Computer Vision 2023 Assignment 2: Image matching and retrieval

In this prac, you will experiment with image feature detectors, descriptors and matching. There are 3 main parts to the prac:

- matching an object in a pair of images
- searching for an object in a collection of images
- analysis and discussion of results

General instructions

As before, you will use this notebook to run your code and display your results and analysis. Again we will mark a PDF conversion of your notebook, referring to your code if necessary, so you should ensure your code output is formatted neatly.

When converting to PDF, include the outputs and analysis only, not your code. You can do this from the command line using the `nbconvert` command (installed as part of Jupyter) as follows:

```
jupyter nbconvert Assignment2.ipynb --to pdf --no-input --
TagRemovePreprocessor.remove_cell_tags 'remove-cell'
```

This will also remove the preamble text from each question. It has been packaged into a small notebook you can run in colab, called notebooktopdf.ipynb

We will use the `OpenCV` library to complete the prac. It has several built in functions that will be useful. You are expected to consult documentation and use them appropriately.

As with the last assignment it is somewhat up to you how you answer each question. Ensure that the outputs and report are clear and easy to read so that the markers can rapidly assess what you have done, why, and how deep is your understanding. This includes:

- sizing, arranging and captioning image outputs appropriately
- explaining what you have done clearly and concisely
- clearly separating answers to each question

Data

We have provided some example images for this assignment, available through a link on the MyUni assignment page. The images are organised by subject matter, with one folder containing images of book covers, one of museum exhibits, and another of urban landmarks. You should copy these data into a directory A2_smvs, keeping the directory structure the same as in the zip file.

Within each category (within each folder), there is a "Reference" folder containing a clean image of each object and a "Query" folder containing images taken on a mobile device. Within each category, images with the same name contain the same object (so 001.jpg in the Reference folder contains the same book as 001.jpg in the Query folder). The data is a subset of the Stanford Mobile Visual Search Dataset which is available at

http://web.cs.wpi.edu/~claypool/mmsys-dataset/2011/stanford/index.html.

The full data set contains more image categories and more query images of the objects we have provided, which may be useful for your testing!

Do not submit your own copy of the data or rename any files or folders! For marking, we will assume the datasets are available in subfolders of the working directory using the same folder names provided.

Here is some general setup code, which you can edit to suit your needs.

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

In [ ]:
```
!pwd
%cd drive/MyDrive/Colab\ Notebooks
!pwd
```

In [21]:
```
# Numpy is the main package for scientific computing with Python.
import numpy as np
import cv2

# Matplotlib is a useful plotting library for python
import matplotlib.pyplot as plt
# This code is to make matplotlib figures appear inline in the
# notebook rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots, can be ch
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# Some more magic so that the notebook will reload external python modules;
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
%reload_ext autoreload
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

In [22]:
```python
def draw_outline(ref, query, model):
    """
        Draw outline of reference image in the query image.
        This is just an example to show the steps involved.
        You can modify to suit your needs.
        Inputs:
            ref: reference image
            query: query image
            model: estimated transformation from query to reference image
    """
    h,w = ref.shape[:2]
    pts = np.float32([ [0,0],[0,h-1],[w-1,h-1],[w-1,0] ]).reshape(-1,1,2)
    dst = cv2.perspectiveTransform(pts,model)

    img = query.copy()
    img = cv2.polylines(img,[np.int32(dst)],True,255,3, cv2.LINE_AA)
    plt.imshow(img, 'gray'), plt.show()

def draw_inliers(img1, img2, kp1, kp2, matches, matchesMask):
    """
        Draw inlier between images
        img1 / img2: reference/query  img
        kp1 / kp2: their keypoints
        matches : list of (good) matches after ratio test
        matchesMask: Inlier mask returned in cv2.findHomography()
    """
    matchesMask = matchesMask.ravel().tolist()
    draw_params = dict(matchColor = (0,255,0), # draw matches in green color
                       singlePointColor = None,
                       matchesMask = matchesMask, # draw only inliers
                       flags = 2)
    img3 = cv2.drawMatches(img1,kp1,img2,kp2,matches,None,**draw_params)
    plt.imshow(img3, 'gray'),plt.show()
```

Question 1: Matching an object in a pair of images (60%)

In this question, the aim is to accurately locate a reference object in a query image, for example:

 Books

0. Download and read through the paper ORB: an efficient alternative to SIFT or SURF by Rublee et al. You don't need to understand all the details, but try to get an idea of how it works. ORB combines the FAST corner detector (covered in week 3) and the BRIEF descriptor. BRIEF is based on similar ideas to the SIFT descriptor we covered week 3, but with some changes for efficiency.

1. [Load images] Load the first (reference, query) image pair from the "book_covers" category using opencv (e.g. `img=cv2.imread()`). Check the parameter option in " cv2.imread()" to ensure that you read the gray scale image, since it is necessary for computing ORB features.

2. [Detect features] Create opencv ORB feature extractor by `orb=cv2.ORB_create()`. Then you can detect keypoints by `kp = orb.detect(img,None)`, and compute descriptors by `kp, des = orb.compute(img, kp)`. You need to do this for each image, and then you can use `cv2.drawKeypoints()` for visualization.

3. [Match features] As ORB is a binary feature, you need to use HAMMING distance for matching, e.g., `bf = cv2.BFMatcher(cv2.NORM_HAMMING)`. Then you are requried to do KNN matching (k=2) by using `bf.knnMatch()`. After that, you are required to use "ratio_test" to find good matches. By default, you can set `ratio=0.8`.

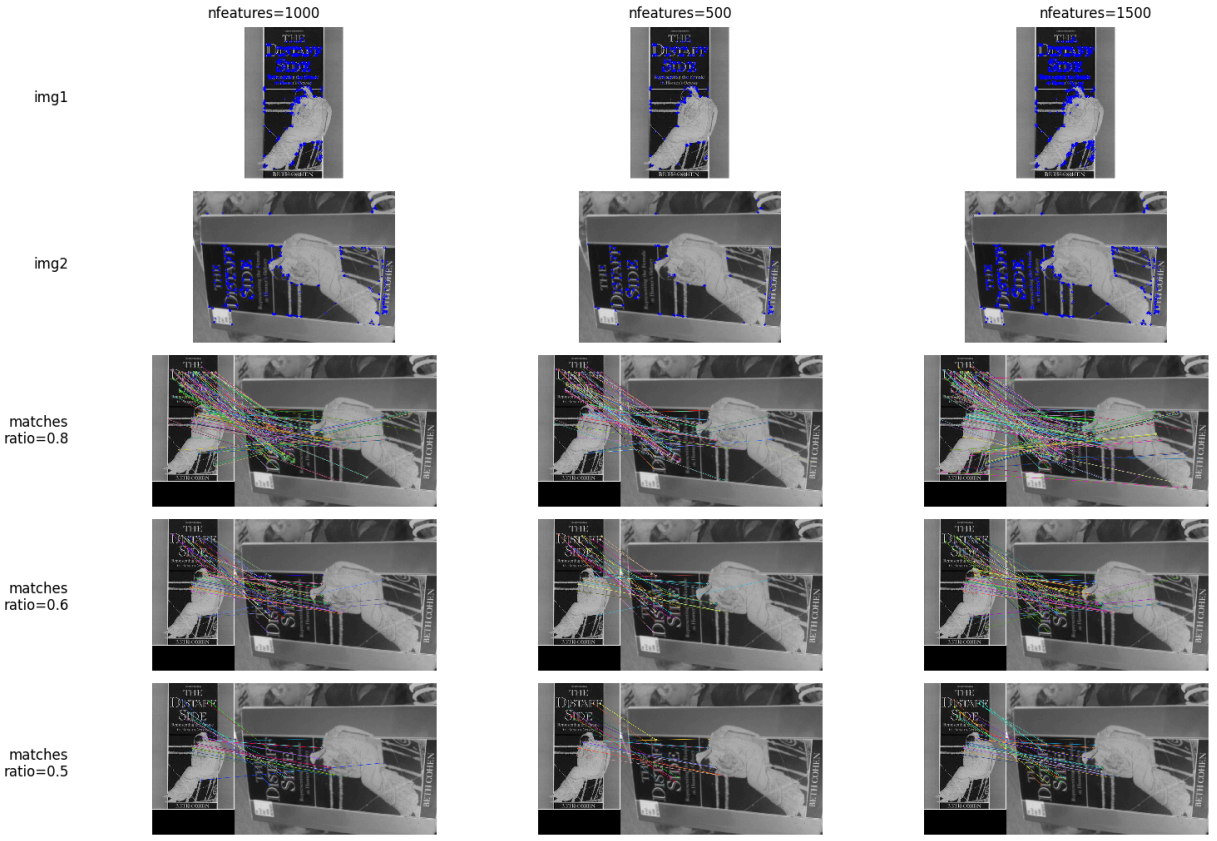
4. [Plot and analyze] You need to visualize the matches by using the `cv2.drawMatches()` function. Also you can change the ratio values, parameters in `cv2.ORB_create()`, and distance functions in `cv2.BFMatcher()`. Please discuss how these changes influence the match numbers.

In [23]:
```
# load images as grey scale
img1 = cv2.imread('a2/A2_smvs/book_covers/Reference/001.jpg', 0)
if not np.shape(img1):
    # Error message and print current working dir
    print("Could not load img1. Check the path, filename and current working direct
    !pwd
img2 = cv2.imread("a2/A2_smvs/book_covers/Query/001.jpg", 0)
if not np.shape(img2):
    # Error message and print current working dir
    print("Could not load img2. Check the path, filename and current working direct
    !pwd
```

In [24]:
```
# Your code for descriptor matching tests here

# compute detector and descriptor, see (2) above
# find the keypoints and descriptors with ORB, see (2) above
orb_1000_hm = cv2.ORB_create(nfeatures=1000)
kp1_orb_1000_hm, des1_orb_1000_hm = orb_1000_hm.detectAndCompute(img1, None)
kp2_orb_1000_hm, des2_orb_1000_hm = orb_1000_hm.detectAndCompute(img2, None)

orb_500_hm = cv2.ORB_create(nfeatures=500)
kp1_orb_500_hm, des1_orb_500_hm = orb_500_hm.detectAndCompute(img1, None)
kp2_orb_500_hm, des2_orb_500_hm = orb_500_hm.detectAndCompute(img2, None)

orb_1500_hm = cv2.ORB_create(nfeatures=1500)
kp1_orb_1500_hm, des1_orb_1500_hm = orb_1500_hm.detectAndCompute(img1, None)
kp2_orb_1500_hm, des2_orb_1500_hm = orb_1500_hm.detectAndCompute(img2, None)
```

```
# draw keypoints, see (2) above
img1_with_keypoints_1000_hm = cv2.drawKeypoints(img1, kp1_orb_1000_hm, None, color=
img1_with_keypoints_500_hm = cv2.drawKeypoints(img1, kp1_orb_500_hm, None, color=(2
img1_with_keypoints_1500_hm = cv2.drawKeypoints(img1, kp1_orb_1500_hm, None, color=

img2_with_keypoints_1000_hm = cv2.drawKeypoints(img2, kp2_orb_1000_hm, None, color=
img2_with_keypoints_500_hm = cv2.drawKeypoints(img2, kp2_orb_500_hm, None, color=(2
img2_with_keypoints_1500_hm = cv2.drawKeypoints(img2, kp2_orb_1500_hm, None, color=

# create BFMatcher object, see (3) above
bf_hm = cv2.BFMatcher(cv2.NORM_HAMMING)

# Match descriptors, see (3) above
matches_1000_hm = bf_hm.knnMatch(des1_orb_1000_hm, des2_orb_1000_hm, k=2)
matches_500_hm = bf_hm.knnMatch(des1_orb_500_hm, des2_orb_500_hm, k=2)
matches_1500_hm = bf_hm.knnMatch(des1_orb_1500_hm, des2_orb_1500_hm, k=2)

# Apply ratio test, see (3) above
good_1000_hm, good_500_hm, good_1500_hm = [], [], []
good_1000_06_hm, good_500_06_hm, good_1500_06_hm = [], [], []
good_1000_05_hm, good_500_05_hm, good_1500_05_hm = [], [], []
ratio_08, ratio_06, ratio_05 = 0.8, 0.6, 0.5
for m, n in matches_1000_hm:
    if m.distance < ratio_08 * n.distance:
        good_1000_hm.append([m])
    if m.distance < ratio_06 * n.distance:
        good_1000_06_hm.append([m])
    if m.distance < ratio_05 * n.distance:
        good_1000_05_hm.append([m])
for m, n in matches_500_hm:
    if m.distance < ratio_08 * n.distance:
        good_500_hm.append([m])
    if m.distance < ratio_06 * n.distance:
        good_500_06_hm.append([m])
    if m.distance < ratio_05 * n.distance:
        good_500_05_hm.append([m])
for m, n in matches_1500_hm:
    if m.distance < ratio_08 * n.distance:
        good_1500_hm.append([m])
    if m.distance < ratio_06 * n.distance:
        good_1500_06_hm.append([m])
    if m.distance < ratio_05 * n.distance:
        good_1500_05_hm.append([m])

# draw matches, see (4) above
res_1000_hm = cv2.drawMatchesKnn(img1, kp1_orb_1000_hm, img2, kp2_orb_1000_hm, good
res_500_hm = cv2.drawMatchesKnn(img1, kp1_orb_500_hm, img2, kp2_orb_500_hm, good_50
res_1500_hm = cv2.drawMatchesKnn(img1, kp1_orb_1500_hm, img2, kp2_orb_1500_hm, good

res_1000_06_hm = cv2.drawMatchesKnn(img1, kp1_orb_1000_hm, img2, kp2_orb_1000_hm, g
res_500_06_hm = cv2.drawMatchesKnn(img1, kp1_orb_500_hm, img2, kp2_orb_500_hm, good
res_1500_06_hm = cv2.drawMatchesKnn(img1, kp1_orb_1500_hm, img2, kp2_orb_1500_hm, g

res_1000_05_hm = cv2.drawMatchesKnn(img1, kp1_orb_1000_hm, img2, kp2_orb_1000_hm, g
res_500_05_hm = cv2.drawMatchesKnn(img1, kp1_orb_500_hm, img2, kp2_orb_500_hm, good
```

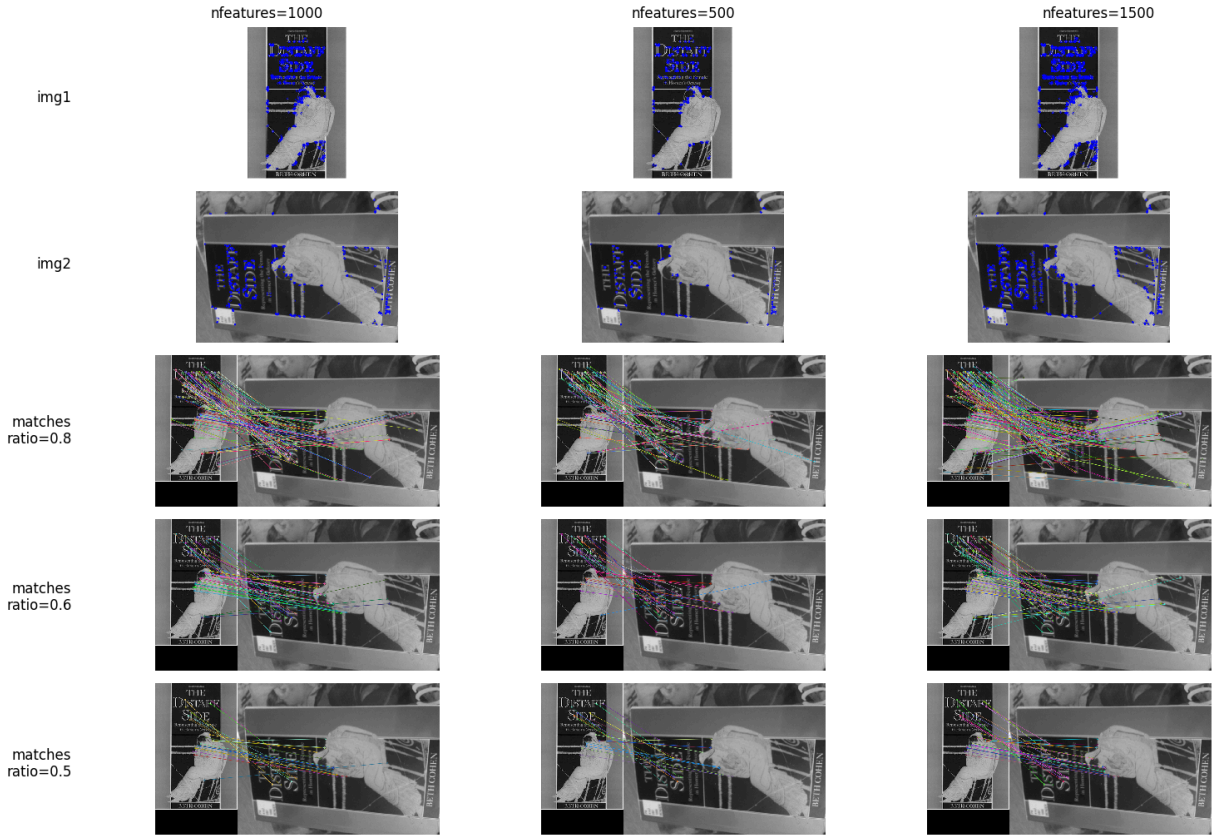
```
res_1500_05_hm = cv2.drawMatchesKnn(img1, kp1_orb_1500_hm, img2, kp2_orb_1500_hm, g

fig1, axs1 = plt.subplots(5, 3, figsize=(15, 10))

row_labels = ['img1', 'img2', 'matches\nratio=0.8', 'matches\nratio=0.6', 'matches\
col_labels = ['nfeatures=1000', 'nfeatures=500', 'nfeatures=1500']
images = [
    [img1_with_keypoints_1000_hm, img1_with_keypoints_500_hm, img1_with_keypoints_1
    [img2_with_keypoints_1000_hm, img2_with_keypoints_500_hm, img2_with_keypoints_1
    [res_1000_hm, res_500_hm, res_1500_hm],
    [res_1000_06_hm, res_500_06_hm, res_1500_06_hm],
    [res_1000_05_hm, res_500_05_hm, res_1500_05_hm]
]

row_height = 1 / len(axs1)
for i, label in enumerate(row_labels):
    fig1.text(0.01, 1 - (i + 0.5) * row_height, label, va='center', ha='right', siz

col_width = 1 / len(axs1[0])
for i, label in enumerate(col_labels):
    fig1.text((i + 0.5) * col_width, 1.01, label, va='top', ha='center', size='larg

for i, row in enumerate(images):
    for j, img in enumerate(row):
        axs1[i, j].imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
        axs1[i, j].axis('off')

plt.tight_layout()
plt.show()
```



In [18]:

```
# Your code for descriptor matching tests here

# compute detector and descriptor, see (2) above
# find the keypoints and descriptors with ORB, see (2) above
orb_1000_l2 = cv2.ORB_create(nfeatures=1000)
kp1_orb_1000_l2, des1_orb_1000_l2 = orb_1000_l2.detectAndCompute(img1, None)
kp2_orb_1000_l2, des2_orb_1000_l2 = orb_1000_l2.detectAndCompute(img2, None)

orb_500_l2 = cv2.ORB_create(nfeatures=500)
kp1_orb_500_l2, des1_orb_500_l2 = orb_500_l2.detectAndCompute(img1, None)
kp2_orb_500_l2, des2_orb_500_l2 = orb_500_l2.detectAndCompute(img2, None)

orb_1500_l2 = cv2.ORB_create(nfeatures=1500)
kp1_orb_1500_l2, des1_orb_1500_l2 = orb_1500_l2.detectAndCompute(img1, None)
kp2_orb_1500_l2, des2_orb_1500_l2 = orb_1500_l2.detectAndCompute(img2, None)

# draw keypoints, see (2) above
img1_with_keypoints_1000_l2 = cv2.drawKeypoints(img1, kp1_orb_1000_l2, None, color=
img1_with_keypoints_500_l2 = cv2.drawKeypoints(img1, kp1_orb_500_l2, None, color=(2
img1_with_keypoints_1500_l2 = cv2.drawKeypoints(img1, kp1_orb_1500_l2, None, color=

img2_with_keypoints_1000_l2 = cv2.drawKeypoints(img2, kp2_orb_1000_l2, None, color=
img2_with_keypoints_500_l2 = cv2.drawKeypoints(img2, kp2_orb_500_l2, None, color=(2
img2_with_keypoints_1500_l2 = cv2.drawKeypoints(img2, kp2_orb_1500_l2, None, color=

# create BFMatcher object, see (3) above
bf_l2 = cv2.BFMatcher(cv2.NORM_HAMMING)

# Match descriptors, see (3) above
matches_1000_l2 = bf_l2.knnMatch(des1_orb_1000_l2, des2_orb_1000_l2, k=2)
matches_500_l2 = bf_l2.knnMatch(des1_orb_500_l2, des2_orb_500_l2, k=2)
matches_1500_l2 = bf_l2.knnMatch(des1_orb_1500_l2, des2_orb_1500_l2, k=2)

# Apply ratio test, see (3) above
good_1000_l2, good_500_l2, good_1500_l2 = [], [], []
good_1000_06_l2, good_500_06_l2, good_1500_06_l2 = [], [], []
good_1000_05_l2, good_500_05_l2, good_1500_05_l2 = [], [], []
ratio_08, ratio_06, ratio_05 = 0.8, 0.6, 0.5
for m, n in matches_1000_l2:
    if m.distance < ratio_08 * n.distance:
        good_1000_l2.append([m])
    if m.distance < ratio_06 * n.distance:
        good_1000_06_l2.append([m])
    if m.distance < ratio_05 * n.distance:
        good_1000_05_l2.append([m])
for m, n in matches_500_l2:
    if m.distance < ratio_08 * n.distance:
        good_500_l2.append([m])
    if m.distance < ratio_06 * n.distance:
        good_500_06_l2.append([m])
    if m.distance < ratio_05 * n.distance:
        good_500_05_l2.append([m])
for m, n in matches_1500_l2:
    if m.distance < ratio_08 * n.distance:
        good_1500_l2.append([m])
```

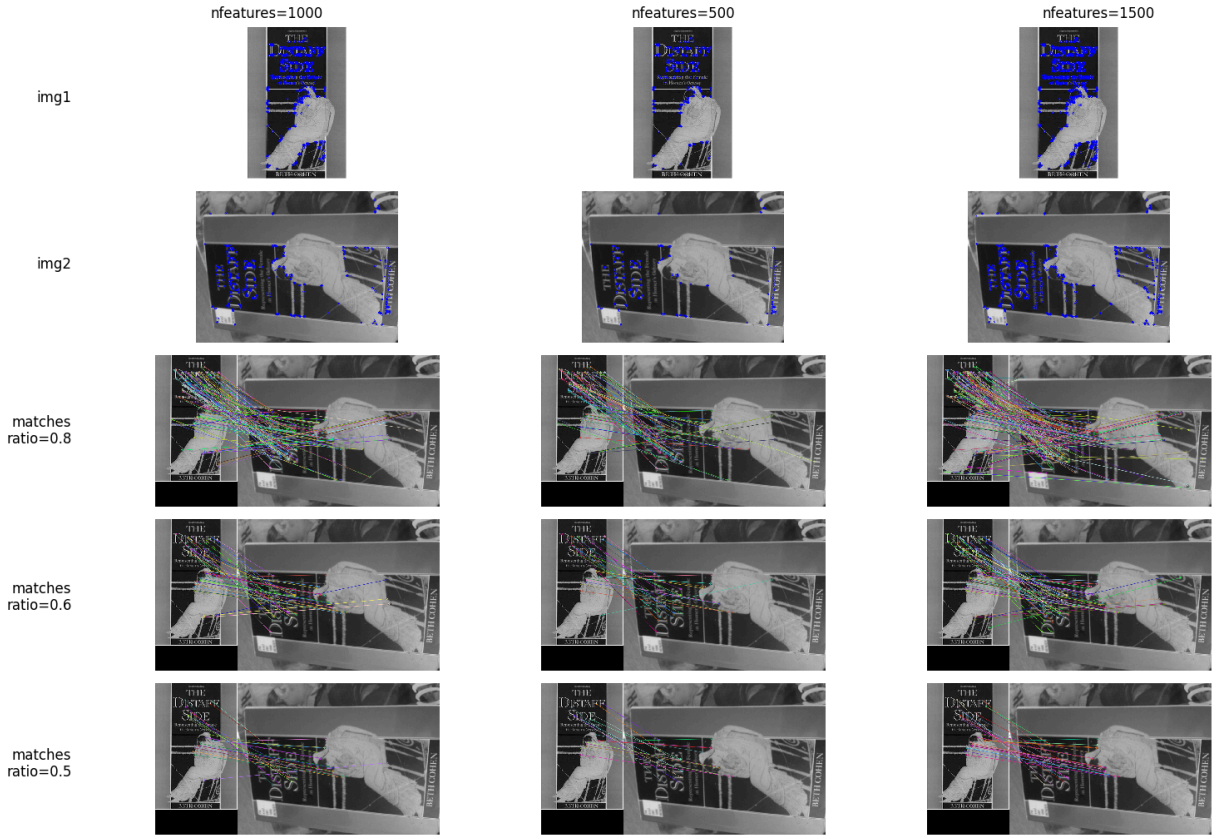
```
    if m.distance < ratio_06 * n.distance:
        good_1500_06_l2.append([m])
    if m.distance < ratio_05 * n.distance:
        good_1500_05_l2.append([m])

# draw matches, see (4) above
res_1000_l2 = cv2.drawMatchesKnn(img1, kp1_orb_1000_l2, img2, kp2_orb_1000_l2, good
res_500_l2 = cv2.drawMatchesKnn(img1, kp1_orb_500_l2, img2, kp2_orb_500_l2, good_50
res_1500_l2 = cv2.drawMatchesKnn(img1, kp1_orb_1500_l2, img2, kp2_orb_1500_l2, good

res_1000_06_l2 = cv2.drawMatchesKnn(img1, kp1_orb_1000_l2, img2, kp2_orb_1000_l2, g
res_500_06_l2 = cv2.drawMatchesKnn(img1, kp1_orb_500_l2, img2, kp2_orb_500_l2, good
res_1500_06_l2 = cv2.drawMatchesKnn(img1, kp1_orb_1500_l2, img2, kp2_orb_1500_l2, g

res_1000_05_l2 = cv2.drawMatchesKnn(img1, kp1_orb_1000_l2, img2, kp2_orb_1000_l2, g
res_500_05_l2 = cv2.drawMatchesKnn(img1, kp1_orb_500_l2, img2, kp2_orb_500_l2, good
res_1500_05_l2 = cv2.drawMatchesKnn(img1, kp1_orb_1500_l2, img2, kp2_orb_1500_l2, g

fig1, axs1 = plt.subplots(5, 3, figsize=(15, 10))

row_labels = ['img1', 'img2', 'matches\nratio=0.8', 'matches\nratio=0.6', 'matches\
col_labels = ['nfeatures=1000', 'nfeatures=500', 'nfeatures=1500']
images = [
    [img1_with_keypoints_1000_l2, img1_with_keypoints_500_l2, img1_with_keypoints_1
    [img2_with_keypoints_1000_l2, img2_with_keypoints_500_l2, img2_with_keypoints_1
    [res_1000_l2, res_500_l2, res_1500_l2],
    [res_1000_06_l2, res_500_06_l2, res_1500_06_l2],
    [res_1000_05_l2, res_500_05_l2, res_1500_05_l2]
]

row_height = 1 / len(axs1)
for i, label in enumerate(row_labels):
    fig1.text(0.01, 1 - (i + 0.5) * row_height, label, va='center', ha='right', siz

col_width = 1 / len(axs1[0])
for i, label in enumerate(col_labels):
    fig1.text((i + 0.5) * col_width, 1.01, label, va='top', ha='center', size='larg

for i, row in enumerate(images):
    for j, img in enumerate(row):
        axs1[i, j].imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
        axs1[i, j].axis('off')

plt.tight_layout()
plt.show()
```



In [19]:
```
# Your code for descriptor matching tests here

# compute detector and descriptor, see (2) above
# find the keypoints and descriptors with ORB, see (2) above
orb_1000_hm2 = cv2.ORB_create(nfeatures=1000)
kp1_orb_1000_hm2, des1_orb_1000_hm2 = orb_1000_hm2.detectAndCompute(img1, None)
kp2_orb_1000_hm2, des2_orb_1000_hm2 = orb_1000_hm2.detectAndCompute(img2, None)

orb_500_hm2 = cv2.ORB_create(nfeatures=500)
kp1_orb_500_hm2, des1_orb_500_hm2 = orb_500_hm2.detectAndCompute(img1, None)
kp2_orb_500_hm2, des2_orb_500_hm2 = orb_500_hm2.detectAndCompute(img2, None)

orb_1500_hm2 = cv2.ORB_create(nfeatures=1500)
kp1_orb_1500_hm2, des1_orb_1500_hm2 = orb_1500_hm2.detectAndCompute(img1, None)
kp2_orb_1500_hm2, des2_orb_1500_hm2 = orb_1500_hm2.detectAndCompute(img2, None)

# draw keypoints, see (2) above
img1_with_keypoints_1000_hm2 = cv2.drawKeypoints(img1, kp1_orb_1000_hm2, None, colo
img1_with_keypoints_500_hm2 = cv2.drawKeypoints(img1, kp1_orb_500_hm2, None, color=
img1_with_keypoints_1500_hm2 = cv2.drawKeypoints(img1, kp1_orb_1500_hm2, None, colo

img2_with_keypoints_1000_hm2 = cv2.drawKeypoints(img2, kp2_orb_1000_hm2, None, colo
img2_with_keypoints_500_hm2 = cv2.drawKeypoints(img2, kp2_orb_500_hm2, None, color=
img2_with_keypoints_1500_hm2 = cv2.drawKeypoints(img2, kp2_orb_1500_hm2, None, colo

# create BFMatcher object, see (3) above
bf_hm2 = cv2.BFMatcher(cv2.NORM_HAMMING)

# Match descriptors, see (3) above
matches_1000_hm2 = bf_hm2.knnMatch(des1_orb_1000_hm2, des2_orb_1000_hm2, k=2)
```

```
matches_500_hm2 = bf_hm2.knnMatch(des1_orb_500_hm2, des2_orb_500_hm2, k=2)
matches_1500_hm2 = bf_hm2.knnMatch(des1_orb_1500_hm2, des2_orb_1500_hm2, k=2)

# Apply ratio test, see (3) above
good_1000_hm2, good_500_hm2, good_1500_hm2 = [], [], []
good_1000_06_hm2, good_500_06_hm2, good_1500_06_hm2 = [], [], []
good_1000_05_hm2, good_500_05_hm2, good_1500_05_hm2 = [], [], []
ratio_08, ratio_06, ratio_05 = 0.8, 0.6, 0.5
for m, n in matches_1000_hm2:
    if m.distance < ratio_08 * n.distance:
        good_1000_hm2.append([m])
    if m.distance < ratio_06 * n.distance:
        good_1000_06_hm2.append([m])
    if m.distance < ratio_05 * n.distance:
        good_1000_05_hm2.append([m])
for m, n in matches_500_hm2:
    if m.distance < ratio_08 * n.distance:
        good_500_hm2.append([m])
    if m.distance < ratio_06 * n.distance:
        good_500_06_hm2.append([m])
    if m.distance < ratio_05 * n.distance:
        good_500_05_hm2.append([m])
for m, n in matches_1500_hm2:
    if m.distance < ratio_08 * n.distance:
        good_1500_hm2.append([m])
    if m.distance < ratio_06 * n.distance:
        good_1500_06_hm2.append([m])
    if m.distance < ratio_05 * n.distance:
        good_1500_05_hm2.append([m])

# draw matches, see (4) above
res_1000_hm2 = cv2.drawMatchesKnn(img1, kp1_orb_1000_hm2, img2, kp2_orb_1000_hm2, g
res_500_hm2 = cv2.drawMatchesKnn(img1, kp1_orb_500_hm2, img2, kp2_orb_500_hm2, good
res_1500_hm2 = cv2.drawMatchesKnn(img1, kp1_orb_1500_hm2, img2, kp2_orb_1500_hm2, g

res_1000_06_hm2 = cv2.drawMatchesKnn(img1, kp1_orb_1000_hm2, img2, kp2_orb_1000_hm2
res_500_06_hm2 = cv2.drawMatchesKnn(img1, kp1_orb_500_hm2, img2, kp2_orb_500_hm2, g
res_1500_06_hm2 = cv2.drawMatchesKnn(img1, kp1_orb_1500_hm2, img2, kp2_orb_1500_hm2

res_1000_05_hm2 = cv2.drawMatchesKnn(img1, kp1_orb_1000_hm2, img2, kp2_orb_1000_hm2
res_500_05_hm2 = cv2.drawMatchesKnn(img1, kp1_orb_500_hm2, img2, kp2_orb_500_hm2, g
res_1500_05_hm2 = cv2.drawMatchesKnn(img1, kp1_orb_1500_hm2, img2, kp2_orb_1500_hm2

fig1, axs1 = plt.subplots(5, 3, figsize=(15, 10))

row_labels = ['img1', 'img2', 'matches\nratio=0.8', 'matches\nratio=0.6', 'matches\
col_labels = ['nfeatures=1000', 'nfeatures=500', 'nfeatures=1500']
images = [
    [img1_with_keypoints_1000_hm2, img1_with_keypoints_500_hm2, img1_with_keypoints
    [img2_with_keypoints_1000_hm2, img2_with_keypoints_500_hm2, img2_with_keypoints
    [res_1000_hm2, res_500_hm2, res_1500_hm2],
    [res_1000_06_hm2, res_500_06_hm2, res_1500_06_hm2],
    [res_1000_05_hm2, res_500_05_hm2, res_1500_05_hm2]
]

row_height = 1 / len(axs1)
```

```
for i, label in enumerate(row_labels):
    fig1.text(0.01, 1 - (i + 0.5) * row_height, label, va='center', ha='right', siz

col_width = 1 / len(axs1[0])
for i, label in enumerate(col_labels):
    fig1.text((i + 0.5) * col_width, 1.01, label, va='top', ha='center', size='larg

for i, row in enumerate(images):
    for j, img in enumerate(row):
        axs1[i, j].imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
        axs1[i, j].axis('off')

plt.tight_layout()
plt.show()
```



Your explanation of what you have done, and your results, here I tested the difference between the violent matcher matching img1 and img2 keypoints under different distance metrics. In total, it was chosen:

- Distance metrics: Hamming distance, Hamming distance 2, L2 paradigm
- The nfeatures parameter of the ORB function: 1000, 500, 1500
- Ratio of keypoint matching: 0.8, 0.6, 0.5 Thus a total of 45 plots were output in the three cells as a way to see the differences between the results due to different parameter settings.

I found that for ORB, a binary descriptor-based matching method, the use of the Hamming distance (including the modified Hamming distance 2) is better than the use of the traditional L2 paradigm.The ORB matching method under the L2 paradigm

yields fewer keypoints, and although the matches are more accurate, they are almost not obtained at smaller ratios; whereas the Hamming distance with the Hamming distance 2 performs comparably, getting more accurate matches despite shrinking the ratio.

Also, an increase in nfeatures leads to an increase in the final number of matches while keeping k constant.

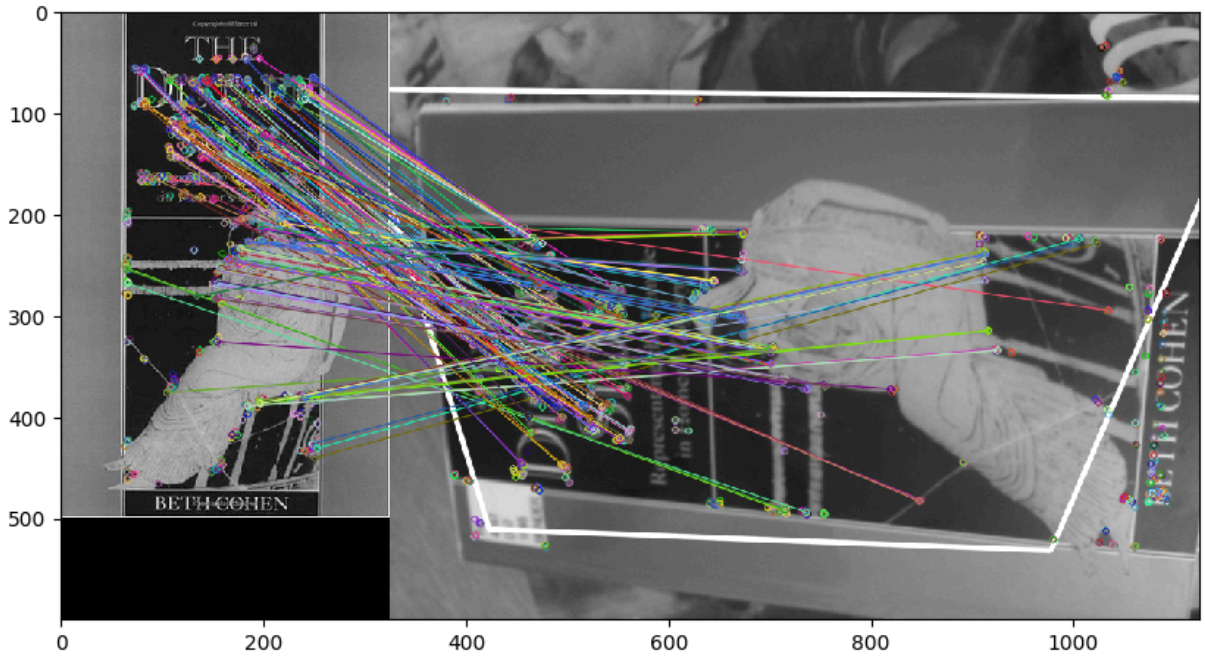
However, if the number of nfeatures set is too large, it will lead to the overlapping of some key points, which will result in unclear final matching results. So when using the ORB method to match keypoints, it should be ensured that the number of keypoints and the ratio are set appropriately. For img1 and img2, the optimal parameter sets (only the tested parameter sets) are: nfeatures=1000, ratio=0.6, and the distance calculation function using either Hamming distance or Hamming distance2 can be used.

3. Estimate a homography transformation based on the matches, using `cv2.findHomography()`. Display the transformed outline of the first reference book cover image on the query image, to see how well they match.
    - We provide a function `draw_outline()` to help with the display, but you may need to edit it for your needs.
    - Try the 'least squre method' option to compute homography, and visualize the inliers by using `cv2.drawMatches()`. Explain your results.
    - Again, you don't need to compare results numerically at this stage. Comment on what you observe visually.

In [20]:
```
# Create src_pts and dst_pts as float arrays to be passed into cv2.,findHomography
good = [m[0] for m in good_1000_hm]
src_pts = np.float32([kp1_orb_1000_hm[m.queryIdx].pt for m in good]).reshape(-1, 1,
dst_pts = np.float32([kp2_orb_1000_hm[m.trainIdx].pt for m in good]).reshape(-1, 1,

# using standard method
H, mask = cv2.findHomography(src_pts, dst_pts)
h, w = img1.shape
# img1_warped = cv2.warpPerspective(img1, H, (h_1000_hm, w_1000_hm))
pts = np.float32([[0, 0], [0, h - 1], [w - 1, h - 1], [w - 1, 0]]).reshape(-1, 1, 2
dst = cv2.perspectiveTransform(pts, H)

# draw frame
img2_with_outline = cv2.polylines(img2, [np.int32(dst)], True, 255, 3, cv2.LINE_AA)
# img2_with_outline = cv2.polylines(img2, [np.int32(dst)], True, 255, 3, cv2.LINE_A

# draw inliers
matchesMask = mask.ravel().tolist()
draw_params = dict(matchColor = (0,255,0), singlePointColor=None, matchesMask=match
img3_1000_hm = cv2.drawMatchesKnn(img1, kp1_orb_1000_hm, img2, kp2_orb_1000_hm, goo
```

```
# plt.imshow(cv2.cvtColor(img1_warped, cv2.COLOR_BGR2RGB))
plt.imshow(cv2.cvtColor(img2_with_outline, cv2.COLOR_BGR2RGB))
plt.imshow(cv2.cvtColor(img3_1000_hm, cv2.COLOR_BGR2RGB))
```

Out[20]: <matplotlib.image.AxesImage at 0x2bd8d222210>



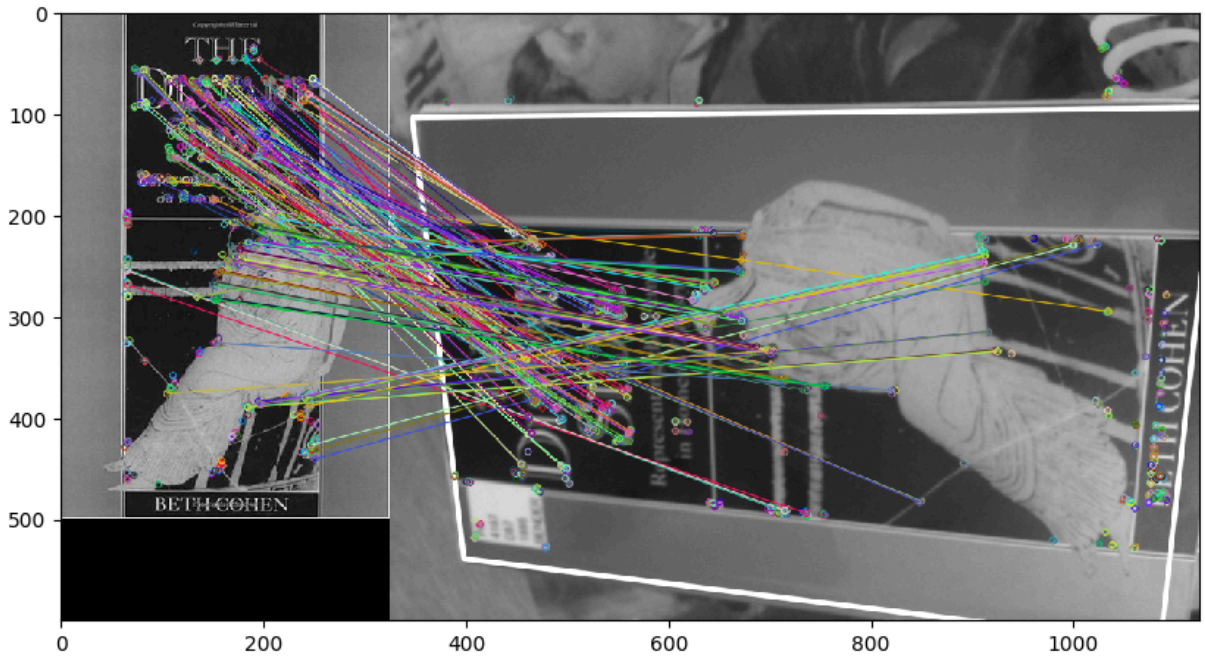
Your explanation of results here I have used the standard method for detection here, but I always don't get the correct result when drawing the border. Although drawing it on a reference does not report an error, the result I get is equally unsatisfactory: he produces an incorrect angular deviation, which is not in the reference and does not exist in the query, and I still don't get the correct result after dozens of attempts.

Try the RANSAC option to compute homography. Change the RANSAC parameters, and explain your results. Print and analyze the inlier numbers.

In [25]:
```
# Create src_pts and dst_pts as float arrays to be passed into cv2.,findHomography
good = [m[0] for m in good_1000_hm]
src_pts = np.float32([kp1_orb_1000_hm[m.queryIdx].pt for m in good]).reshape(-1, 1,
dst_pts = np.float32([kp2_orb_1000_hm[m.trainIdx].pt for m in good]).reshape(-1, 1,

# using RANSAC
H, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
h, w = img1.shape
# img1_warped = cv2.warpPerspective(img1, H, (h_1000_hm, w_1000_hm))
pts = np.float32([[0, 0], [0, h - 1], [w - 1, h - 1], [w - 1, 0]]).reshape(-1, 1, 2
dst = cv2.perspectiveTransform(pts, H)

# draw frame
img2_with_outline = cv2.polylines(img2, [np.int32(dst)], True, 255, 3, cv2.LINE_AA)
# img2_with_outline = cv2.polylines(img2, [np.int32(dst)], True, 255, 3, cv2.LINE_A

# draw inliers
```

```
matchesMask = mask.ravel().tolist()
draw_params = dict(matchColor = (0,255,0), singlePointColor=None, matchesMask=match
img3_1000_hm = cv2.drawMatchesKnn(img1, kp1_orb_1000_hm, img2, kp2_orb_1000_hm, goo

# plt.imshow(cv2.cvtColor(img1_warped, cv2.COLOR_BGR2RGB))
plt.imshow(cv2.cvtColor(img2_with_outline, cv2.COLOR_BGR2RGB))
plt.imshow(cv2.cvtColor(img3_1000_hm, cv2.COLOR_BGR2RGB))
```

Out[25]: <matplotlib.figure.AxesImage at 0x2bd8b6c6950>

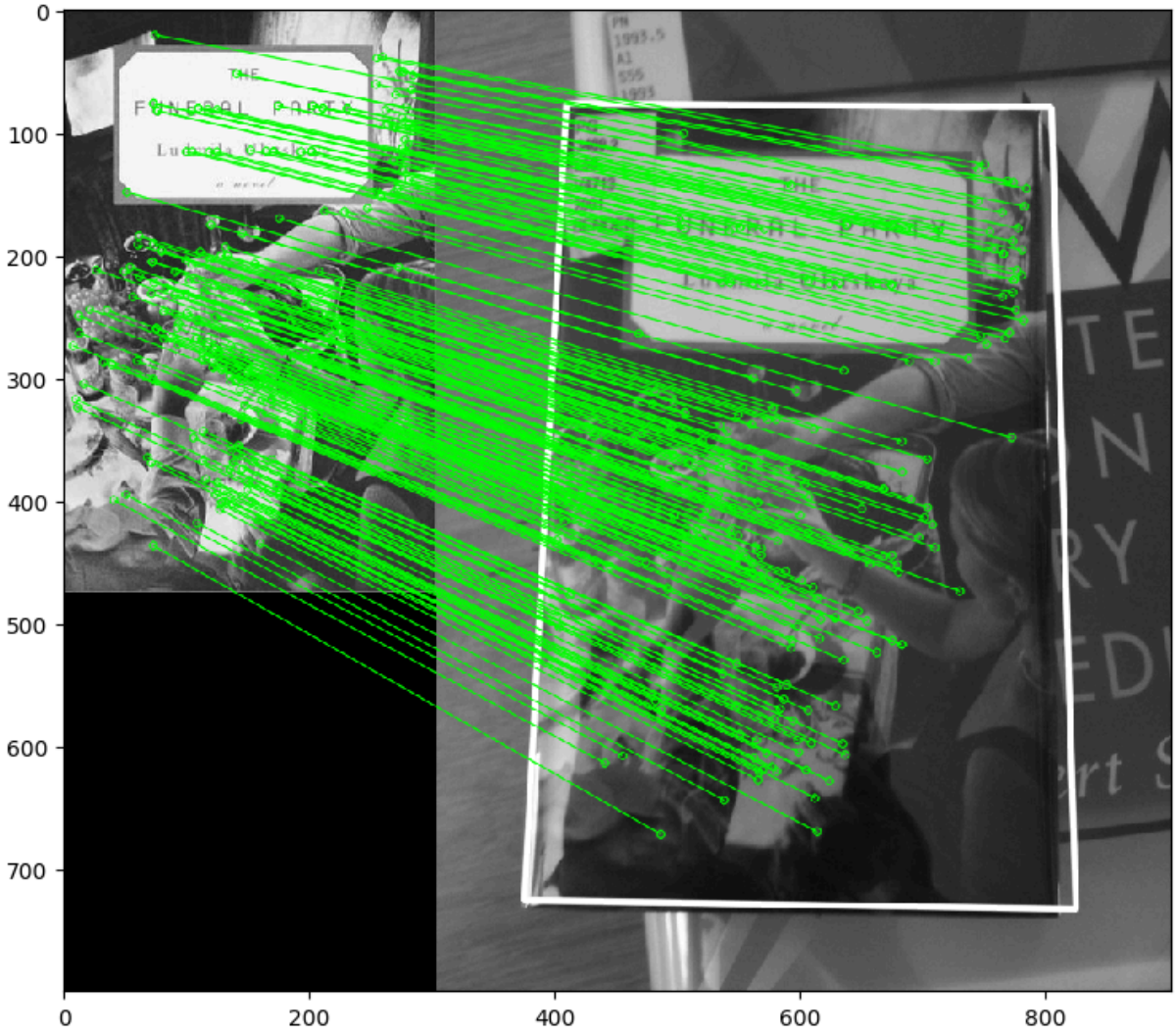


Your explanation of what you have tried, and results here After using the RANSAC method, the angle of the border obtained by plotting is only slightly shifted, but it is still not the correct border. This skewed angle is also not present in any of the plots. In my opinion: this is related to the detection of keypoints, using ORB detection does not give an accurate description of the position of the image match, and therefore does not give accurate results when performing the uni-responsive transformation.

6. Finally, try matching several different image pairs from the data provided, including at least one success and one failure case. For the failure case, test and explain what step in the feature matching has failed, and try to improve it. Display and discuss your findings.
   A. Hint 1: In general, the book covers should be the easiest to match, while the landmarks are the hardest.
   B. Hint 2: Explain why you chose each example shown, and what parameter settings were used.
   C. Hint 3: Possible failure points include the feature detector, the feature descriptor, the matching strategy, or a combination of these.

In [26]:
```
MIN_MATCH_COUNT = 10
img1 = cv2.imread(r'./a2/A2_smvs/book_covers/Reference/005.jpg', 0)
```

```
img2 = cv2.imread(r'./a2/A2_smvs/book_covers/Query/005.jpg', 0)

sift = cv2.SIFT_create()

kp1, des1 = sift.detectAndCompute(img1, None)
kp2, des2 = sift.detectAndCompute(img2, None)

FLANN_INDEX_KDTREE = 0
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)

search_params = dict(checks=50)
flann = cv2.FlannBasedMatcher(index_params, search_params)
matches = flann.knnMatch(des1, des2, k=2)

good = []
for m, n in matches:
    if m.distance < 0.7 * n.distance:
        good.append(m)
if len(good) > MIN_MATCH_COUNT:
    src_pts = np.float32([kp1[m.queryIdx].pt for m in good]).reshape(-1, 1, 2)
    dst_pts = np.float32([kp2[m.trainIdx].pt for m in good]).reshape(-1, 1, 2)
    M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
    matchesMask = mask.ravel().tolist()
    h, w = img1.shape
    pts = np.float32([[0, 0], [0, h - 1], [w - 1, h - 1], [w - 1, 0]]).reshape(-1,
    dst = cv2.perspectiveTransform(pts, M)
    img2 = cv2.polylines(img2, [np.int32(dst)], True, 255, 3, cv2.LINE_AA)
else:
    print("Not enough matches are found - %d/%d" % (len(good), MIN_MATCH_COUNT))
    matchesMask = None
draw_params = dict(matchColor=(0, 255, 0),
                   singlePointColor=None,
                   matchesMask=matchesMask,
                   flags=2)
img3 = cv2.drawMatches(img1, kp1, img2, kp2, good, None, **draw_params)
plt.imshow(img3, 'gray'), plt.show()
```

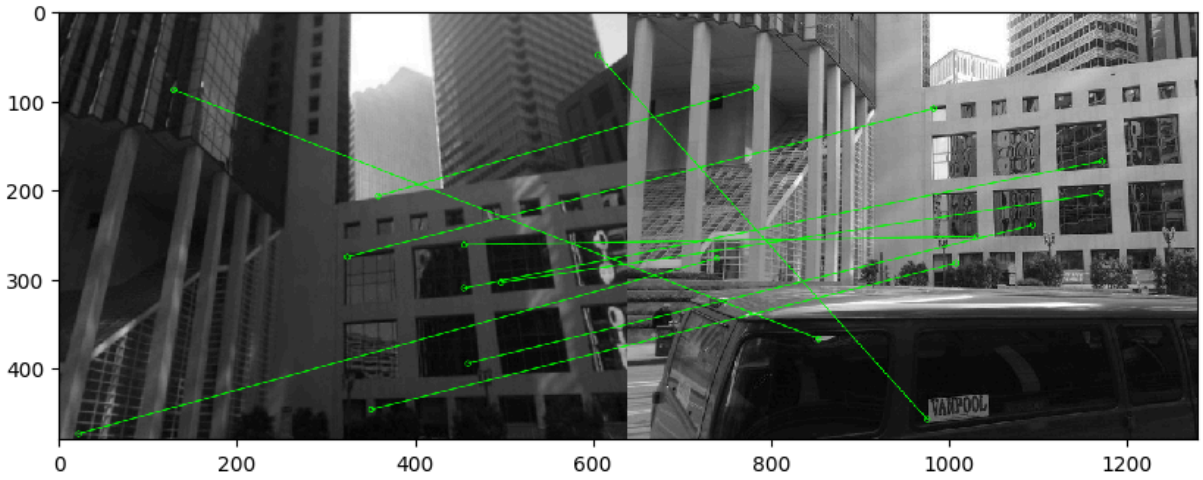


Out[26]: (<matplotlib.image.AxesImage at 0x2bd8b639390>, None)

In [27]:
```
MIN_MATCH_COUNT = 10
img1 = cv2.imread(r'./a2/A2_smvs/landmarks/Reference/002.jpg', 0)
img2 = cv2.imread(r'./a2/A2_smvs/landmarks/Query/002.jpg', 0)

sift = cv2.SIFT_create()

kp1, des1 = sift.detectAndCompute(img1, None)
kp2, des2 = sift.detectAndCompute(img2, None)

FLANN_INDEX_KDTREE = 0
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)

search_params = dict(checks=50)
flann = cv2.FlannBasedMatcher(index_params, search_params)
matches = flann.knnMatch(des1, des2, k=2)

good = []
for m, n in matches:
    if m.distance < 0.7 * n.distance:
        good.append(m)
if len(good) > MIN_MATCH_COUNT:
```

```
    src_pts = np.float32([kp1[m.queryIdx].pt for m in good]).reshape(-1, 1, 2)
    dst_pts = np.float32([kp2[m.trainIdx].pt for m in good]).reshape(-1, 1, 2)
    M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)
    matchesMask = mask.ravel().tolist()
    h, w = img1.shape
    pts = np.float32([[0, 0], [0, h - 1], [w - 1, h - 1], [w - 1, 0]]).reshape(-1,
    dst = cv2.perspectiveTransform(pts, M)
    img2 = cv2.polylines(img2, [np.int32(dst)], True, 255, 3, cv2.LINE_AA)
else:
    print("Not enough matches are found - %d/%d" % (len(good), MIN_MATCH_COUNT))
    matchesMask = None
draw_params = dict(matchColor=(0, 255, 0),
                   singlePointColor=None,
                   matchesMask=matchesMask,
                   flags=2)
img3 = cv2.drawMatches(img1, kp1, img2, kp2, good, None, **draw_params)
plt.imshow(img3, 'gray'), plt.show()
```

```
Not enough matches are found - 10/10
```




```
Out[27]: (<matplotlib.image.AxesImage at 0x2bd8b6db510>, None)
```

Your explanation of results here In this part I used SIFT transform + FLANN features for the above work, it is obvious that for the book cover, its recognition is successful, but for the building, its not only not drawn with white border, but also there are a lot of mismatches in the key points of recognition. Therefore I think that the success of image feature matching should depend on the following points:

- The problem of shooting angle. If the angle of shooting will obscure and damage some parts of the image, then an error will occur when matching, because the reference part (or call it the template part) is not damaged, it will lead to the loss of some of the corresponding keypoints in the query item, which will lead to the matching error. Such as cover 000 and landmark 002.
- Clarity. Not only in the image matching problem, the clarity of the image directly determines the success of the image algorithm. If the image is clear enough, the algorithm can identify the key adequately, but if the image is blurry, many features are masked and therefore many opportunities sufficient for matching are lost.

- Obstacles. Obstacles in this context are objects that do not appear in one of the side images and are a larger part of the other, such objects take up an entire portion of the image, thus unbalancing the weights of the locations to be matched, and the matcher will produce false matches as a result.

Question 2: What am I looking at? (40%)

In this question, the aim is to identify an "unknown" object depicted in a query image, by matching it to multiple reference images, and selecting the highest scoring match. Since we only have one reference image per object, there is at most one correct answer. This is useful for example if you want to automatically identify a book from a picture of its cover, or a painting or a geographic location from an unlabelled photograph of it.

The steps are as follows:

1. Select a set of reference images and their corresponding query images.
    A. Hint 1: Start with the book covers, or just a subset of them.
    B. Hing 2: This question can require a lot of computation to run from start to finish, so cache intermediate results (e.g. feature descriptors) where you can.
2. Choose one query image corresponding to one of your reference images. Use RANSAC to match your query image to each reference image, and count the number of inlier matches found in each case. This will be the matching score for that image.
3. Identify the query object. This is the identity of the reference image with the highest match score, or "not in dataset" if the maximum score is below a threshold.
4. Repeat steps 2-3 for every query image and report the overall accuracy of your method (that is, the percentage of query images that were correctly matched in the dataset). Discussion of results should include both overall accuracy and individual failure cases.
    A. Hint 1: In case of failure, what ranking did the actual match receive? If we used a "top-k" accuracy measure, where a match is considered correct if it appears in the top k match scores, would that change the result?

In [30]:
```
# Your code to identify query objects and measure search accuracy for data set here
import os
import random
from a2code import get_match_score, get_best_matches

threshold = 100
```

```
# query = cv2.imread(r'./a2/A2_smvs/book_covers/Query/005.jpg', 0)
query_path = r'./a2/A2_smvs/book_covers/Query/'
ref_path = r'./a2/A2_smvs/book_covers/Reference/'
refs, scores_index = [], []

random_files = random.sample(os.listdir(query_path), 10)
queries_index = [int(e.split('.')[0])-1 for e in random_files]

for file in os.listdir(ref_path):
    refs.append(cv2.imread(ref_path + file, 0))

for file in random_files:
    query = cv2.imread(query_path + file, 0)
    scores_index.append(get_best_matches(get_match_score(query, refs), threshold))
```

```
100%|██████████| 101/101 [01:02<00:00,  1.63it/s]
100%|██████████| 101/101 [00:14<00:00,  7.00it/s]
100%|██████████| 101/101 [00:16<00:00,  6.03it/s]
100%|██████████| 101/101 [01:29<00:00,  1.13it/s]
100%|██████████| 101/101 [00:14<00:00,  6.78it/s]
100%|██████████| 101/101 [02:59<00:00,  1.78s/it]
100%|██████████| 101/101 [02:05<00:00,  1.24s/it]
100%|██████████| 101/101 [01:06<00:00,  1.51it/s]
100%|██████████| 101/101 [01:03<00:00,  1.59it/s]
100%|██████████| 101/101 [00:15<00:00,  6.45it/s]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[30], line 23
     19     query = cv2.imread(query_path + file, 0)
     20     scores_index.append(get_best_matches(get_match_score(query, refs), thres
hold))
---> 23 print(f'Accuracy: {sum(p == l for p, l in zip(queries_index, scores_index) /
len(queries_index))}')

TypeError: unsupported operand type(s) for /: 'zip' and 'int'
```

In [48]:
```
get_index = [e[1] if len(e) == 2 else -1 for e in scores_index]
print(get_index)
print(queries_index)
print(f'Accuracy: {sum(p == l for p, l in zip(queries_index, get_index)) / len(quer
```

```
[100, 42, 12, 55, 33, 74, 79, 99, -1, 48]
[100, 42, 12, 55, 33, 74, 79, 99, 68, 48]
Accuracy: 0.9
```

Your explanation of what you have done, and your results, here I've written two functions in a2code.py that are used to calculate the fraction of interior points that match, versus finding the location of the best matching image. Among them:

- get_match_score() is used to calculate the interior point score, and since the question did not require which algorithm to use, I used the same FLANN feature + SIFT detection method from the previous question
- get_best_matches() is used to find the subscript of the best matching image, where threshold I set to 100, which means that only matches with an inner point

count greater than or equal to 100 are considered to be matched.

Then, I randomly took 10 query images and used all the book covers as refs. I used tqdm to observe the progress of matching for each image, and it can be noticed that: not all images have the same speed of matching computation, it can be up to 7 it/s in the faster speed, but in many cases it is a slower match with a speed of 1.5 it/s or so.

Eventually, I saved the positions of the obtained matches and compared them with their original positions as a way of judging whether the best matching images obtained were as expected. It turned out that Figure 68 was not queried for the matched image, but the reference image for 68 was indeed in the list. After observing the 68 images, I concluded that: it could be caused by too much blurring of the shot, or too much difference in angle.

My program has an accuracy of 0.9, but the decision making is slow.

5. Choose some extra query images of objects that do not occur in the reference dataset. Repeat step 4 with these images added to your query set. Accuracy is now measured by the percentage of query images correctly identified in the dataset, or correctly identified as not occurring in the dataset. Report how accuracy is altered by including these queries, and any changes you have made to improve performance.

In [49]:
```
# Your code to run extra queries and display results here
random_files_2 = random.sample(os.listdir(query_path), 5)
queries_index += [int(e.split('.')[0])-1 for e in random_files_2]
queries_index += [-1, -1]
query_landmark = cv2.imread(r'./a2/A2_smvs/landmarks/Query/100.jpg', 0)
query_museum = cv2.imread(r'./a2/A2_smvs/museum_paintings/Query/033.jpg', 0)

for file in random_files_2:
    query = cv2.imread(query_path + file, 0)
    scores_index.append(get_best_matches(get_match_score(query, refs), threshold))
scores_index.append(get_best_matches(get_match_score(query_landmark, refs), thresho
scores_index.append(get_best_matches(get_match_score(query_museum, refs), threshold
```

```
100%|██████████| 101/101 [01:27<00:00,  1.16it/s]
100%|██████████| 101/101 [01:23<00:00,  1.21it/s]
100%|██████████| 101/101 [00:11<00:00,  8.60it/s]
100%|██████████| 101/101 [00:13<00:00,  7.48it/s]
100%|██████████| 101/101 [01:18<00:00,  1.28it/s]
100%|██████████| 101/101 [00:19<00:00,  5.23it/s]
100%|██████████| 101/101 [00:27<00:00,  3.64it/s]
```

In [50]:
```
get_index = [e[1] if len(e) == 2 else -1 for e in scores_index]
print(get_index)
print(queries_index)
print(f'Accuracy: {sum(p == l for p, l in zip(queries_index, get_index)) / len(quer
```

```
[100, 42, 12, 55, 33, 74, 79, 99, -1, 48, 58, -1, -1, 40, 63, -1, -1]
[100, 42, 12, 55, 33, 74, 79, 99, 68, 48, 58, 94, 25, 40, 63, -1, -1]
Accuracy: 0.8235294117647058
```

Your explanation of results and any changes made here I re-randomized 5 more book covers and randomly (no program was written here, but rather artificially randomized) selected two images from Landmark and museum_paintings, respectively, which the program can identify well enough to know if they are from the dataset, since -1 is used to indicate that they don't exist in the original dataset. However, Figures 94 and 25 are still not accurately recognized, for the same possible reasons as above, or the THRESHOLD is too large and too few keypoints were originally recognized, leading to misclassification.

6. Repeat step 4 and 5 for at least one other set of reference images from museum_paintings or landmarks, and compare the accuracy obtained. Analyse both your overall result and individual image matches to diagnose where problems are occurring, and what you could do to improve performance. Test at least one of your proposed improvements and report its effect on accuracy.

In [54]:
```
# Your code to search images and display results here
landmark_ref_path = r'./a2/A2_smvs/landmarks/Reference/'
landmark_query_path = r'./a2/A2_smvs/museum_paintings/Reference/'
landmark_refs, scores_index = [], []

landmark_random_files = random.sample(os.listdir(landmark_query_path), 10)
queries_index = [int(e.split('.')[0])-1 for e in random_files]

for file in os.listdir(landmark_ref_path):
    landmark_refs.append(cv2.imread(landmark_ref_path + file, 0))

for file in landmark_random_files:
    query = cv2.imread(landmark_query_path + file, 0)
    scores_index.append(get_best_matches(get_match_score(query, landmark_refs), thr
    get_index = [e[1] if len(e) == 2 else -1 for e in scores_index]
print(get_index)
print(queries_index)
print(f'Accuracy: {sum(p == l for p, l in zip(queries_index, get_index)) / len(quer
```

```
100%|██████████| 101/101 [00:09<00:00, 11.15it/s]
100%|██████████| 101/101 [00:43<00:00,  2.30it/s]
100%|██████████| 101/101 [00:09<00:00, 11.20it/s]
100%|██████████| 101/101 [00:44<00:00,  2.29it/s]
100%|██████████| 101/101 [00:31<00:00,  3.17it/s]
100%|██████████| 101/101 [00:39<00:00,  2.55it/s]
100%|██████████| 101/101 [00:48<00:00,  2.08it/s]
100%|██████████| 101/101 [01:18<00:00,  1.29it/s]
100%|██████████| 101/101 [00:29<00:00,  3.45it/s]
100%|██████████| 101/101 [00:10<00:00,  9.71it/s]
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
[90, 63, 78, 83, 74, 48, 65, 50, 60, 73]
Accuracy: 0.0
```

In [56]:
```
# Your code to search images and display results here
paintings_ref_path = r'./a2/A2_smvs/landmarks/Reference/'
paintings_query_path = r'./a2/A2_smvs/museum_paintings/Reference/'
paintings_refs, scores_index = [], []

paintings_random_files = random.sample(os.listdir(paintings_query_path), 10)
queries_index = [int(e.split('.')[0])-1 for e in random_files]

for file in os.listdir(paintings_ref_path):
    paintings_refs.append(cv2.imread(paintings_ref_path + file, 0))

for file in paintings_random_files:
    query = cv2.imread(paintings_query_path + file, 0)
    scores_index.append(get_best_matches(get_match_score(query, paintings_refs), th

get_index = [e[1] if len(e) == 2 else -1 for e in scores_index]
print(get_index)
print(queries_index)
print(f'Accuracy: {sum(p == l for p, l in zip(queries_index, get_index)) / len(quer
```

```
100%|██████████| 101/101 [00:13<00:00,  7.24it/s]
100%|██████████| 101/101 [00:16<00:00,  6.15it/s]
100%|██████████| 101/101 [00:55<00:00,  1.81it/s]
100%|██████████| 101/101 [00:16<00:00,  6.18it/s]
100%|██████████| 101/101 [00:20<00:00,  4.83it/s]
100%|██████████| 101/101 [00:07<00:00, 12.66it/s]
100%|██████████| 101/101 [01:17<00:00,  1.30it/s]
100%|██████████| 101/101 [00:30<00:00,  3.29it/s]
100%|██████████| 101/101 [01:08<00:00,  1.48it/s]
100%|██████████| 101/101 [01:00<00:00,  1.68it/s]
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
[90, 63, 78, 83, 74, 48, 65, 50, 60, 73]
Accuracy: 0.0
```

In [62]:
```
from a2code import get_best_matches_with_topk

# Your code to search images and display results here

kernel = np.ones((5, 5), np.float32) / 25

paintings_ref_path = r'./a2/A2_smvs/landmarks/Reference/'
paintings_query_path = r'./a2/A2_smvs/museum_paintings/Reference/'
paintings_refs, scores_index = [], []

paintings_random_files = random.sample(os.listdir(paintings_query_path), 10)
queries_index = [int(e.split('.')[0])-1 for e in random_files]

for file in os.listdir(paintings_ref_path):
    paintings_refs.append(cv2.imread(paintings_ref_path + file, 0))

for file in paintings_random_files:
    query = cv2.imread(paintings_query_path + file, 0)
    scores_index.append(get_best_matches_with_topk(get_match_score(query, paintings

get_index = [e[1] if len(e) == 2 else -1 for e in scores_index]
```

```
res = []
for i in range(len(scores_index)):
    if get_index[i] in scores_index[i]:
        res.append(1)
    else:
        res.append(-1)

print(f'Accuracy: {sum(p == l for p, l in zip(res, [1]*len(scores_index))) / len(qu
```

```
100%|██████████| 101/101 [01:00<00:00,  1.67it/s]
100%|██████████| 101/101 [00:29<00:00,  3.38it/s]
100%|██████████| 101/101 [00:16<00:00,  6.06it/s]
100%|██████████| 101/101 [00:07<00:00, 14.33it/s]
100%|██████████| 101/101 [00:46<00:00,  2.19it/s]
100%|██████████| 101/101 [00:16<00:00,  6.02it/s]
100%|██████████| 101/101 [00:58<00:00,  1.73it/s]
100%|██████████| 101/101 [00:08<00:00, 12.57it/s]
100%|██████████| 101/101 [00:16<00:00,  6.09it/s]
100%|██████████| 101/101 [00:35<00:00,  2.86it/s]
Accuracy: 0.0
```

Your description of what you have done, and explanation of results, here I did the same for landmark and paintings as I did when checking book cover references against the query sequence and found 0 accuracy.

At first I thought there was something wrong with my algorithm, so I checked the algorithm half a dozen times, but did not find anything unusual. As I examined the data I realized that the images had some of the same characteristics:

- Shooting angle. The shooting angle here is not like the cover of a book, which is rotated at a random angle. The shooting angle of these two datasets makes the query image retain only a part of the refs, and there are a lot of "obstacles" mentioned before, so it is difficult to recognize them.
- Color. It may be due to the difference in shooting equipment, fill light, exposure conditions, etc., resulting in obvious differences in the color tones of the refs and query, even though it is read in the form of a grayscale map, it will still fail to be recognized.
- Clarity. Clarity is the most important, some query is very fuzzy, key point detection problems, so can not correctly image matching.

Generally it is after image preprocessing and then matching will get better results, but after I tried it still could not achieve the result, so here I used a simple method to try if I can improve the performance: use top_k algorithm, as long as the subscripts have appeared in these k elements, it proves that the recognition is successful, and set k to a larger value. But still no improvement is obtained, supposedly because the k value is still too small, or the randomly selected image itself is difficult to match with the original image.

In [ ]: