Maxime Plumejeau
ID : 92454759 2024/06/04

Exercise : geometric transform

Geometric transform in computer vision is a fundamental process aiming to compute and describe the 2D geometric transformation between 2 images. This process follows a required step of feature extractions and feature matching between the images, and has several applications such as image reconstruction, stitching and photomontage.

In the first part, the goal of this exercise is presented, then, the experimental method is explained. The obtained results are presented in a third part. After that, a discussion is conducted about the results, and a conclusion ends this report.

I - Goal

The objective of this project is to develop a program that constructs a composite image from multiple images of the same scene. This process involves aligning and blending the images to create a seamless panorama.

For this purpose, a geometric transformation of all the initial images (except one) is required. Indeed, the idea is to transform images so that they appear to have been taken from the same point. Between 2 images of the same scene, several transformations are possible : translation, euclidean, similarity, affine and projective (as presented in the image below). The more complex the transformation, the more DoF are involved, and the more parameters are required to describe the transformation.

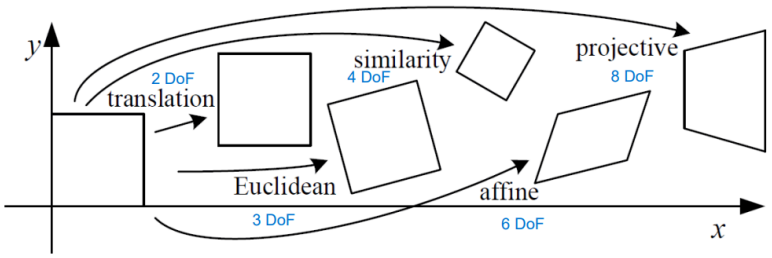


Figure 1 : basic set of 2D planar transformations

There are as many parameters characteristic of the transformation as DoF for a given transformation. It is possible to compute them from a set of matched points between both images, a set of matched points is a set of pairs of points of the same feature of the scene in both images. From this set and the characteristic Jacobian matrix of the transformation, it is possible to compute a matrix A and a vector b (as represented in the image below for the translation). Then, we obtain the parameters' vector p of the translation by solving the matrix equation : A.p = b.

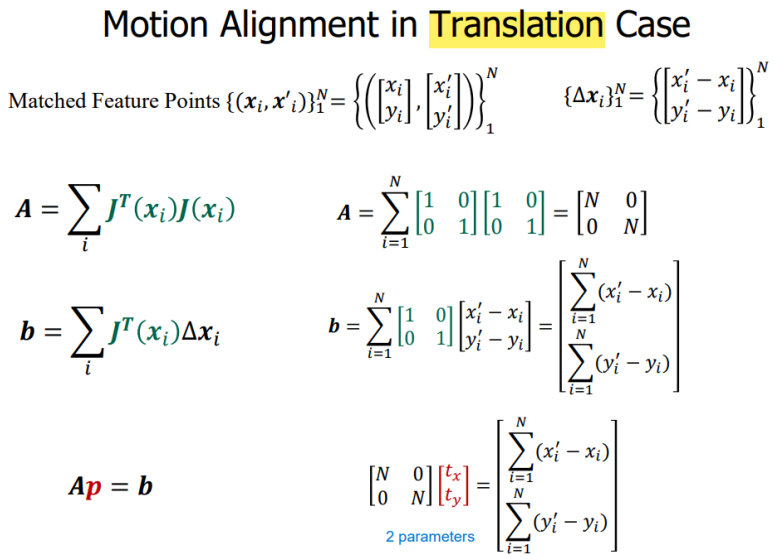


Figure 2 : computation method to obtain translation parameters

Hence, it is convenient to compute the transformation parameters between 2 images using a program script since it is mostly matrix operations. Then, the parameters can be put into a matrix, representative of the transformation. It is a 2x3 matrix for affine transformations and a 3x3 matrix for the projective transformation. However, it is possible to use a 3x3 for affine transformation for convenience by adding the third row [0 0 1]. The image below presents the transformation matrix and the Jacobian matrix of the different 2D transformations.

Jacobians of the 2D coordinate transformations

Transform	Matrix	Parameters p	Jacobian J
translation	$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix}$	(t_x, t_y)	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
Euclidean	$\begin{bmatrix} c_\theta & -s_\theta & t_x \\ s_\theta & c_\theta & t_y \end{bmatrix}$	(t_x, t_y, θ)	$\begin{bmatrix} 1 & 0 & -s_\theta x - c_\theta y \\ 0 & 1 & c_\theta x - s_\theta y \end{bmatrix}$
similarity	$\begin{bmatrix} 1+a & -b & t_x \\ b & 1+a & t_y \end{bmatrix}$	(t_x, t_y, a, b)	$\begin{bmatrix} 1 & 0 & x & -y \\ 0 & 1 & y & x \end{bmatrix}$
affine	$\begin{bmatrix} 1+a_{00} & a_{01} & t_x \\ a_{10} & 1+a_{11} & t_y \end{bmatrix}$	$(t_x, t_y, a_{00}, a_{01}, a_{10}, a_{11})$	$\begin{bmatrix} 1 & 0 & x & y & 0 & 0 \\ 0 & 1 & 0 & 0 & x & y \end{bmatrix}$
projective	$\begin{bmatrix} 1+h_{00} & h_{01} & h_{02} \\ h_{10} & 1+h_{11} & h_{12} \\ h_{20} & h_{21} & 1 \end{bmatrix}$	$(h_{00}, h_{01}, \ldots, h_{21})$	(see Section 8.1.3)

re-parameterized the motions so that they are identity $\boldsymbol{I}$ for $\boldsymbol{p} = \boldsymbol{0}$.

Figure 3 : transformation and Jacobian matrices for the different transformations

A noticeable point is that the euclidean is different from the others. Indeed, one of the transformation parameters, θ, is included in the Jacobian matrix. Hence, as for the projective transformation, it is not possible to compute the parameters with a linear method. An effective way to avoid this problem is to compute the similarity parameters, and then estimate the euclidean parameter θ with : $\theta = \tan^{-1}(a/b)$.

II - Method

In order to construct an image resulting from the appropriate blending of 2 images of the same scene, several steps are required :

- feature extraction and matching between both images
- computation of transformation matrix between the reference image and the query image
- geometric transformation of the query image
- blending the reference image and the transformed query image on a canvas

Let's detail the different steps.

a) Feature extraction and matching

This first step of the script is performed by the *matched_keypoints* function (presented in the images below), using the orb library, efficient for fastly realizing feature extractions and feature matching between images.

```python
# find keypoints of 2 input grey scale images, perform matches between both images and return the strongest matches into an array
def match_keypoints(gray1, gray2, ratio, res_path):

    # initiate ORB detector
    orb = cv.ORB_create()

    # kp --> Keypoints : array containing the feature points of the grayscale image
    # des --> Descriptor : array of size [nb keypoints]x128 containing informations about feature points

    # find the keypoints and compute the descriptors with ORB
    kp1 = orb.detect(gray1, None)
    kp2 = orb.detect(gray2, None)
    kp1, des1 = orb.compute(gray1, kp1)
    kp2, des2 = orb.compute(gray2, kp2)

    print('nb keypoints img1 : ' + str(len(kp1)))

    # draw keypoints and save the result image
    img1_kp = gray1
    img1_kp = cv.drawKeypoints(gray1, kp1, img1_kp)
    cv.imwrite(res_path + 'orb_keypoints.jpg', img1_kp)
```

```python
    # match keypoints between both images using brute-force
    bf = cv.BFMatcher()
    matches = bf.knnMatch(des1, des2, k=2)

    # apply ratio test to select only the strongest matches
    good_matches = []
    for m,n in matches:
        if m.distance < ratio * n.distance:
            good_matches.append([m])

    # draw matches and save the result image
    img_matches = cv.drawMatchesKnn(gray1, kp1, gray2, kp2, good_matches, None, flags=cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
    cv.imwrite(res_path + 'orb_matches.jpg', img_matches)

    # extract the 4 best matches from good_matches (those with the smallest distance)
    if len(good_matches) < 4:
        print("less than 4 matches between images")
    else:
        sorted_good_matches = sorted(good_matches, key=lambda x: x[0].distance)
        best_4_matches = sorted_good_matches[:4]
```

```
    # draw matches and save the result image
    img_best_matches = cv.drawMatchesKnn(gray1, kp1, gray2, kp2, best_4_matches, None, flags=cv.DrawMatchesFlags_NOT_DRAW_SINGLE_P
    cv.imwrite(res_path + 'orb_best_matches.jpg', img_best_matches)

    # construct an array of pair of matched points
    matched_points = []
    for m in good_matches:
        pt1 = kp1[m[0].queryIdx].pt
        pt2 = kp2[m[0].trainIdx].pt
        matched_points.append([pt1, pt2])

    # construct an array of pair of the 4 best matched points
    best_4_matched_points = []
    for m in best_4_matches:
        pt1 = kp1[m[0].queryIdx].pt
        pt2 = kp2[m[0].trainIdx].pt
        best_4_matched_points.append([pt1, pt2])

    return matched_points, best_4_matched_points
```

Figure 4 : match_keypoints function

The *match_keypoints* function takes 2 gray images as inputs and returns a set of matched points as output. After performing the feature detection and matching using the orb library, a ratio-test is applied in order to improve the quality of the matched points in the *good_matches* variable. From this, the *matched_points* array is constructed to store only the pairs of matched points between images. Besides, another array, *best_4_matched_points* is constructed and returned by the function in order to be used by OpenCV functions. It stores only the 4 best pairs of matched points, i.e. those with the smallest “distance” in terms of feature matching.

b) Computation of transformation matrix

Once the feature matching has been performed, a set of pairs of matched points between both images is available. From this set, it is possible to compute the transformation matrix for the translation, euclidean, similarity and affine using the method presented in the *Goal* section. The following images show the different functions of the script implemented to compute the transformation parameters and matrixes.

```
# compute the translation parameters between 2 images using the matched keypoints and return the 3x3 transformation matrix
def compute_translation_matrix(matched_points):

    # dimension of matrix A (biggest dim of J)
    n = 2

    # compute matrix A and vector b
    A, b = compute_A_b(matched_points, J_translation, n)

    # compute translation 2x1 vector p
    p = solve(A, b)
    print('translation vector :')
    print(p)

    # 3x3 transformation matrix
    M = np.array([[1, 0, p[0][0]],
                  [0, 1, p[1][0]],
                  [0, 0, 1      ]])

    return M
```

Figure 5 : compute_translation_matrix function

```python
# compute the euclidean parameters between 2 images using the matched keypoints and return the 3x3 transformation matrix
def compute_euclidean_matrix(matched_points):

    # for the estimation of the euclidean transformation, we first compute a similarity transformation since it is a non-linear problem

    # dimension of matrix A (biggest dim of J)
    ns = 4

    # compute matrix A and vector b
    As, bs = compute_A_b(matched_points, J_similarity, ns)

    # compute similarity 4x1 vector p
    ps = solve(As, bs)

    # compute euclidean 3x1 vector p from the similarity vector
    p = np.zeros([3, 1])
    p[0][0] = ps[0][0]
    p[1][0] = ps[1][0]
    p[2][0] = np.arctan(ps[3][0]/ps[2][0])
    print('euclidean vector :')
    print(p)

    # 3x3 transformation matrix
    M = np.array([[np.cos(p[2][0]), -np.sin(p[2][0]), p[0][0]],
                  [np.sin(p[2][0]), np.cos(p[2][0]) , p[1][0]],
                  [0              , 0               , 1      ]])

    return M
```

Figure 6 : compute_euclidean_matrix function

```python
# compute the similarity parameters between 2 images using the matched keypoints and return the 3x3 transformation matrix
def compute_similarity_matrix(matched_points):

    # dimension of matrix A (biggest dim of J)
    n = 4

    # compute matrix A and vector b
    A, b = compute_A_b(matched_points, J_similarity, n)

    # compute similarity 4x1 vector p
    p = solve(A, b)
    print('similarity vector :')
    print(p)

    # 3x3 transformation matrix
    M = np.array([[1+p[2][0], -p[3][0] , p[0][0]],
                  [p[3][0]  , 1+p[2][0], p[1][0]],
                  [0        , 0        , 1      ]])

    return M
```

Figure 7 : compute_similarity_matrix function

```python
# compute the affine parameters between 2 images using the matched keypoints and return the 3x3 transformation matrix
def compute_affine_matrix(matched_points):

    # dimension of matrix A (biggest dim of J)
    n = 6

    # compute matrix A and vector b
    A, b = compute_A_b(matched_points, J_affine, n)

    # compute affine 6x1 vector p
    p = solve(A, b)
    print('affine vector :')
    print(p)

    # 3x3 transformation matrix
    M = np.array([[1+p[2][0], p[3][0]  , p[0][0]],
                  [p[4][0]  , 1+p[5][0], p[1][0]],
                  [0        , 0        , 1      ]])

    return M
```

Figure 8 : compute_affine_matrix function

```python
# compute the nxn matrix A and the nx1 vector b from the matched points between 2 images and the Jacobian matrix
def compute_A_b(matched_points, J_def, n):

    A = np.zeros([n, n])
    b = np.zeros([n, 1])

    for m in matched_points:

        x1 = m[0][0]
        y1 = m[0][1]
        x2 = m[1][0]
        y2 = m[1][1]
        delta = [[x2 - x1],
                 [y2 - y1]]

        # Jacobian matrix
        J = J_def(x1, y1)
        Jt = J.transpose()

        A = A + Jt.dot(J)
        b = b + Jt.dot(delta)

    return A, b
```

Figure 9 : compute_A_b function

```python
def J_translation(x, y):
    J = np.array([[1, 0],
                  [0, 1]])
    return J

def J_similarity(x, y):
    J = np.array([[1, 0, x, -y],
                  [0, 1, y, x]])
    return J

def J_affine(x, y):
    J = np.array([[1, 0, x, y, 0, 0],
                  [0, 1, 0, 0, x, y]])
    return J
```

Figure 10 : implementation of Jacobian matrices

As explained in the *Goal* section, it is not possible to compute the projective transformation's matrix with a linear algorithm. However, the OpenCV function *getPerspectiveTransform* returns this matrix from the 4 pairs of matched points given as input. This solution has been implemented in the script in order to have all the transformation available.

```python
def get_homography_matrix(matched_points):

    pts1 = []
    pts2 = []

    # creation of a list of 4 points for each image
    for m in matched_points:
        pts1.append(m[0])
        pts2.append(m[1])

    print(pts1)
    print(pts2)

    # conversion into np array
    pts1_np = np.array(pts1, dtype=np.float32)
    pts2_np = np.array(pts2, dtype=np.float32)

    # 3x3 transformation matrix
    M = cv.getPerspectiveTransform(pts1_np, pts2_np)

    return M
```

Figure 11 : get_homography_matrix function

c) Geometric transformation

It is possible to transform the query image using the obtained transformation matrix and the *warpPerspective* function. This is performed in the *transform* function of the script, presented in the image below. Hence, the resulting image appears to have been taken from the same point as the reference image.

```python
# transform an image using the 3x3 transformation matrix M and save the result image
def transform(gray, M, res_path):

    img_width = gray.shape[1]
    img_height = gray.shape[0]

    img_transformed = cv.warpPerspective(gray, M, (img_width, img_height))

    print(img_transformed.shape[:2])

    cv.imwrite(res_path + 'img_query_transform.jpg', img_transformed)
```

Figure 12 : transform function

d) Blending

The *transform* function only modifies the query image, but the goal is to blend the reference image and the transformed query image, in order to create a panograph. This is performed by the *blend_image* function of the script.

```python
# stitch the reference image and the transformed query image on a single image
def blend_images(img_query, img_ref, M):

    # compute canva dimension
    canvas_size, translation = calculate_canvas_size(img_query, img_ref, M)
    tx, ty = translation

    # multiplication of transformation matrix and translation matrix
    translation_matrix = np.array([[1, 0, tx], [0, 1, ty], [0, 0, 1]])
    M_translation = np.dot(translation_matrix, M)

    # canvas creation
    canvas = np.zeros((canvas_size[1], canvas_size[0]), dtype=np.uint8)

    # put ref image on the canvas
    canvas[ty:ty+img_ref.shape[0], tx:tx+img_ref.shape[1]] = img_ref

    # transformation of the query image
    img_query_transformed = cv.warpPerspective(img_query, M_translation, canvas_size)

    # blend images on the canvas
    canvas = cv.addWeighted(canvas, 1, img_query_transformed, 1, 0)

    return canvas
```

Figure 13 : blend_images function

The *blend_images* function starts by computing the canvas dimension using the *calculate_canvas_size* function. Then, the transformation matrix is multiplied by the translation matrix computed in the *calculate_canvas_size*, the purpose of this translation is to be sure that every point's coordinates are positive. After that, a canvas of the right dimension is created and the reference image and the transformed query image are successfully added to create a stitched image.

```python
# compute the canvas dimension required to contain the reference and the transformed query images
def calculate_canvas_size(img_query, img_ref, M):

    # get images dimensions
    h1, w1 = img_query.shape[:2]
    h2, w2 = img_ref.shape[:2]

    # get corners' positions for both images
    corners_img_query = np.float32([[0, 0], [0, h1], [w1, h1], [w1, 0]]).reshape(-1, 1, 2)
    corners_img_ref = np.float32([[0, 0], [0, h2], [w2, h2], [w2, 0]]).reshape(-1, 1, 2)

    # get corners' positions of the transformed query image
    transformed_corners_img_query = cv.perspectiveTransform(corners_img_query, M)

    # combine corners of both images
    all_corners = np.vstack((corners_img_ref, transformed_corners_img_query))

    # find limits of the canvas
    [xmin, ymin] = np.int32(all_corners.min(axis=0).ravel() - 0.5)
    [xmax, ymax] = np.int32(all_corners.max(axis=0).ravel() + 0.5)
    translation = [-xmin, -ymin]

    # canvas dimensions
    canvas_size = (xmax - xmin, ymax - ymin)

    return canvas_size, translation
```

Figure 14 : calculate_canvas_size function

The *calculate_canvas_size* function computes the dimension of the canvas required to stitch the images. For this purpose, it calculates the transformed coordinates of the corners of the query image and computes the translation assuring the positivity of the corners.

III - Results

This section presents the results of the different parts of the script, following the order of the functions presented previously. The initial images used for the results' presentation are almost separated only by a translation of the camera, in order to make every transformation relevant between them. Hence, it is possible to compare the accuracy of the different transformations, which is performed in the *Discussion* section.

Figure 15 : initial reference image (left) and query image (right)

Figure 16 : feature detection in the query image

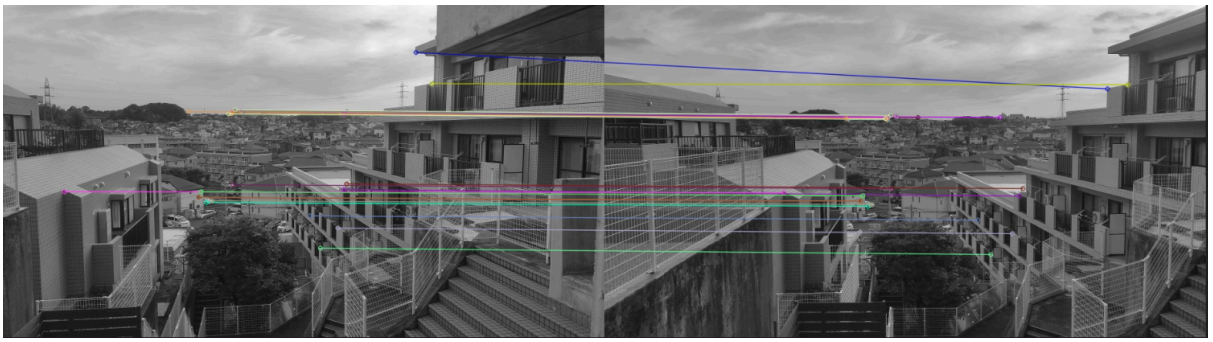


Figure 17 : feature matching between the reference image and the query image

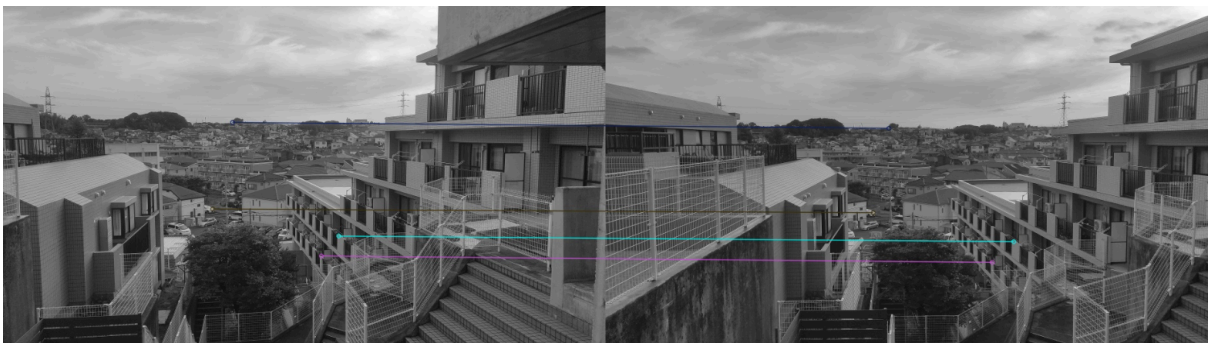


Figure 18 : 4 best feature matching between the reference image and the query image

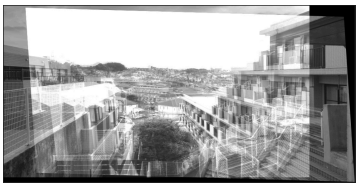
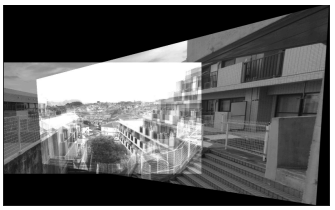
	Transformation matrix	Transformed query image	Stitched image
Translation	translation matrix : [[1. 0. 97.21451173] [0. 1. 8.97631469] [0. 0. 1.]]		
Similarity	similarity matrix : [[1.00910425e+00 -7.69020727e-03 9.54211708e+01] [7.69020727e-03 1.00910425e+00 3.82438215e+00] [0.00000000e+00 0.00000000e+00 1.00000000e+00]]		
Affine	affine matrix : [[1.04137586e+00 6.27207440e-02 6.62926748e+01] [2.96760428e-02 9.62063763e-01 6.13124898e+00] [0.00000000e+00 0.00000000e+00 1.00000000e+00]]		
Homography	homography matrix from func : [[5.09131526e-01 6.51604737e-02 1.52605360e+02] [-2.07580870e-01 7.93465473e-01 7.95727743e+01] [-6.37221667e-04 7.77257117e-05 1.00000000e+00]]		

Figure 19 : stitching comparison between the different transformation

It is also possible to stitch 2 images which are separated by a rotation using the script, as shown in figure 20 and 21.

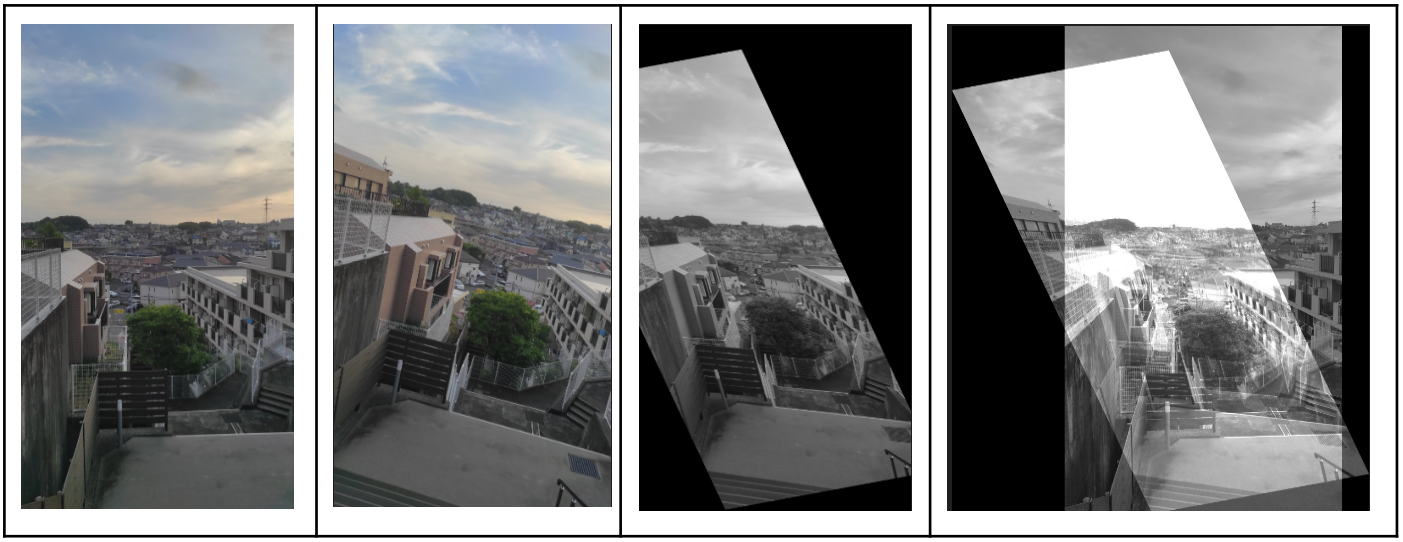


Figure 20 : stitching between 2 images using affine transformation

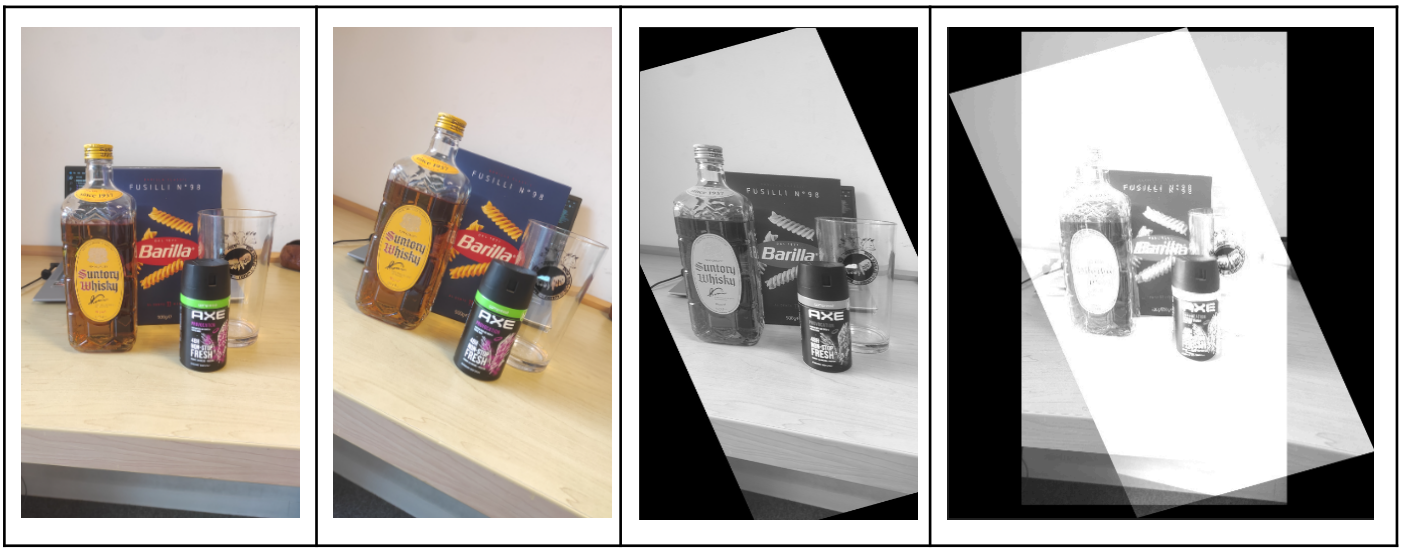


Figure 21 : stitching between 2 images using affine transformation

The different resulting images come from the developed script. Hence, it is possible to conclude that all the steps are well performed. Indeed, the final stitching works for every transformation.

An noticeable limit of the script is the homography transformation. Indeed, it is performed using the *getPerspectiveTranform* function, which takes 4 pairs of points as input and returns a transformation matrix. However, a matching error in these 4 pairs have huge consequences on the output matrix. Therefore, in several cases, the transformed query image was completely wrong because of such an error. The script used the selection of the theoretically 4 best matches (using the distances) but another method may be better.

IV - Discussion

Different geometric transformations are implemented in the script. Thus, it is possible to compare them through the exemple proposed in the *Result* section. Indeed, the query image transformation and the stitching of 2 images has been performed for the translation, the similarity transform, the affine transform and the homography. The latter's matrix has been computed through an OpenCV function while the others have been calculated using linear algebra and the Jacobian matrix.

On the figure 19, containing the different resulting images, we can see that the more complex the transformation (with the more DoF and parameters), the more accurate is the transformed query image and the stitching image. Indeed, the translation stitching looks like a draft, the different buildings are not well superposed while the homography stitching is more accurate, it is possible to clearly see that the center of the reference image has been well superposed. However, about the borders of the reference images, the superposition is not perfect for all the transformations. This is probably a consequence of the fact that the matched feature points are mainly located in the center of both images.

Hence, it is possible to conclude that more complex transformations create better stitching results. However, it becomes more complicated to compute the matrix when the complexity increases, with a particular difficulty for the homography, as this transformation required to use non-linear algebra. Moreover, the functions provided by OpenCV required a great accuracy in the input data, such as *getPerspectiveTranform* which takes only 4 pairs of matched points (as precised in the *Result* section) and is therefore non-robust to error during the feature matching process.

V - Conclusion

To conclude this exercise, I have successfully developed a python script that stitches 2 images of the same scene in order to create a panograph. For this purpose, different steps such as feature detection, feature matching, transformation matrix computation, and blending are implemented using OpenCV library and linear algebra concepts.