Analyse traitement d'image et vision 3D CM

Assignment #1

LE Thi Hoa - 12310380
TRAN Hai Linh - 12310487
Sophin CHHENG - 223419025
Saad Belaouad

1. Introduction

This report consists of three sections equivalent to three tasks in assignment 1, including: template matching, image transformations, panoramic image stitching.

We use the *OpenCV* library for reading and storing images, the *Matplotlib* library for displaying images, and the *NumPy* library for computations. We use *Python* for implementation

2. Template Matching

2.1. Introduction:

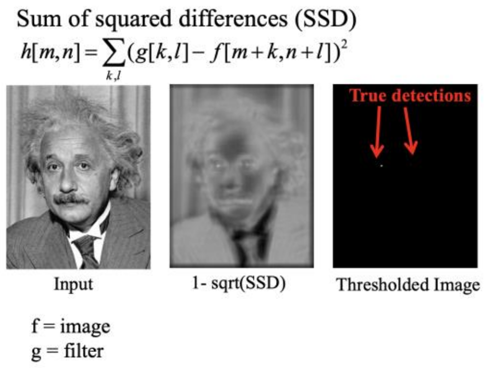
The problem is about implementing a template matching technique using the sum of squared distances (SSD) metric to find a template pattern (Waldo's face) in a cluttered scene (amusement park image). The provided template images are the clear template of Waldo's face (template1.png) and a noisy version of the template (template2.png).

2.2. Methodology and Procedure:

- Load the Images:
    - Load the scene image (where_is_waldo.jpg) as the reference image and load the clear template (template1.png) and the noisy template (template2.png).

```
Show image
```

- Iterate Over Pixel Locations and Calculate SSD
  - Iterate over all pixel locations in the reference image to compare the local patch with the template image using the SSD metric.
  - At each pixel, calculate the squared difference between the pixel value of the local patch and the corresponding template.
  - Sum these squared differences (sum of square differences) to compute SSD for each pixel.
  - We have the SSD calculation formula as follows:



Sum of squared differences (SSD)

$$h[m,n] = \sum_{k,l} (g[k,l] - f[m+k, n+l])^2$$

f = image
g = filter

- Display SSD for the Whole Image:
  - Display the SSD values for the entire reference image, showing the differences between the local patches and the template.

```
Display SSD for the Whole Image::
[[365083. 363007. 364775. ... 366630. 368696. 367141.]
 [369842. 363411. 362711. ... 368173. 371514. 357173.]
 [364263. 357207. 362505. ... 361299. 369628. 366648.]
 ...
 [325574. 337094. 334454. ... 365883. 362922. 366755.]
 [327210. 328456. 329839. ... 364343. 365260. 369687.]
 [327557. 330552. 334805. ... 365076. 360568. 360719.]]
```

- Find Minimum SSD Location:
    - Find the location (x, y) coordinate where the SSD is minimum. This represents the location where the template (Waldo's face) is the closest match to the reference image.

```
Min value of ssd:  0.0
Location of ssd min value:  [  74 1258]
Display result
```

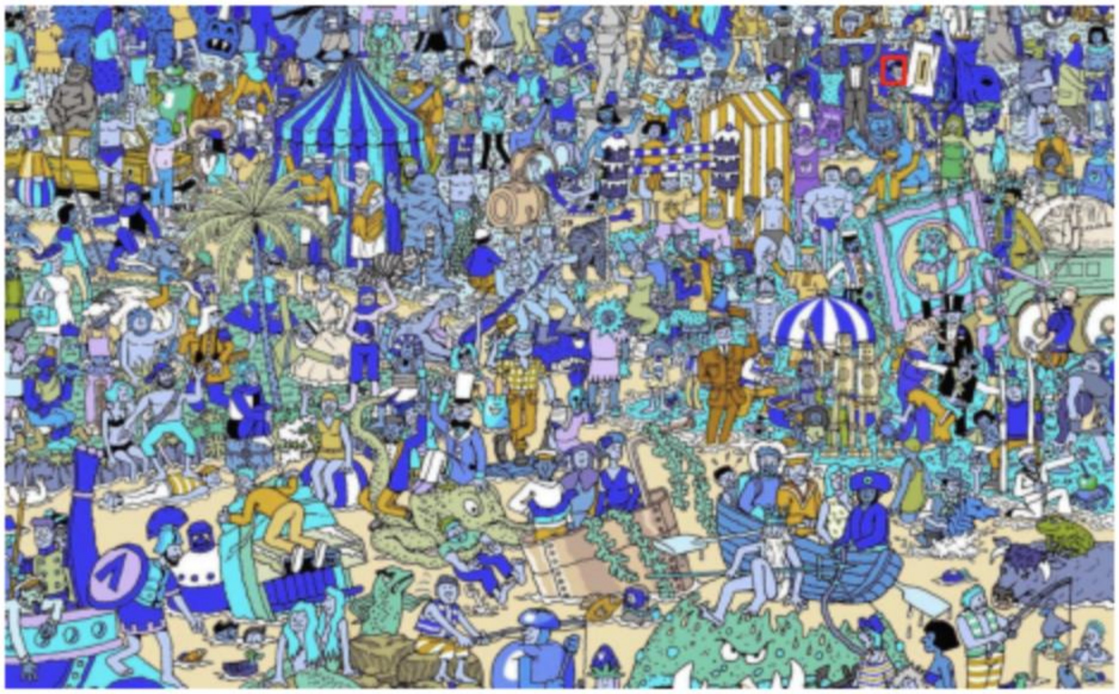


Image displaying the Waldo search results

❖ Repeat the above process with the noisy template (template2.png). Do you find the same location? If not, what metric (refer to class lectures) should be beneficial for you?
- When replacing with a noisy image (template2), we still find the location as in the good image

Link code:
https://github.com/LeHoa98ptit/ComputerVision_Lyon_1/blob/main/Assignment_1/A_Template_Matching/Template_Matching_Waldo_template_1.ipynb

3. Image Transformations

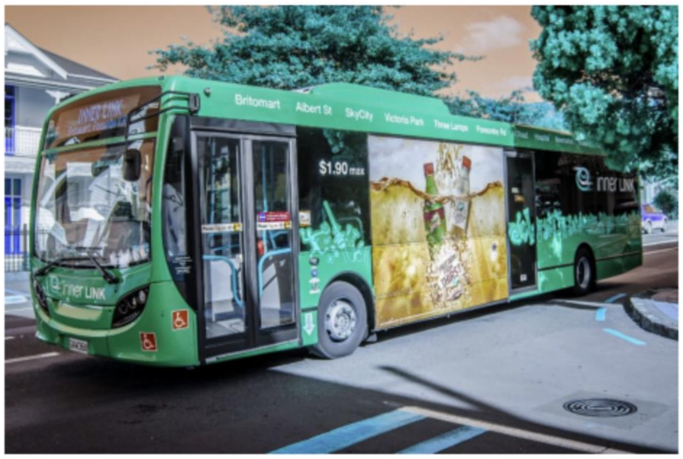
3.1. Introduction

This problem focuses on using homographic transformations to manipulate images. We were provided with an image of a bus containing a Sprite advertisement, and the objective is to replace this advertisement with a Simpson's poster.

3.2. Methodology and Procedure

- Load the Images:
  - Load the bus image (bus.jpeg) containing the Sprite advertisement.
  - Load the Simpsons poster image (simpsons.jpeg) that we will replace the advertisement with.

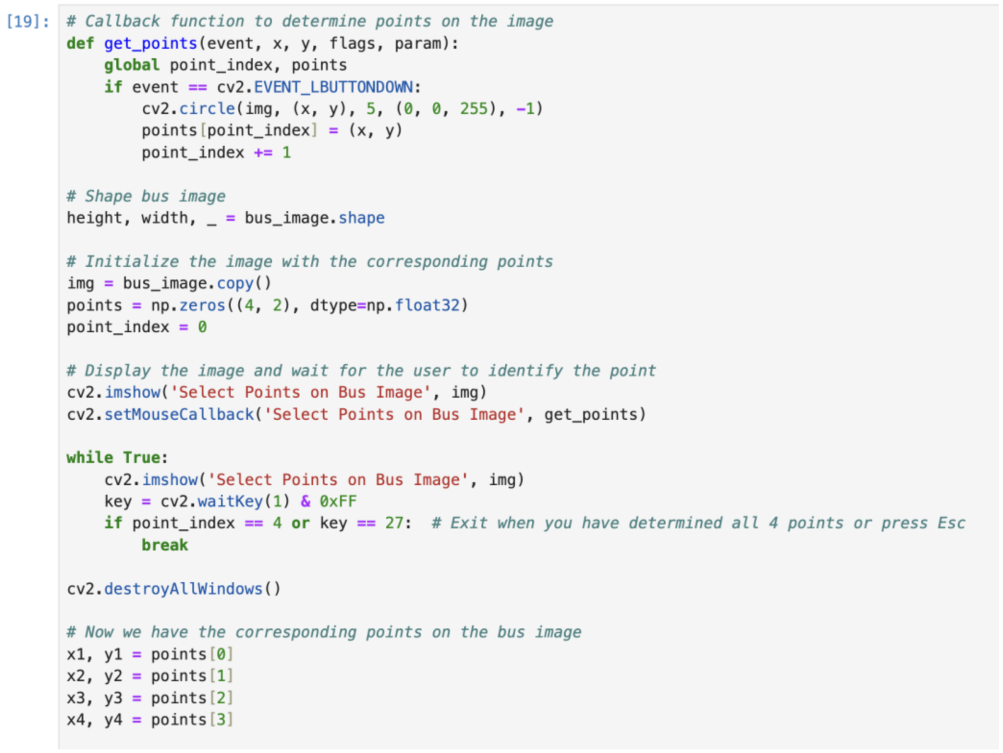
```
[21]: # Names of bus images and Simpsons images
      bus_image_path = '../data_input/bus.jpeg'
      simpsons_image_path = '../data_input/simpsons.jpeg'

      bus_image = cv2.imread('../data_input/bus.jpeg')
      simpsons_image = cv2.imread('../data_input/simpsons.jpeg')

      # Display image
      imshow(bus_image)
      imshow(simpsons_image)
```



THE SIMPSONS

- Point Correspondences:
  - Manually select and match four corresponding points between the two images. These points should represent the corners of the Sprite advertisement on the bus image and the corresponding corners on the Simpsons poster.
  - We have created a function to manually extract the coordinates of corresponding points. This code allows the user to click on the bus image

to select four points manually, which are stored in the points array for later use in estimating a homographic transformation. We can exit the selection process by either selecting all four points or pressing the Esc key.



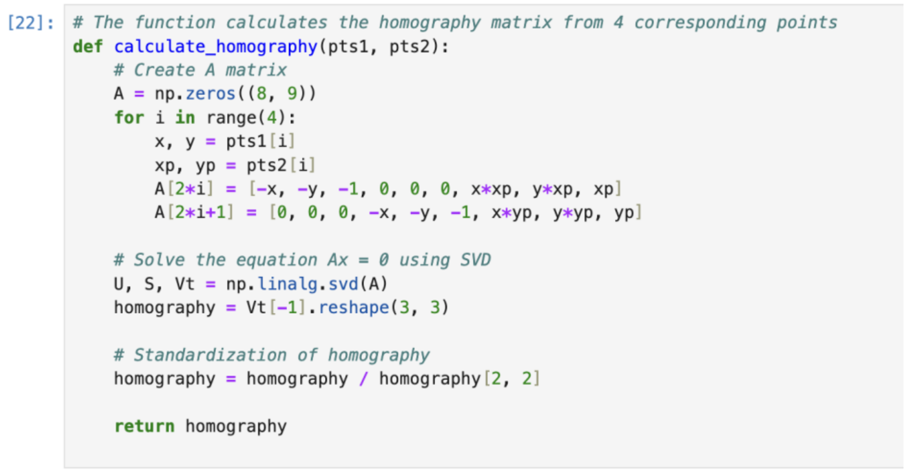
```
[19]:
# Callback function to determine points on the image
def get_points(event, x, y, flags, param):
    global point_index, points
    if event == cv2.EVENT_LBUTTONDOWN:
        cv2.circle(img, (x, y), 5, (0, 0, 255), -1)
        points[point_index] = (x, y)
        point_index += 1

# Shape bus image
height, width, _ = bus_image.shape

# Initialize the image with the corresponding points
img = bus_image.copy()
points = np.zeros((4, 2), dtype=np.float32)
point_index = 0

# Display the image and wait for the user to identify the point
cv2.imshow('Select Points on Bus Image', img)
cv2.setMouseCallback('Select Points on Bus Image', get_points)

while True:
    cv2.imshow('Select Points on Bus Image', img)
    key = cv2.waitKey(1) & 0xFF
    if point_index == 4 or key == 27:  # Exit when you have determined all 4 points or press Esc
        break

cv2.destroyAllWindows()

# Now we have the corresponding points on the bus image
x1, y1 = points[0]
x2, y2 = points[1]
x3, y3 = points[2]
x4, y4 = points[3]
```

- Build a homographic estimate function
  - The calculate_homography function is used to compute the homography matrix from four corresponding points between two images, in this case, from the image of the bus to the image of the Simpsons poster. Homography is a linear transformation between two images and is commonly used to perform image stitching or appropriate image transformation
  - To calculate the homography matrix, we will use the equation:

$$s \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

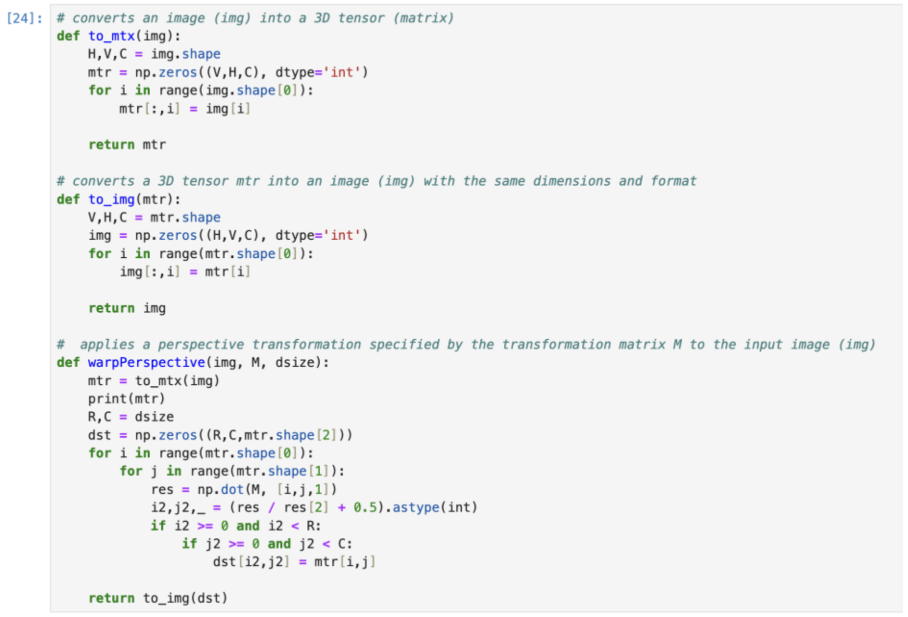
  - In there:
    + (x,y): the coordinates on the Simpsons image
    + (x', y'): the corresponding coordinates on the Sprite advertisement on the bus
    + H: the homography matrix
    + S: a constant for normalization

- The homography matrix H has the form: $H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}$



```
[22]: # The function calculates the homography matrix from 4 corresponding points
def calculate_homography(pts1, pts2):
    # Create A matrix
    A = np.zeros((8, 9))
    for i in range(4):
        x, y = pts1[i]
        xp, yp = pts2[i]
        A[2*i] = [-x, -y, -1, 0, 0, 0, x*xp, y*xp, xp]
        A[2*i+1] = [0, 0, 0, -x, -y, -1, x*yp, y*yp, yp]

    # Solve the equation Ax = 0 using SVD
    U, S, Vt = np.linalg.svd(A)
    homography = Vt[-1].reshape(3, 3)

    # Standardization of homography
    homography = homography / homography[2, 2]

    return homography
```

- Homographic Transformation:
    - Once we have the homography matrix (obtained through the estimation process), we can use it to perform image transformations.
    - The homographic transformation process involves applying the computed homography matrix to all points in the original image to create a new image (the destination image).
    - The function first converts the input image into a tensor, applies the transformation to each pixel, and creates the destination image as a tensor. Finally, it converts the destination tensor back into an image and returns it. This function is responsible for performing the actual homographic image transformation.



```
[24]: # converts an image (img) into a 3D tensor (matrix)
def to_mtx(img):
    H,V,C = img.shape
    mtr = np.zeros((V,H,C), dtype='int')
    for i in range(img.shape[0]):
        mtr[:,i] = img[i]

    return mtr

# converts a 3D tensor mtr into an image (img) with the same dimensions and format
def to_img(mtr):
    V,H,C = mtr.shape
    img = np.zeros((H,V,C), dtype='int')
    for i in range(mtr.shape[0]):
        img[:,i] = mtr[i]

    return img

#  applies a perspective transformation specified by the transformation matrix M to the input image (img)
def warpPerspective(img, M, dsize):
    mtr = to_mtx(img)
    print(mtr)
    R,C = dsize
    dst = np.zeros((R,C,mtr.shape[2]))
    for i in range(mtr.shape[0]):
        for j in range(mtr.shape[1]):
            res = np.dot(M, [i,j,1])
            i2,j2,_ = (res / res[2] + 0.5).astype(int)
            if i2 >= 0 and i2 < R:
                if j2 >= 0 and j2 < C:
                    dst[i2,j2] = mtr[i,j]

    return to_img(dst)
```

- Apply the Transformation to Simpsons Image:
    - We get results like the image below:

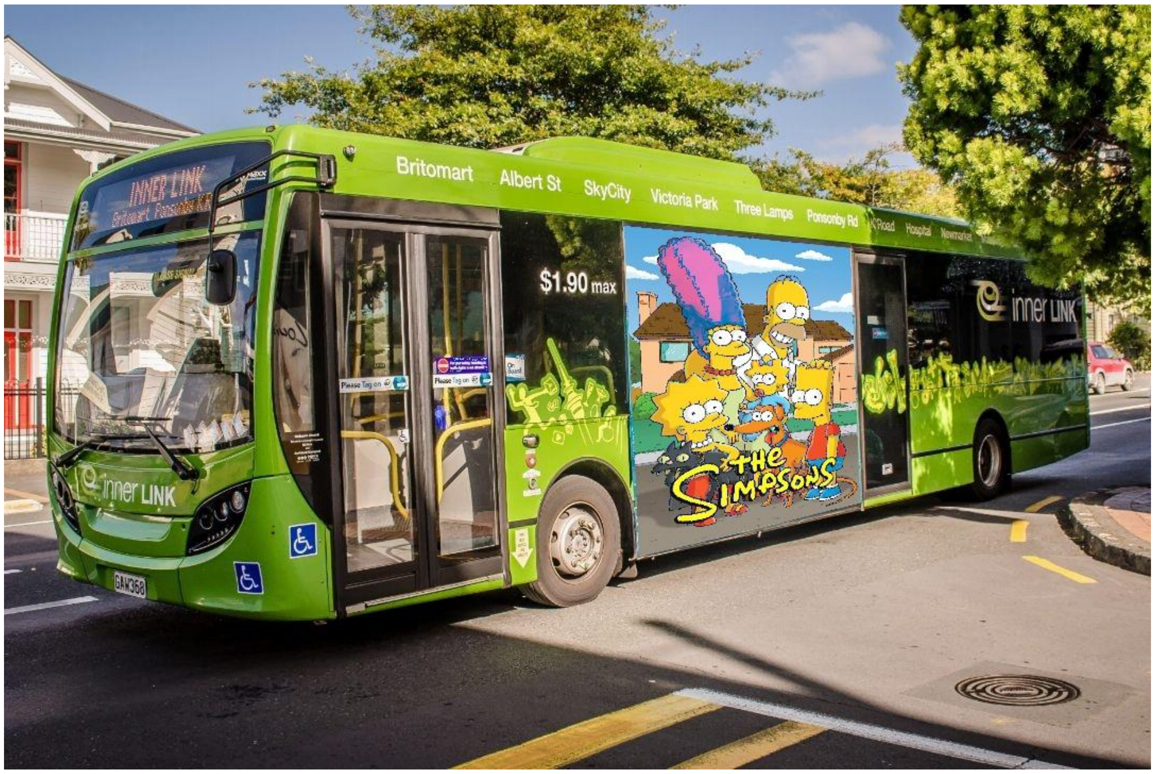
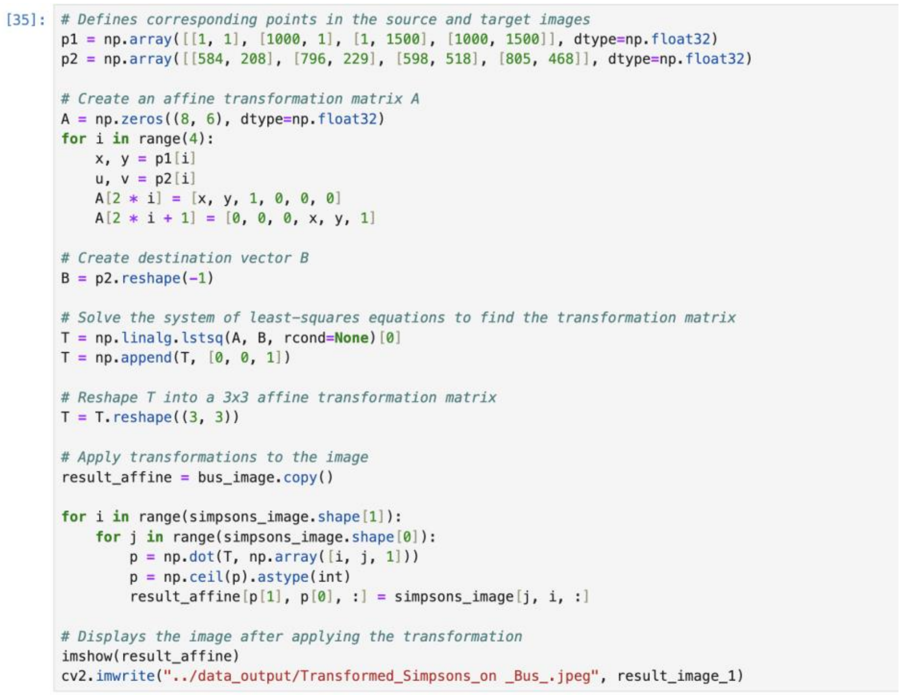


Image results after performing homographic transformation

3.3. Some other experiments

- Apply affine transformation



```
[35]: # Defines corresponding points in the source and target images
p1 = np.array([[1, 1], [1000, 1], [1, 1500], [1000, 1500]], dtype=np.float32)
p2 = np.array([[584, 208], [796, 229], [598, 518], [805, 468]], dtype=np.float32)

# Create an affine transformation matrix A
A = np.zeros((8, 6), dtype=np.float32)
for i in range(4):
    x, y = p1[i]
    u, v = p2[i]
    A[2 * i] = [x, y, 1, 0, 0, 0]
    A[2 * i + 1] = [0, 0, 0, x, y, 1]

# Create destination vector B
B = p2.reshape(-1)

# Solve the system of least-squares equations to find the transformation matrix
T = np.linalg.lstsq(A, B, rcond=None)[0]
T = np.append(T, [0, 0, 1])

# Reshape T into a 3x3 affine transformation matrix
T = T.reshape((3, 3))

# Apply transformations to the image
result_affine = bus_image.copy()

for i in range(simpsons_image.shape[1]):
    for j in range(simpsons_image.shape[0]):
        p = np.dot(T, np.array([i, j, 1]))
        p = np.ceil(p).astype(int)
        result_affine[p[1], p[0], :] = simpsons_image[j, i, :]

# Displays the image after applying the transformation
imshow(result_affine)
cv2.imwrite("../data_output/Transformed_Simpsons_on _Bus_.jpeg", result_image_1)
```

- When applying Affine transformation, we also obtain results similar to performing it using homographic transformation
- Based on my research, the main difference between them is the linearity and non-linearity of the transformation. Affine transformation is a special case of homographic transformation when the homography matrix is a linear matrix
- When we need to perform simple transformations like shifting, scaling, and rotating, affine transformation is a good choice. However, when we need to perform more complex transformations or transformations that unnecessarily preserve parallel lines, homographic transformation is more suitable.

Link code: https://github.com/LeHoa98ptit/ComputerVision_Lyon_1/blob/main/Assignment_1/B_Image_Transformations/Image_Transformation.ipynb

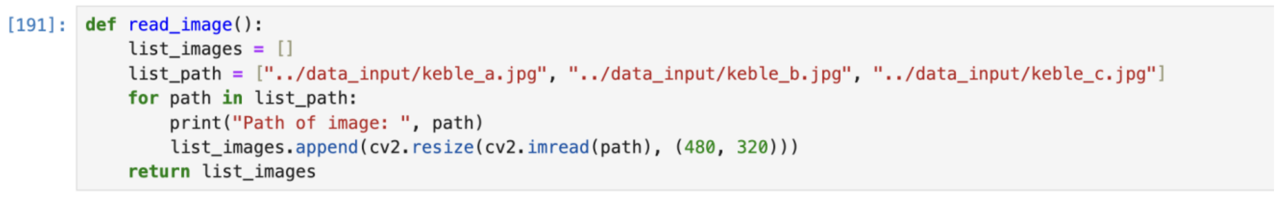
4. Panoramic Image Stitching

4.1. Introduction

In this problem we will learn how to use geometric transformations (homogeneous transformations) to create a panorama from a single image of Keble College, Oxford.

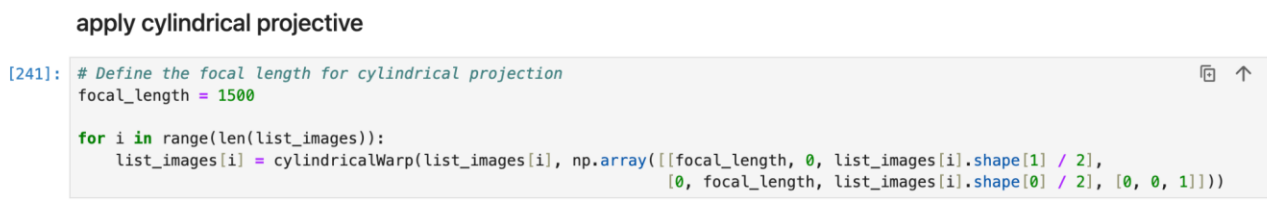
4.2. Methodology and Procedure

- Load images
- We create a function to sequentially read three Keble images



```
[191]: def read_image():
           list_images = []
           list_path = ["../data_input/keble_a.jpg", "../data_input/keble_b.jpg", "../data_input/keble_c.jpg"]
           for path in list_path:
               print("Path of image: ", path)
               list_images.append(cv2.resize(cv2.imread(path), (480, 320)))
           return list_images
```

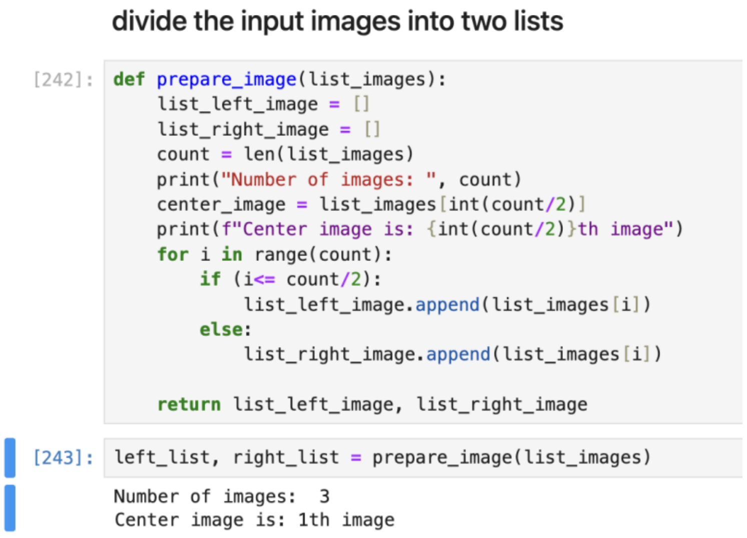
- Then we project each image cylindrically
- The purpose of cylindrical projection is to convert a panoramic scene into a flat image, which can further be used for panoramic compositing or other applications. It is a way to overcome the distortion caused by the wide field of view of panoramic images and make them more suitable for human perception and deeper image processing.



apply cylindrical projective

```
[241]: # Define the focal length for cylindrical projection
focal_length = 1500

for i in range(len(list_images)):
    list_images[i] = cylindricalWarp(list_images[i], np.array([[focal_length, 0, list_images[i].shape[1] / 2],
                                                              [0, focal_length, list_images[i].shape[0] / 2], [0, 0, 1]]))
```

- Divide the input images into two lists
- The purpose of splitting this list may be related to processing the left and right images separately in the panoramic image creation process or other image processing tasks.



divide the input images into two lists

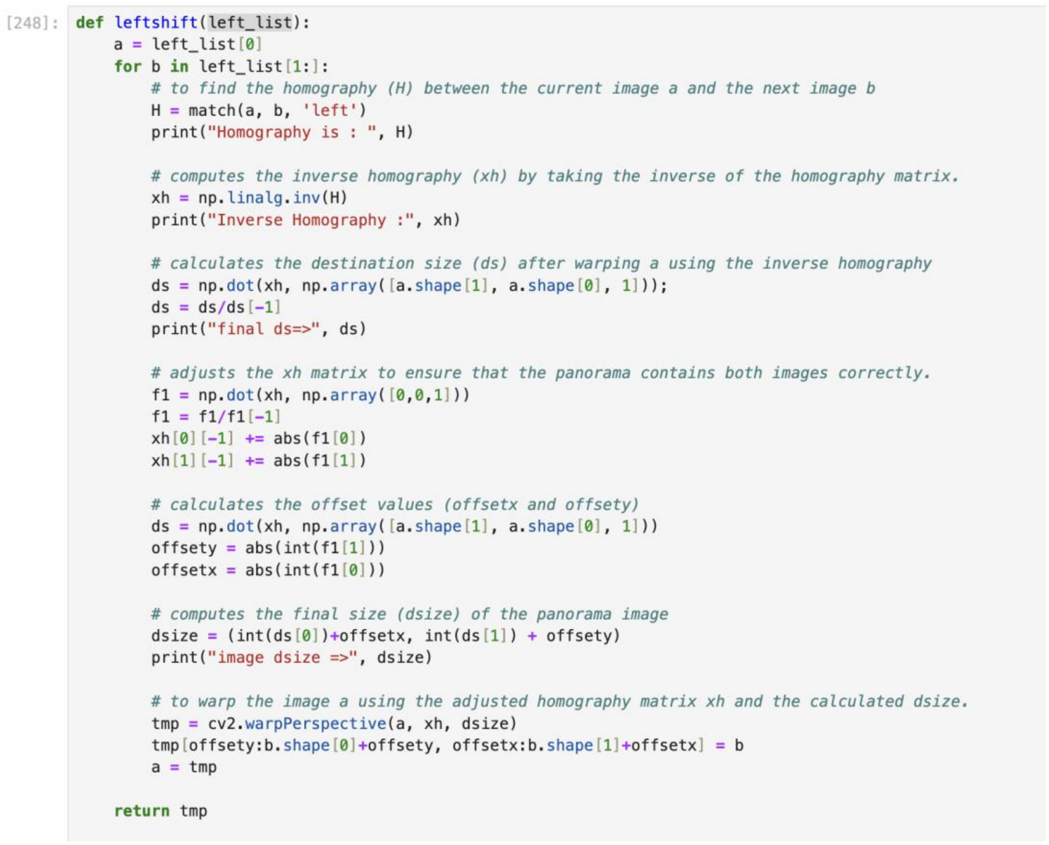
```
[242]: def prepare_image(list_images):
    list_left_image = []
    list_right_image = []
    count = len(list_images)
    print("Number of images: ", count)
    center_image = list_images[int(count/2)]
    print(f"Center image is: {int(count/2)}th image")
    for i in range(count):
        if (i<= count/2):
            list_left_image.append(list_images[i])
        else:
            list_right_image.append(list_images[i])

    return list_left_image, list_right_image
```

```
[243]: left_list, right_list = prepare_image(list_images)

Number of images:  3
Center image is: 1th image
```

- Perform image matching and find a homography transformation matrix
    - This homography matrix is used to align and stitch the images together to create a panoramic image
    - It uses the SIFT (Scale-Invariant Feature Transform) feature detector and descriptor to find keypoints and their descriptors for both input images. Keypoints represent distinctive points in the image, and descriptors are numerical representations of the local image structure around each keypoint.
    - It matches keypoints between image_1 and image_2 using the FLANN (Fast Library for Approximate Nearest Neighbors) based matcher. FLANN is used for efficient matching of keypoints between images.
    - The matches are filtered based on their distances.
    - If there are at least 5 "good" matches (as required for homography estimation), the code proceeds to compute the homography matrix (H) using the cv2.findHomography function with RANSAC (Random Sample Consensus) algorithm.

- Cuối cùng, ma trận homography (H) được trả về.
- If there are not enough "good" matches (less than 5), the code returns None because a reliable homography estimation is not possible, and these two images are not suitable for stitching.

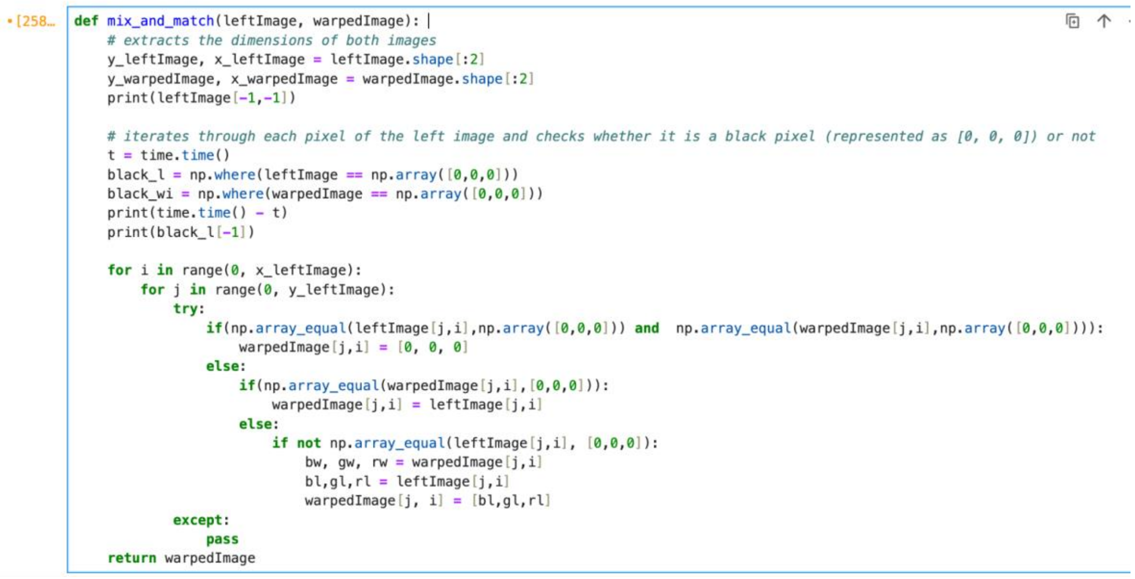
● Perform left image stitching



```
[248]: def leftshift(left_list):
           a = left_list[0]
           for b in left_list[1:]:
               # to find the homography (H) between the current image a and the next image b
               H = match(a, b, 'left')
               print("Homography is : ", H)

               # computes the inverse homography (xh) by taking the inverse of the homography matrix.
               xh = np.linalg.inv(H)
               print("Inverse Homography :", xh)

               # calculates the destination size (ds) after warping a using the inverse homography
               ds = np.dot(xh, np.array([a.shape[1], a.shape[0], 1]));
               ds = ds/ds[-1]
               print("final ds=>", ds)

               # adjusts the xh matrix to ensure that the panorama contains both images correctly.
               f1 = np.dot(xh, np.array([0,0,1]))
               f1 = f1/f1[-1]
               xh[0][-1] += abs(f1[0])
               xh[1][-1] += abs(f1[1])

               # calculates the offset values (offsetx and offsety)
               ds = np.dot(xh, np.array([a.shape[1], a.shape[0], 1]))
               offsety = abs(int(f1[1]))
               offsetx = abs(int(f1[0]))

               # computes the final size (dsize) of the panorama image
               dsize = (int(ds[0])+offsetx, int(ds[1]) + offsety)
               print("image dsize =>", dsize)

               # to warp the image a using the adjusted homography matrix xh and the calculated dsize.
               tmp = cv2.warpPerspective(a, xh, dsize)
               tmp[offsety:b.shape[0]+offsety, offsetx:b.shape[1]+offsetx] = b
               a = tmp

           return tmp
```

● Mix and match the left image and the warped image during the image stitching process

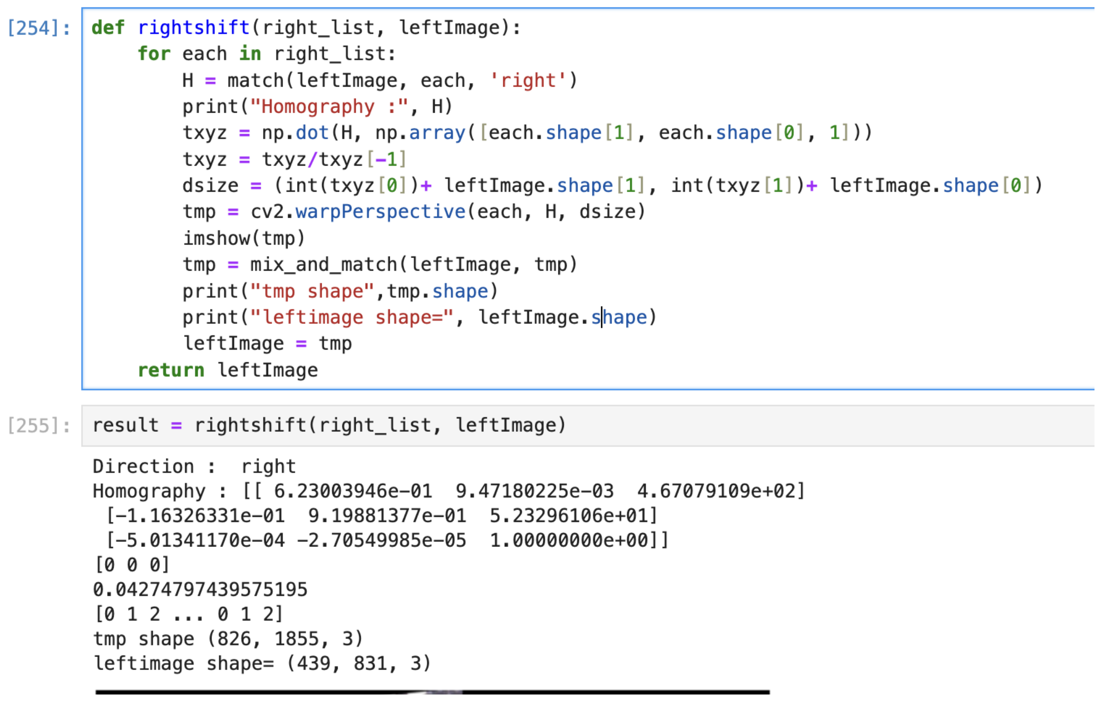
- This process effectively blends the two images, ensuring that any black regions in either image are properly handled, and the final panorama contains a mixture of both images without abrupt transitions



```
[258… def mix_and_match(leftImage, warpedImage):
    # extracts the dimensions of both images
    y_leftImage, x_leftImage = leftImage.shape[:2]
    y_warpedImage, x_warpedImage = warpedImage.shape[:2]
    print(leftImage[-1,-1])

    # iterates through each pixel of the left image and checks whether it is a black pixel (represented as [0, 0, 0]) or not
    t = time.time()
    black_l = np.where(leftImage == np.array([0,0,0]))
    black_wi = np.where(warpedImage == np.array([0,0,0]))
    print(time.time() - t)
    print(black_l[-1])

    for i in range(0, x_leftImage):
        for j in range(0, y_leftImage):
            try:
                if(np.array_equal(leftImage[j,i],np.array([0,0,0])) and  np.array_equal(warpedImage[j,i],np.array([0,0,0]))):
                    warpedImage[j,i] = [0, 0, 0]
                else:
                    if(np.array_equal(warpedImage[j,i],[0,0,0])):
                        warpedImage[j,i] = leftImage[j,i]
                    else:
                        if not np.array_equal(leftImage[j,i], [0,0,0]):
                            bw, gw, rw = warpedImage[j,i]
                            bl,gl,rl = leftImage[j,i]
                            warpedImage[j, i] = [bl,gl,rl]
            except:
                pass
    return warpedImage
```

- Perform the stitching of all images

- This function takes a list of images on the right side, calculates the necessary transformations to align them with the existing left image, and blends them together to create a seamless panoramic image

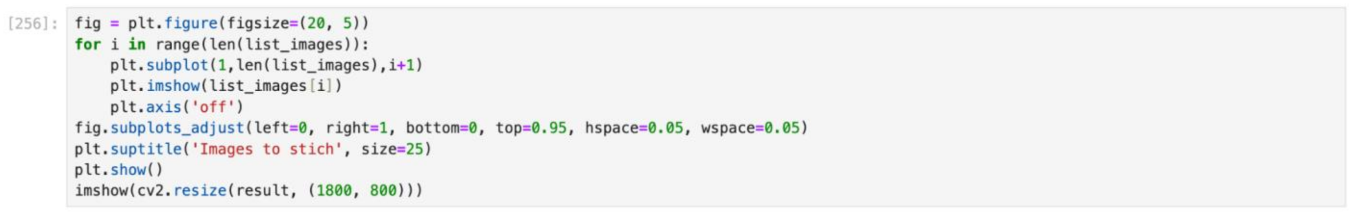


```
[254]: def rightshift(right_list, leftImage):
           for each in right_list:
               H = match(leftImage, each, 'right')
               print("Homography :", H)
               txyz = np.dot(H, np.array([each.shape[1], each.shape[0], 1]))
               txyz = txyz/txyz[-1]
               dsize = (int(txyz[0])+ leftImage.shape[1], int(txyz[1])+ leftImage.shape[0])
               tmp = cv2.warpPerspective(each, H, dsize)
               imshow(tmp)
               tmp = mix_and_match(leftImage, tmp)
               print("tmp shape",tmp.shape)
               print("leftimage shape=", leftImage.shape)
               leftImage = tmp
           return leftImage
```

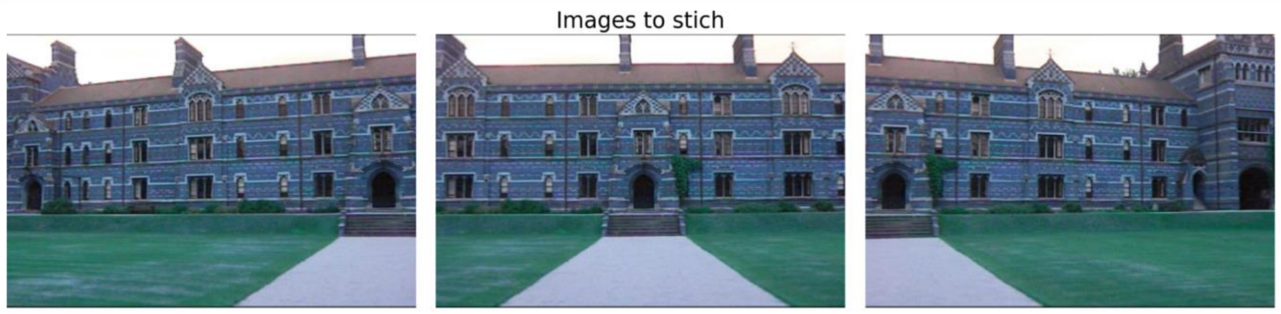
```
[255]: result = rightshift(right_list, leftImage)
```

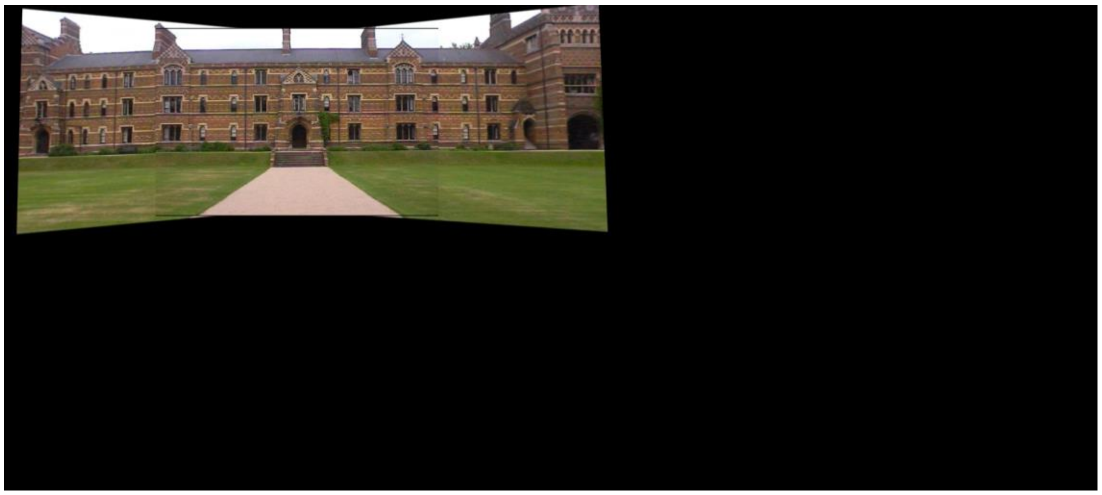
```
Direction :  right
Homography : [[ 6.23003946e-01  9.47180225e-03  4.67079109e+02]
 [-1.16326331e-01  9.19881377e-01  5.23296106e+01]
 [-5.01341170e-04 -2.70549985e-05  1.00000000e+00]]
[0 0 0]
0.04274797439575195
[0 1 2 ... 0 1 2]
tmp shape (826, 1855, 3)
leftimage shape= (439, 831, 3)
```

- Display results



```
[256]: fig = plt.figure(figsize=(20, 5))
       for i in range(len(list_images)):
           plt.subplot(1,len(list_images),i+1)
           plt.imshow(list_images[i])
           plt.axis('off')
       fig.subplots_adjust(left=0, right=1, bottom=0, top=0.95, hspace=0.05, wspace=0.05)
       plt.suptitle('Images to stich', size=25)
       plt.show()
       imshow(cv2.resize(result, (1800, 800)))
```





The image displays the result after stitching

Link code:
https://github.com/LeHoa98ptit/ComputerVision_Lyon_1/blob/main/Assignment_1/C_Panaromic_Image_Stitching/Image_Stitching_Paronama.ipynb