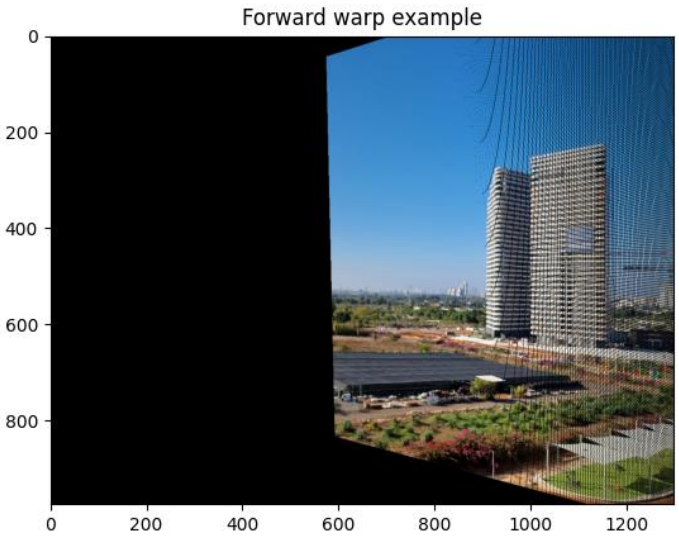
16. Below is the output panorama that produced using the function implemented in Section 15:

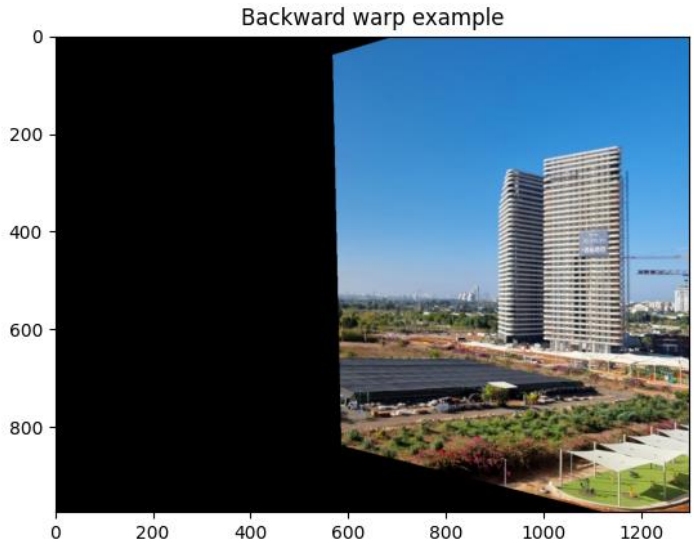


17. Input images, along with the marked matching points:

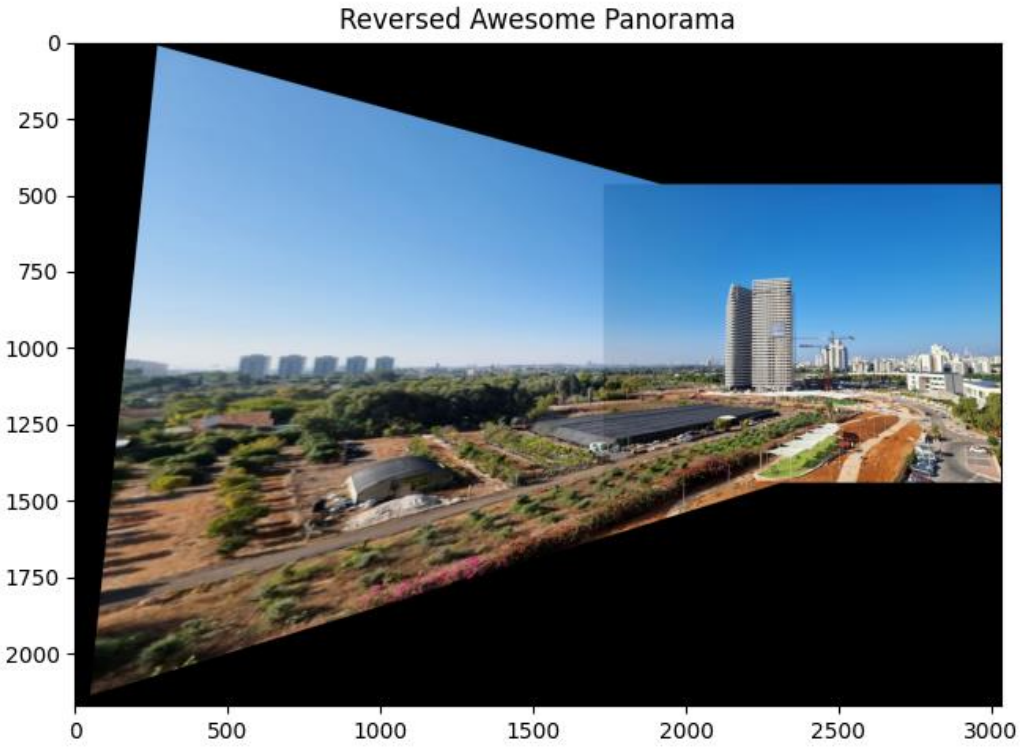
- 25 matching points
- 12% of the matching points are outliers

Forward and backward wrapping:





Output panoramas:


Reversed Awesome Panorama
0
250
500
750
1000
1250
1500
1750
2000
0
500
1000
1500
2000
2500
3000