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 [1]:
```
# from google.colab import drive
# drive.mount('/content/drive')
```

In [2]:
```
# !pwd
# %cd drive/MyDrive/Colab\ Notebooks
# !pwd
```

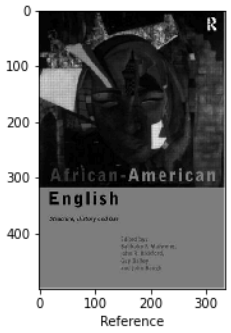
In [3]:
```
# Numpy is the main package for scientific computing with Python.
import numpy as np
import cv2

# 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,
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

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

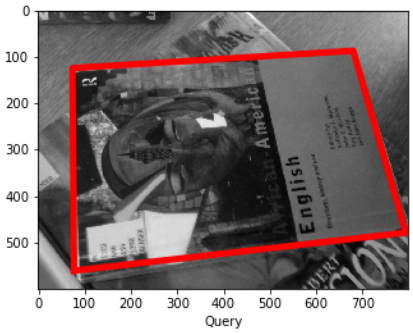
In [4]:
```
def draw_outline(ref, query, model):
    """
        Draw outline of reference image in the query image.
        This is just an example to show the steps involved.
        You can modify to suit your needs.
        Inputs:
            ref: reference image
            query: query image
```

```
            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 co
                       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()

def drawlines(img1,img2,lines,pts1,pts2):
    ''' img1 - image on which we draw the epipolar lines
        img2 - image where the points are defined for visualizing epiline
        pts1, pts2 are your good matches in image 1 and 2.
        lines - corresponding epilines '''
    r,c = img1.shape
    img1 = cv2.cvtColor(img1,cv2.COLOR_GRAY2BGR)
    img2 = cv2.cvtColor(img2,cv2.COLOR_GRAY2BGR)
    for r,pt1,pt2 in zip(lines,pts1,pts2):
        color = tuple(np.random.randint(0,255,3).tolist())
        x0,y0 = map(int, [0, -r[2]/r[1] ])
        x1,y1 = map(int, [c, -(r[2]+r[0]*c)/r[1] ])
        img1 = cv2.line(img1, (x0,y0), (x1,y1), color,1)
        img1 = cv2.circle(img1,tuple(pt1),5,color,-1)
        img2 = cv2.circle(img2,tuple(pt2),5,color,-1)
    return img1,img2
```

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:





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 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.

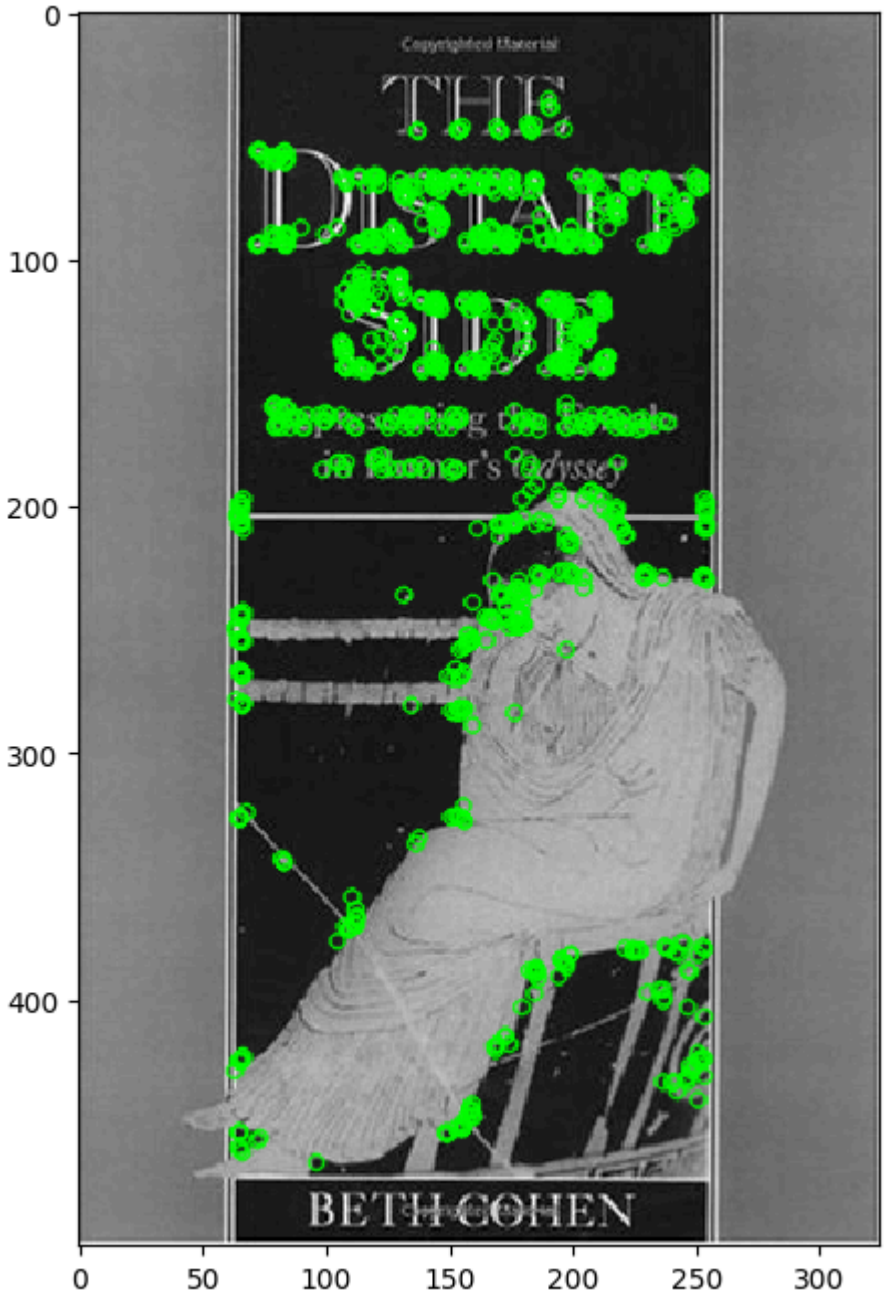
3. [Match features] As ORB is a binary feature, you need to use HAMMING distance for matching, e.g., `bf = cv2.BFMatcher(cv2.NORM_HAMMING)`. Then you are requried to do KNN matching (k=2) by using `bf.knnMatch()`. After that, you are required to use "ratio_test". Ratio test was used in SIFT to find good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

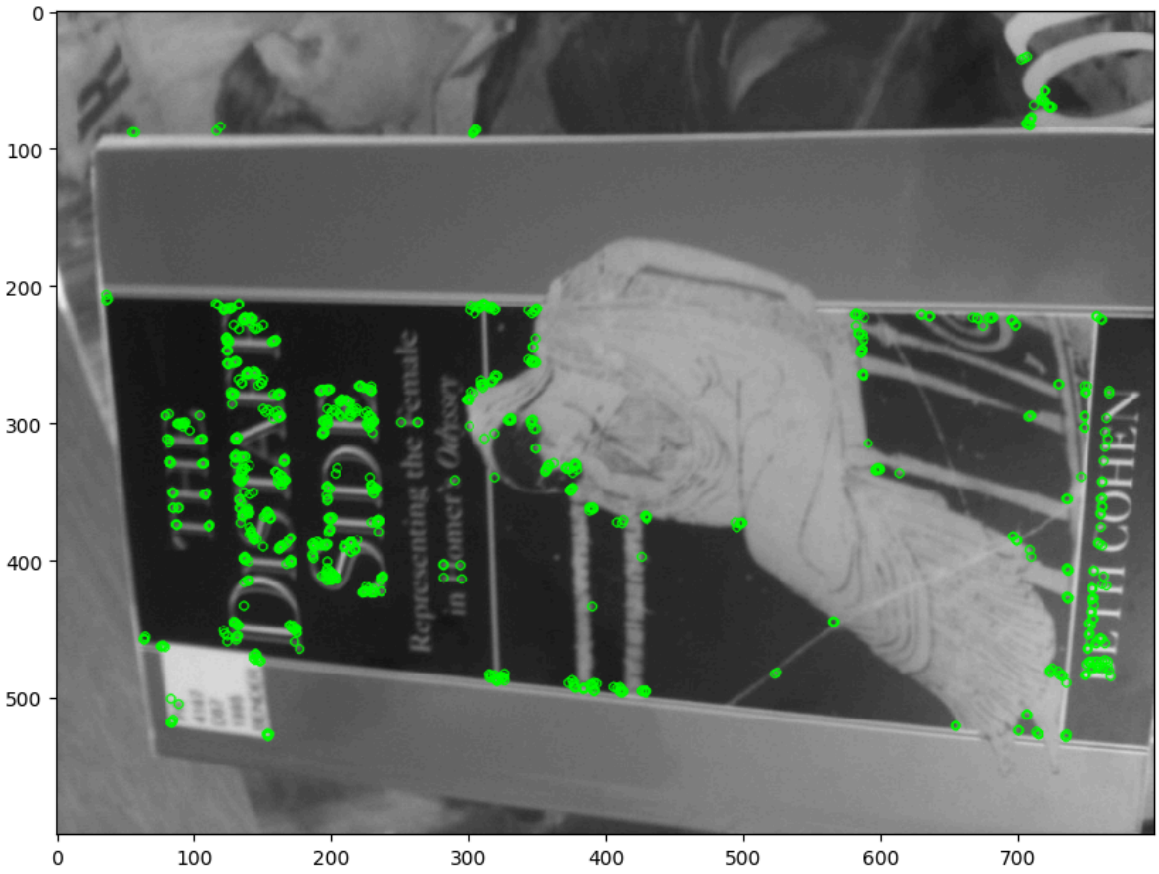
4. [Plot and analyze] You need to visualize the matches by using the `cv2.drawMatches()` function. Also you can change the ratio values, parameters in `cv2.ORB_create()`, and distance functions in `cv2.BFMatcher()`. Please discuss how these changes influence the match numbers.

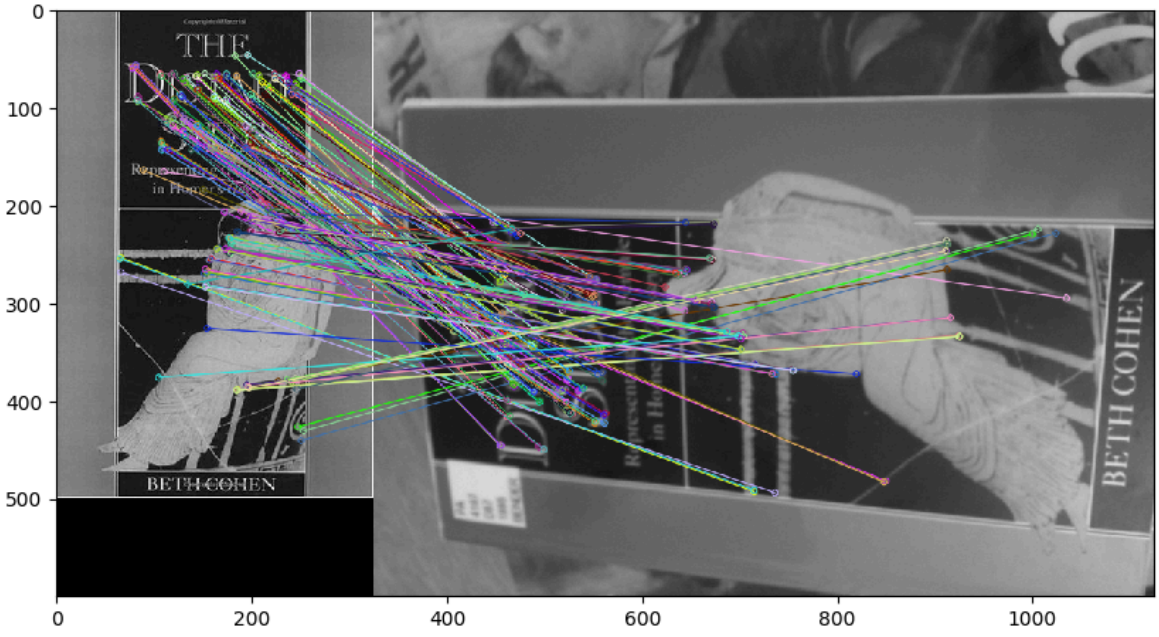
In [5]:
```
# load images as grey scale
img1 = cv2.imread('A2_smvs/book_covers/Reference/001.jpg', 0)
if not np.shape(img1):
  # Error message and print current working dir
  print("Could not load img1. Check the path, filename and current workin
  !pwd
img2 = cv2.imread("A2_smvs/book_covers/Query/001.jpg", 0)
if not np.shape(img2):
  # Error message and print current working dir
  print("Could not load img2. Check the path, filename and current workin
  !pwd
```

In [6]:

```python
# Your code for descriptor matching tests here

# compute detector and descriptor, see (2) above
orb = cv2.ORB_create(nfeatures=1000) # create ORB detector
kp_1 = orb.detect(img1, None) # detect keypoints in img1
kp_2 = orb.detect(img2, None) # detect keypoints in img2

# find the keypoints and descriptors with ORB, see (2) above
kp_1, des_1 = orb.compute(img1, kp_1) # compute descriptors for img1
kp_2, des_2 = orb.compute(img2, kp_2) # compute descriptors for img2

# draw keypoints, see (2) above
kp1_draw = cv2.drawKeypoints(img1, kp_1, img1, color=(0,255,0), flags=0)
kp2_draw = cv2.drawKeypoints(img2, kp_2, img2, color=(0,255,0), flags=0)
plt.imshow(kp1_draw), plt.show()
plt.imshow(kp2_draw), plt.show()

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

# Match descriptors, see (3) above
matches = bf.knnMatch(des_1, des_2, k=2) # find knn matches

# Apply ratio test, see (3) above
good = []
ratio = 0.8
for m,n in matches:
    if m.distance < ratio * n.distance:
        good.append(m) # append good matches to list


# draw matches, see (4) above
img_matches = cv2.drawMatches(img1, kp_1, img2, kp_2, good, None, flags=c
plt.imshow(img_matches), plt.show()

# count good matches
print('Number of good matches: ', len(good))
```


Copyrighted Material
THE
DISTANT
SIDE
BETH COHEN
0
100
200
300
400
0
50
100
150
200
250
300



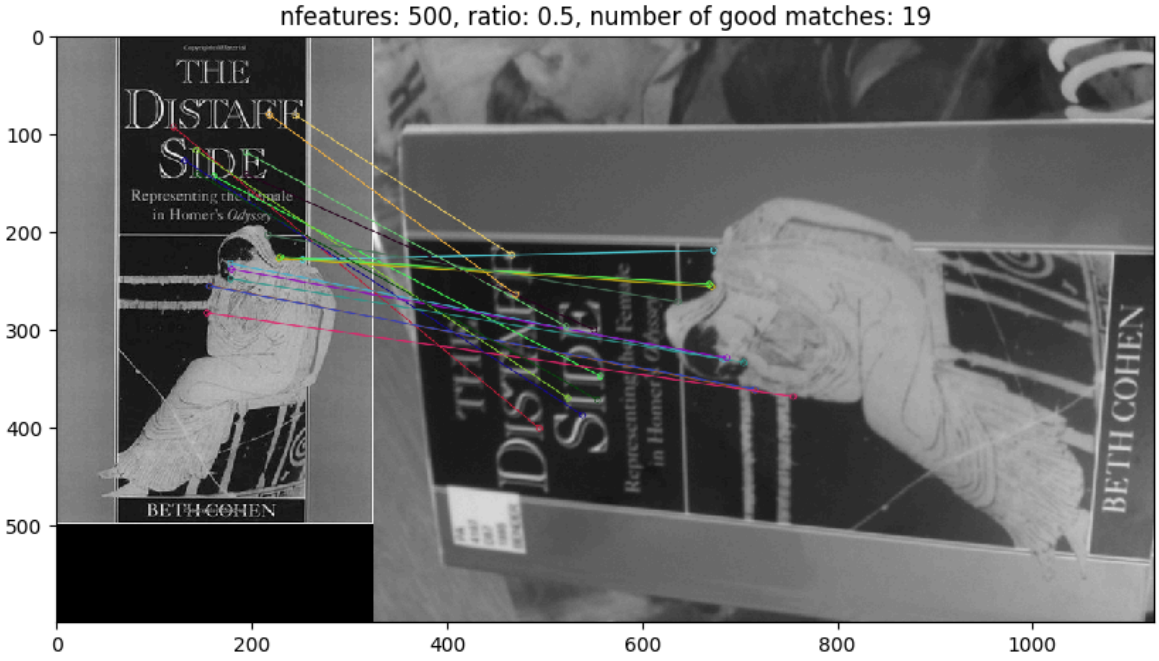


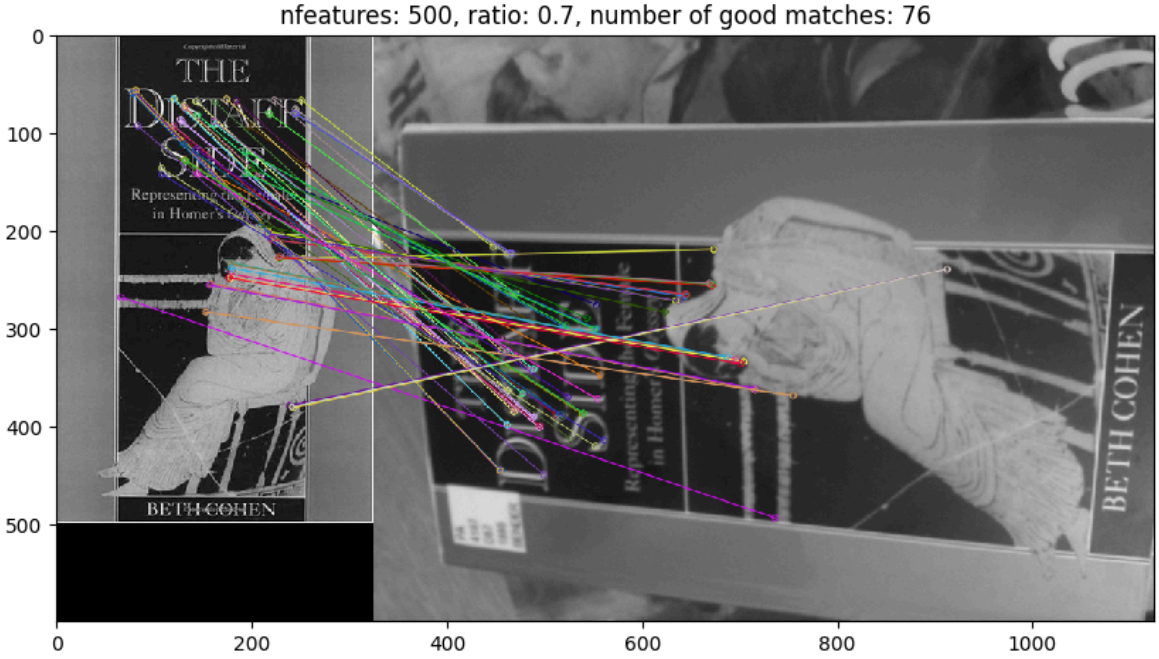

```
Number of good matches:  221
```

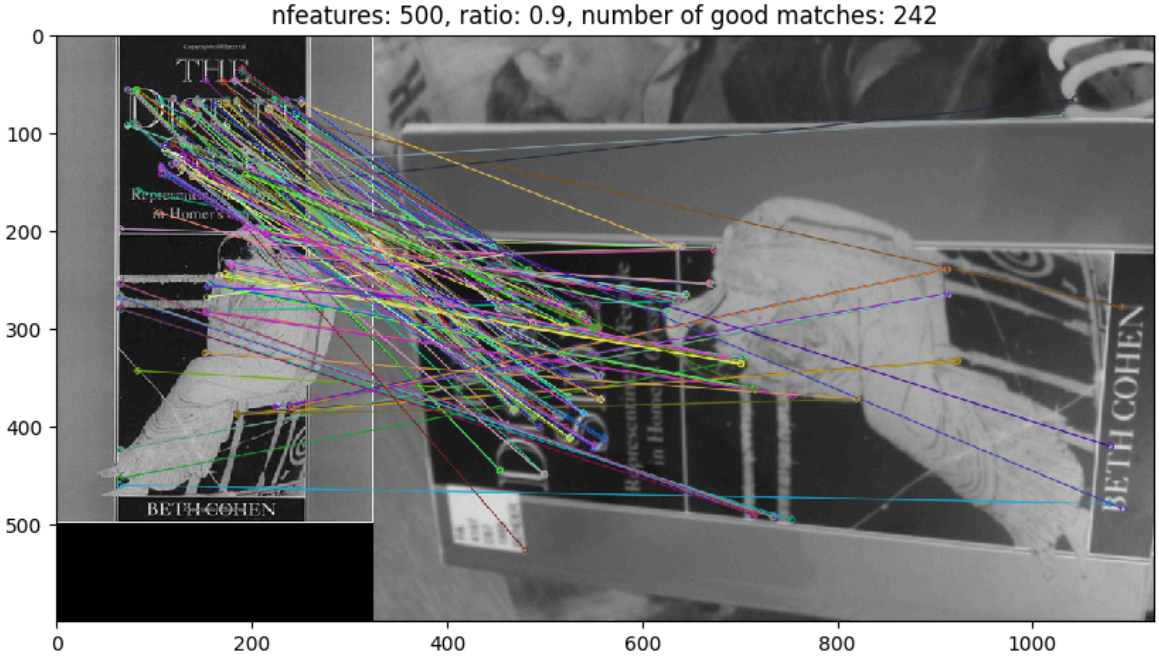
In [7]:

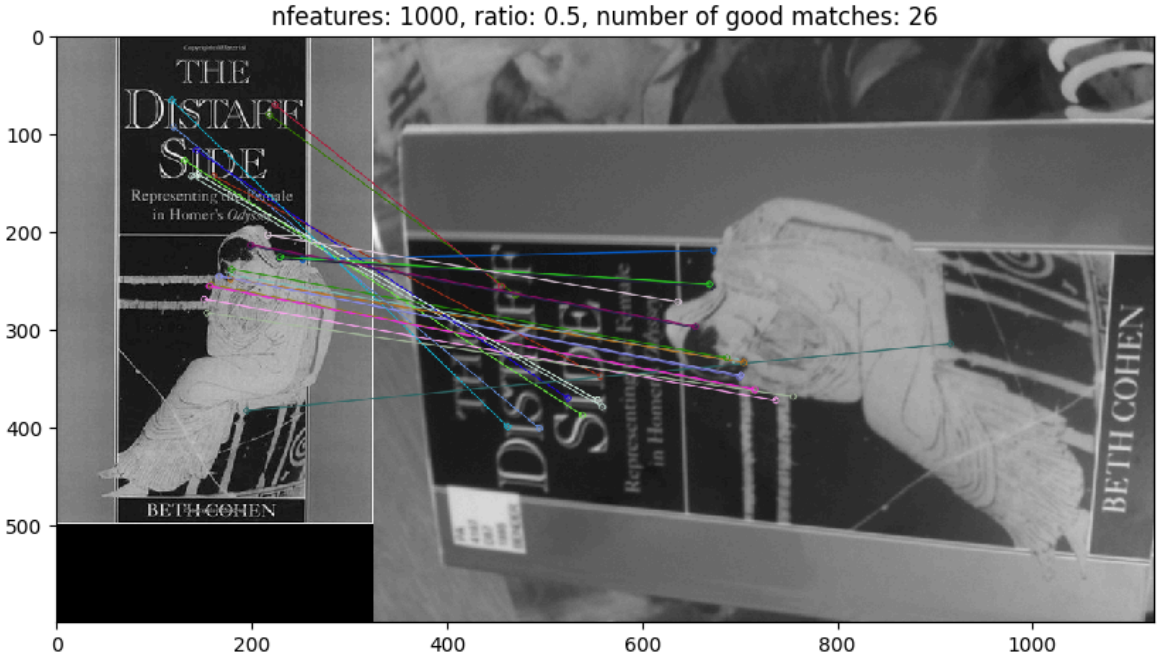
```python
import pandas as pd
import seaborn as sns

# experiment with different values
nfeatures = [500, 1000, 1500]
ratios = [0.5, 0.6, 0.7, 0.8, 0.9]
img1 = cv2.imread('A2_smvs/book_covers/Reference/001.jpg', 0)
img2 = cv2.imread("A2_smvs/book_covers/Query/001.jpg", 0)

test_results = []
# experiment with different values enumerate
for nfeature in nfeatures:
```

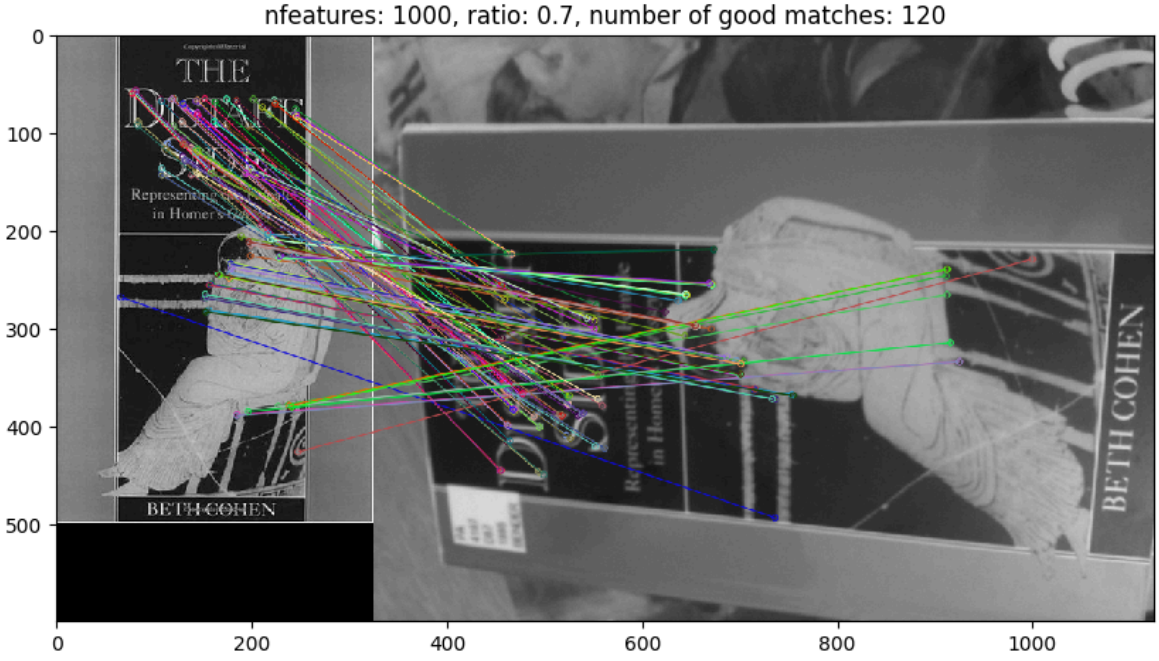
```
    orb = cv2.ORB_create(nfeatures=nfeature) # create ORB detector
    kp_1 = orb.detect(img1, None) # detect keypoints in img1
    kp_2 = orb.detect(img2, None) # detect keypoints in img2

    # find the keypoints and descriptors with ORB, see (2) above
    kp_1, des_1 = orb.compute(img1, kp_1) # compute descriptors for img1
    kp_2, des_2 = orb.compute(img2, kp_2) # compute descriptors for img2

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

    # Match descriptors, see (3) above
    matches = bf.knnMatch(des_1, des_2, k=2) # find knn matches

    # Apply ratio test, see (3) above
    for ratio in ratios:
        good = []
        for m,n in matches:
            if m.distance < ratio * n.distance:
                good.append(m)

        # save results
        test_results.append({
            'nfeatures': nfeature,
            'ratio': ratio,
            'good_matches': len(good)
        })

        # draw matches for 0.5 0.7 0.9
        if ratio in [0.5, 0.7, 0.9]:
            img_matches = cv2.drawMatches(img1, kp_1, img2, kp_2, good, N
            plt.imshow(img_matches), plt.title(f"nfeatures: {nfeature}, r



df = pd.DataFrame(test_results)
plt.figure(figsize=(12, 6))
sns.lineplot(data=df, x='ratio', y='good_matches', style='nfeatures', mar
plt.title("Effect of Ratio and nfeatures on Good Matches")
plt.grid(True)
```

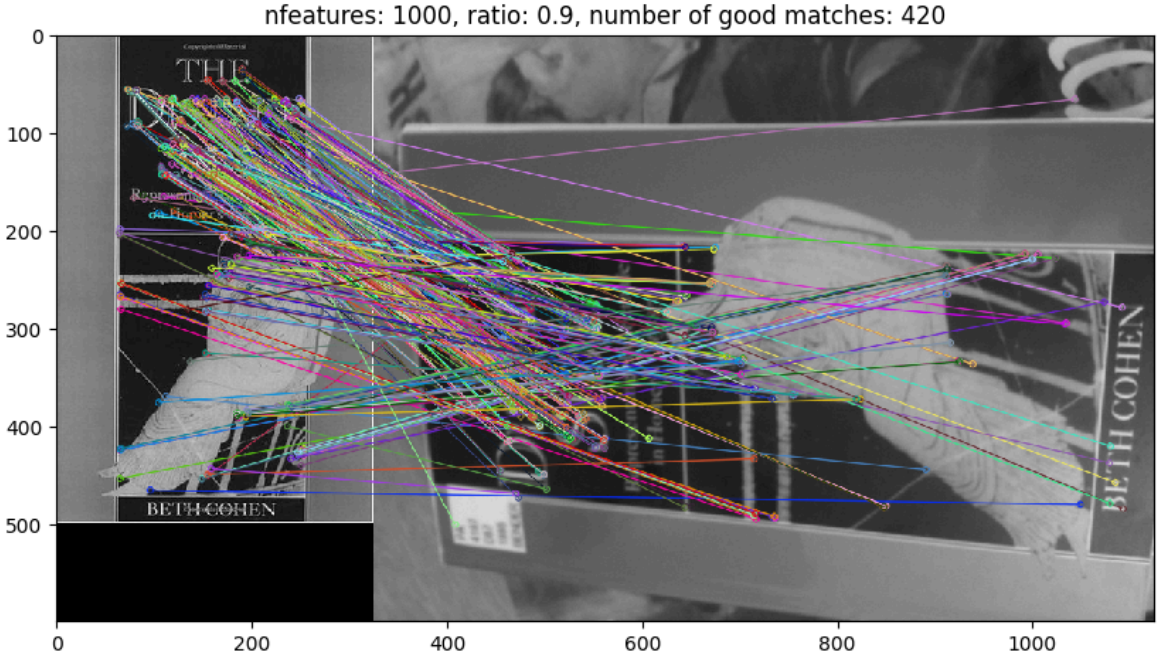

nfeatures: 500, ratio: 0.5, number of good matches: 19
THE DISTAFF SIDE
Representing the Female in Homer's Odyssey
BETH COHEN
0
100
200
300
400
500
0
200
400
600
800
1000

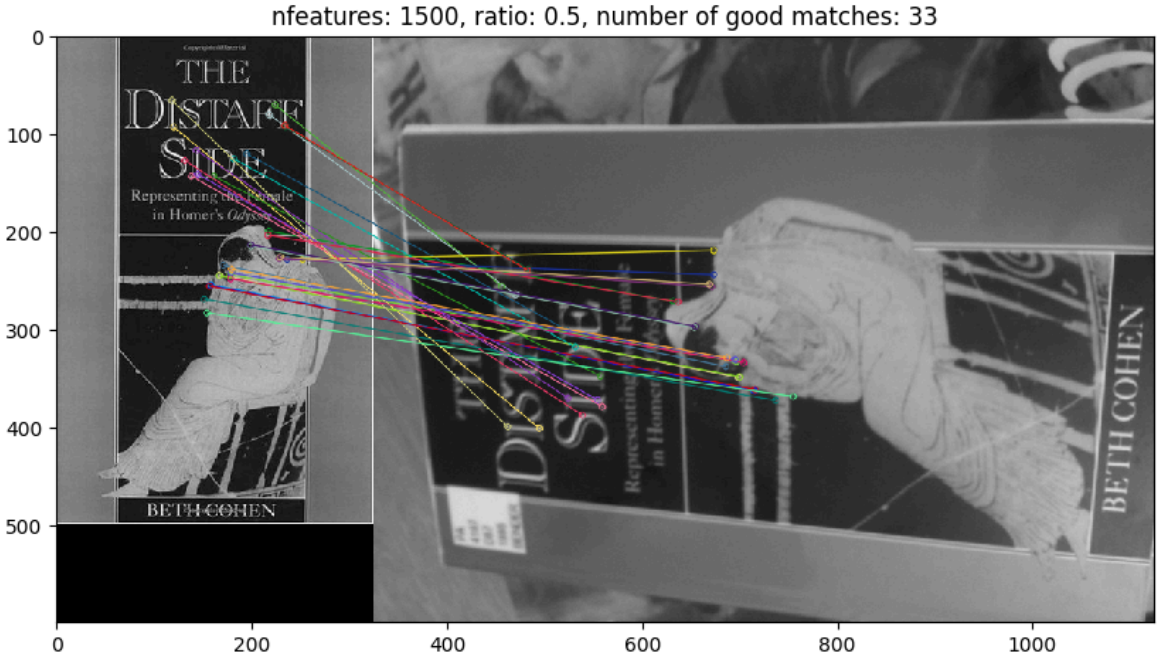

nfeatures: 500, ratio: 0.7, number of good matches: 76
THE DISTAFF SIDE
BETH COHEN
0
100
200
300
400
500
0
200
400
600
800
1000

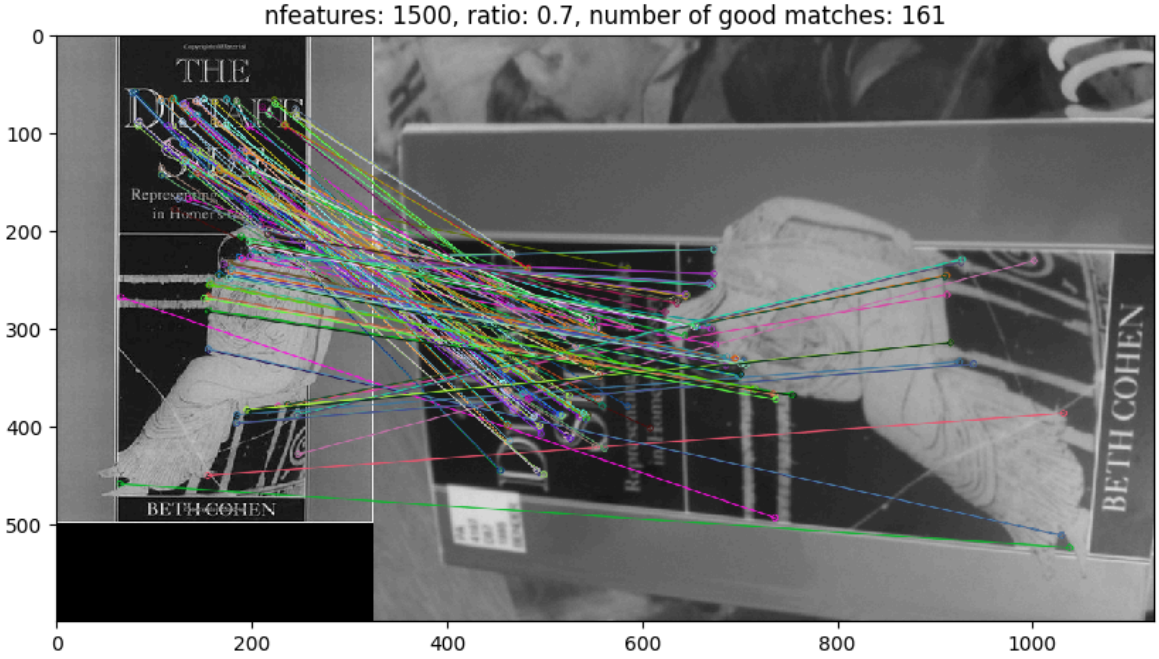

nfeatures: 500, ratio: 0.9, number of good matches: 242
THE DISTAFF SIDE
BETH COHEN
0
100
200
300
400
500
0
200
400
600
800
1000

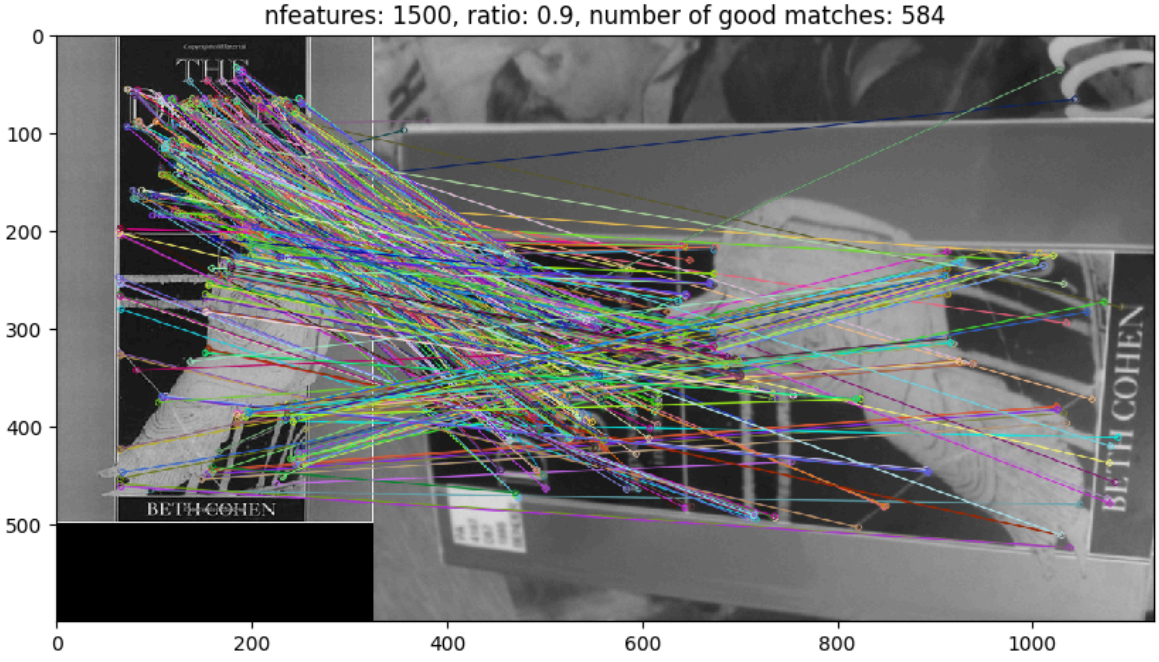

nfeatures: 1000, ratio: 0.5, number of good matches: 26
0
100
200
300
400
500
0
200
400
600
800
1000
THE DISTAFF SIDE
BETH COHEN

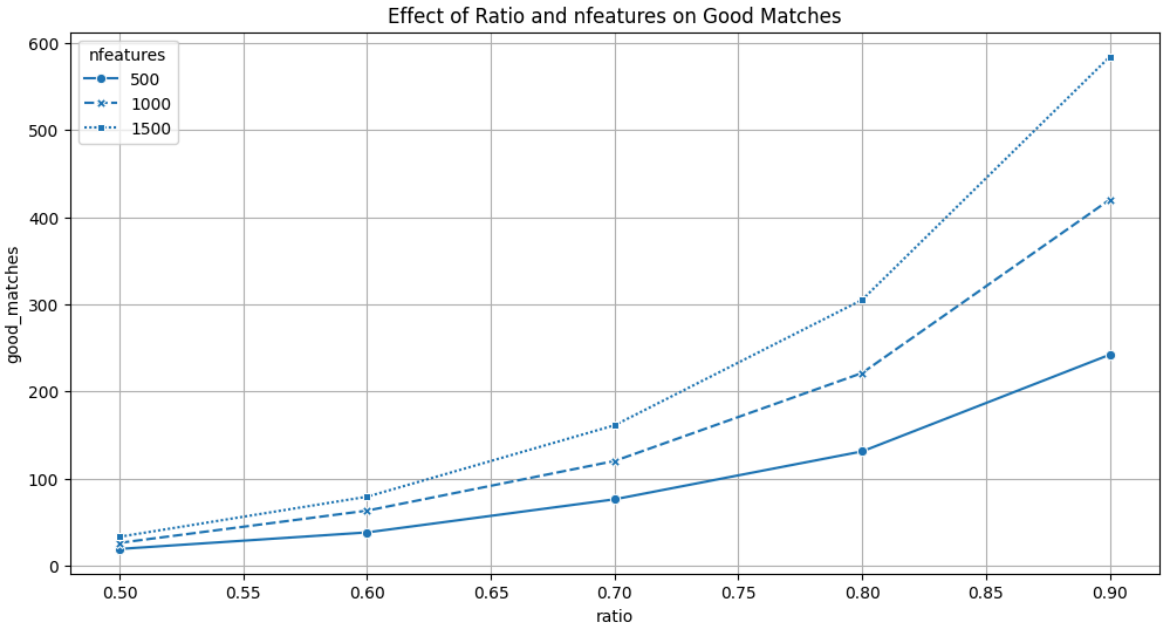

nfeatures: 1000, ratio: 0.7, number of good matches: 120
0
100
200
300
400
500
0
200
400
600
800
1000
BETH COHEN


nfeatures: 1000, ratio: 0.9, number of good matches: 420
0
100
200
300
400
500
0
200
400
600
800
1000
BETH COHEN


nfeatures: 1500, ratio: 0.5, number of good matches: 33
0
100
200
300
400
500
0
200
400
600
800
1000
THE DISTAFF SIDE
BETH COHEN


nfeatures: 1500, ratio: 0.7, number of good matches: 161
0
100
200
300
400
500
0
200
400
600
800
1000
THE
BETH COHEN


nfeatures: 1500, ratio: 0.9, number of good matches: 584
0
100
200
300
400
500
0
200
400
600
800
1000
BETH COHEN



Your explanation of what you have done, and your results, here

Step 1-4 discussion

Feature Detection and Description

- Images were loaded in grayscale, and ORB was used to detect keypoints and compute binary descriptors.
- Keypoints were visualized with cv2.drawKeypoints() to confirm good spatial coverage.

Descriptor Matching and Ratio Test

- Matching was performed using cv2.BFMatcher with Hamming distance, suitable for ORB descriptors.
- KNN matching (k=2) was followed by Lowe's ratio test (default 0.8) to filter ambiguous matches.
- Good matches were visualized using cv2.drawMatches() to verify correct alignment.

Parameter Experiments

- I varied nfeatures (500, 1000, 1500) and ratio thresholds (0.5–0.9) to evaluate match performance.
- Results were recorded and visualized using Seaborn line plots for comparison.

Findings:

- Higher nfeatures led to more keypoints and matches.
- Lower ratios (e.g., 0.5) produced fewer but more accurate matches.
- Ratios of 0.7–0.8 offered a good balance of match count and reliability.

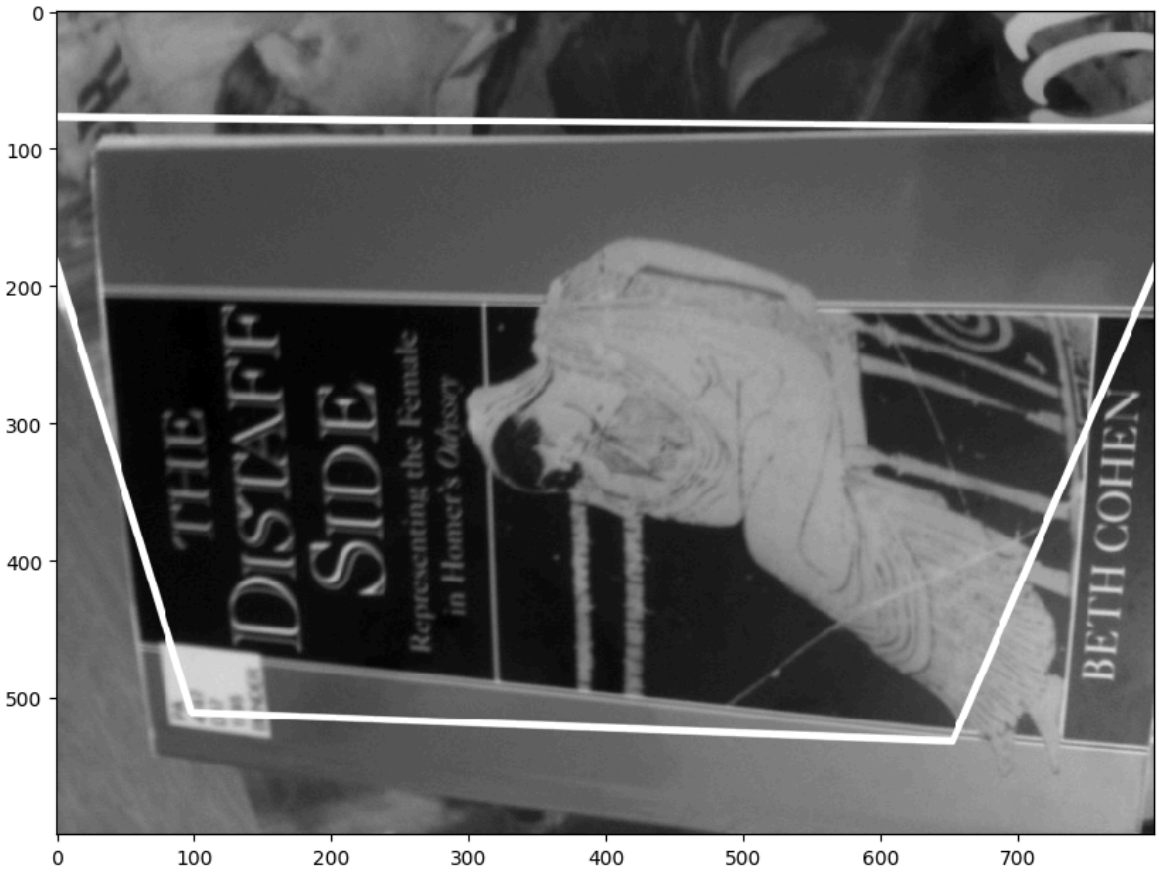
Interpretation and Future Work

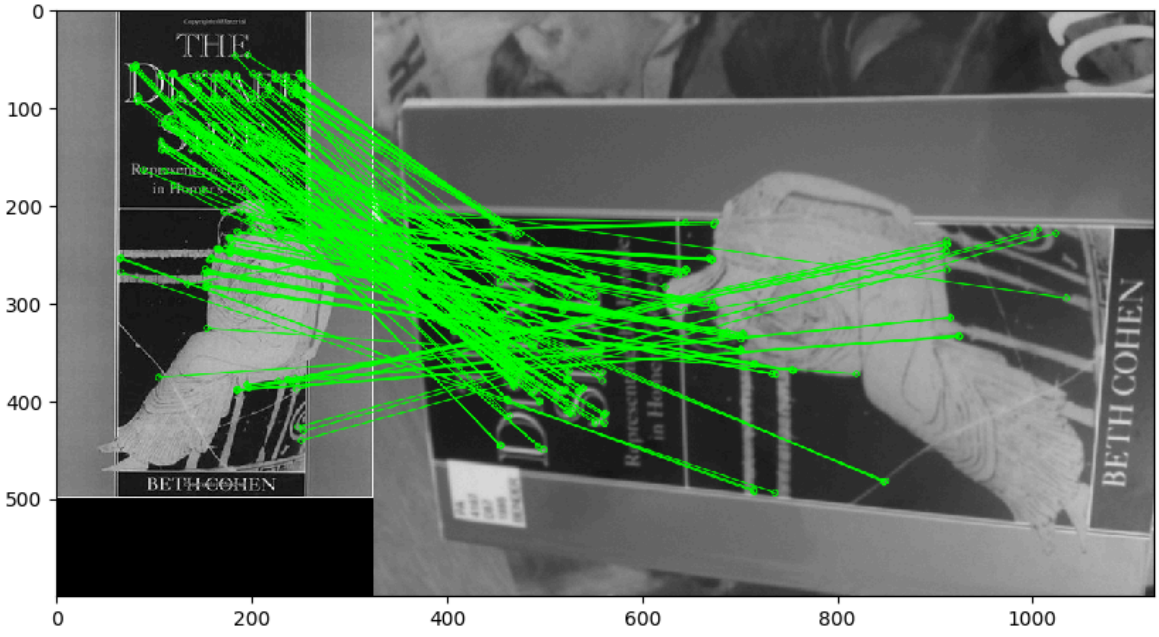
- Match accuracy was assessed visually through manual inspection of drawn matches, focusing on alignment and consistency.
- While match count is useful, it does not guarantee correctness.

5. 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.

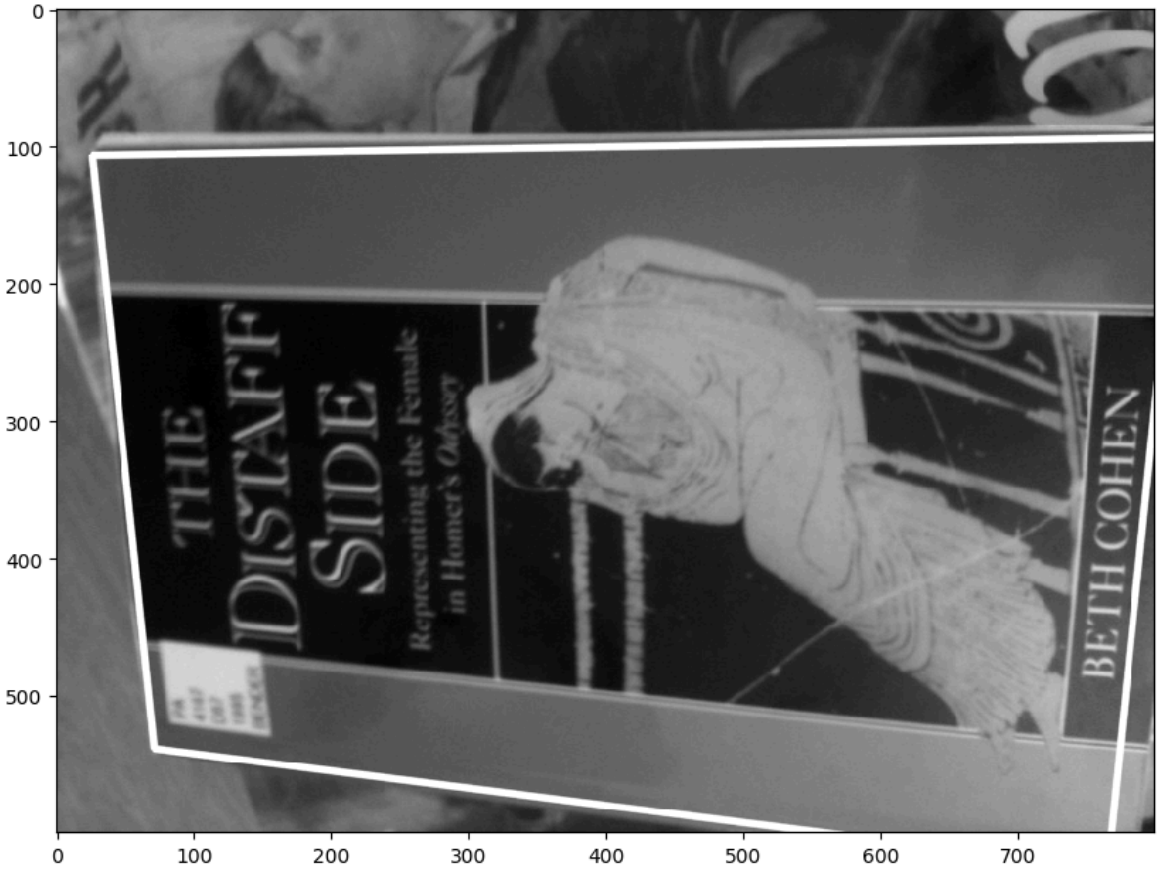
In [8]:
```
img1 = cv2.imread('A2_smvs/book_covers/Reference/001.jpg', 0)
img2 = cv2.imread("A2_smvs/book_covers/Query/001.jpg", 0)

orb = cv2.ORB_create(nfeatures=1000)
kp1, des1 = orb.detectAndCompute(img1, None)
kp2, des2 = orb.detectAndCompute(img2, None)

bf = cv2.BFMatcher(cv2.NORM_HAMMING)
matches = bf.knnMatch(des1, des2, k=2)

good = []
ratio = 0.8
for m,n in matches:
    if m.distance < ratio * n.distance:
        good.append(m) # append good matches to list

# Create src_pts and dst_pts as float arrays to be passed into cv2.,findH
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)

# using cv2 standard method, see (3) above
H, mask = cv2.findHomography(src_pts, dst_pts, method=0)

# draw frame
draw_outline(img1, img2, H)

# draw inliers
draw_inliers(img1, img2, kp1, kp2, good, mask)
```

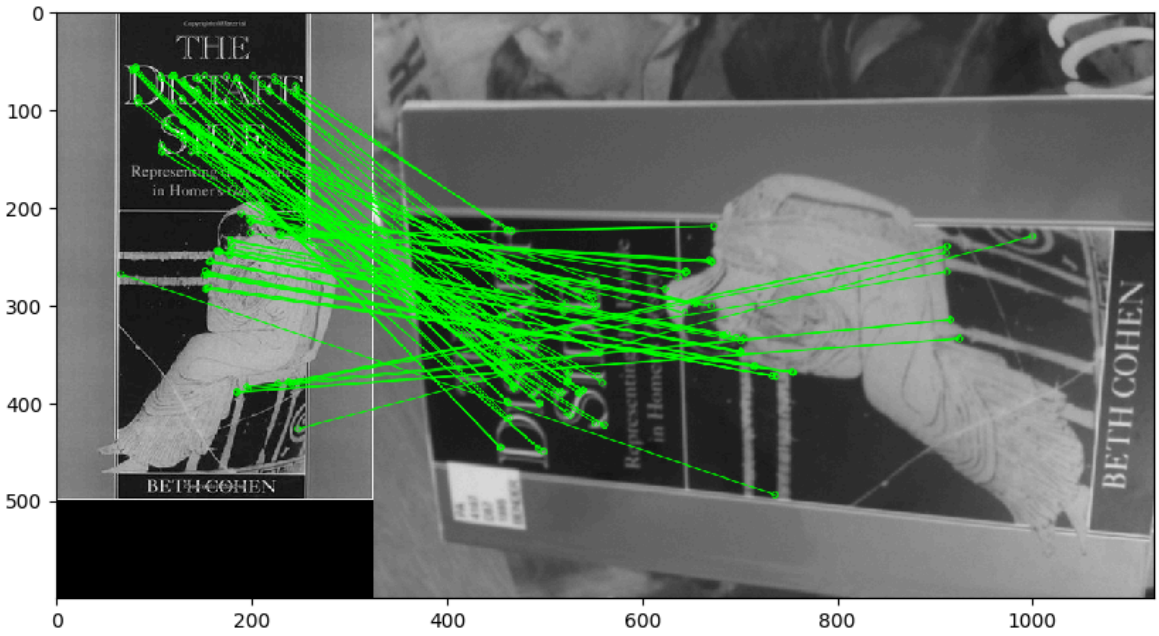




In [9]:
```
img1 = cv2.imread('A2_smvs/book_covers/Reference/001.jpg', 0)
img2 = cv2.imread("A2_smvs/book_covers/Query/001.jpg", 0)

orb = cv2.ORB_create(nfeatures=1000)
kp1, des1 = orb.detectAndCompute(img1, None)
kp2, des2 = orb.detectAndCompute(img2, None)

bf = cv2.BFMatcher(cv2.NORM_HAMMING)
matches = bf.knnMatch(des1, des2, k=2)

good = []
ratio = 0.7
for m,n in matches:
```

```
    if m.distance < ratio * n.distance:
        good.append(m) # append good matches to list

# Create src_pts and dst_pts as float arrays to be passed into cv2.,findH
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)

# using cv2 standard method, see (3) above
H, mask = cv2.findHomography(src_pts, dst_pts, method=0)

# draw frame
draw_outline(img1, img2, H)

# draw inliers
draw_inliers(img1, img2, kp1, kp2, good, mask)
```





Your explanation of results here

Step 5.1 discussion

- Homography Estimation: Used cv2.findHomography() with method=0 (least squares) to estimate the transformation between the reference and query images.
- Ratio Threshold Test
  - At ratio = 0.8, the number of ambiguous matches was higher, leading to a less reliable homography. The outline was distorted and did not tightly fit the book.
  - Lowering the threshold to 0.7 eliminated many poor matches, resulting in a cleaner set of correspondences that successfully enabled homography computation.
- Outline Projection: Using draw_outline(), the reference image was projected onto the query image. With ratio = 0.7, the projected corners accurately matched the true edges of the book.
- Inlier Visualization: Inlier matches consistent with the estimated homography were visualized using cv2.drawMatches() via draw_inliers(). These confirmed spatial consistency between keypoints.
- Visual Interpretation: Even with least square method, the transformation at ratio = 0.7 appeared visually accurate. In contrast, ratio = 0.8 produced a distorted projection in the example.

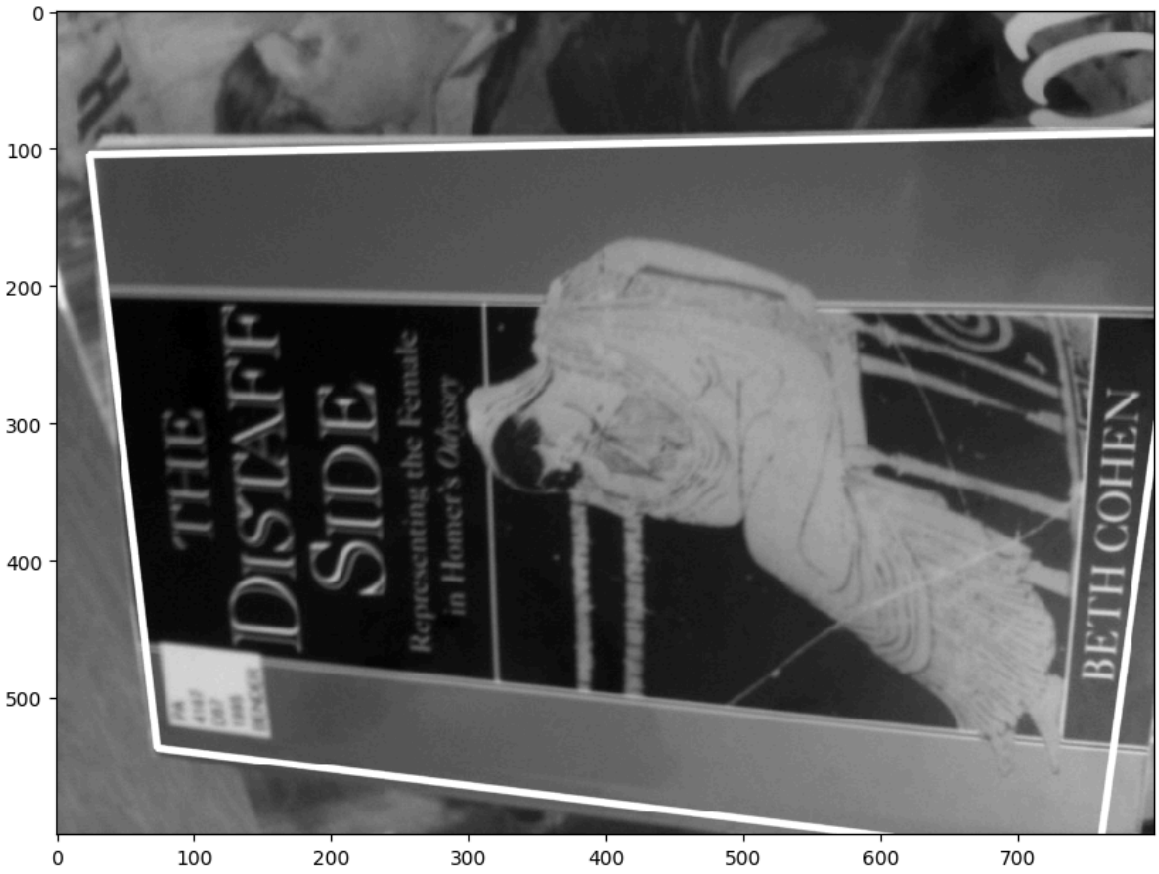
Conclusion

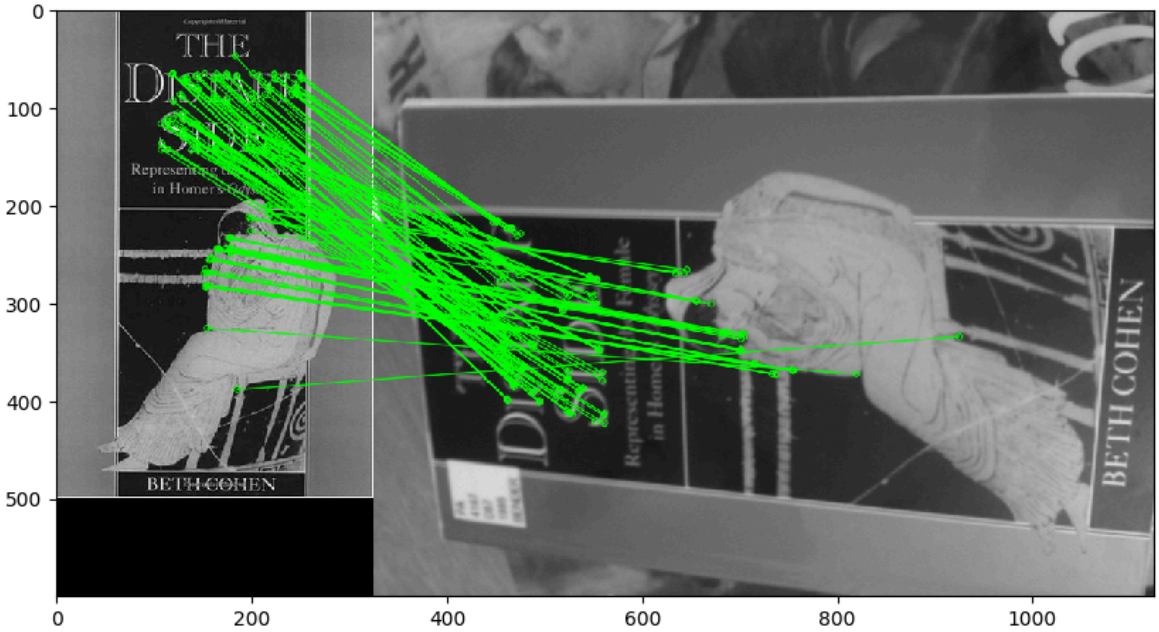
- This highlights the sensitivity of homography to the quality of matched features. Stricter ratio filtering (e.g., 0.7) helped remove false matches and improve the reliability of the transformation.

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

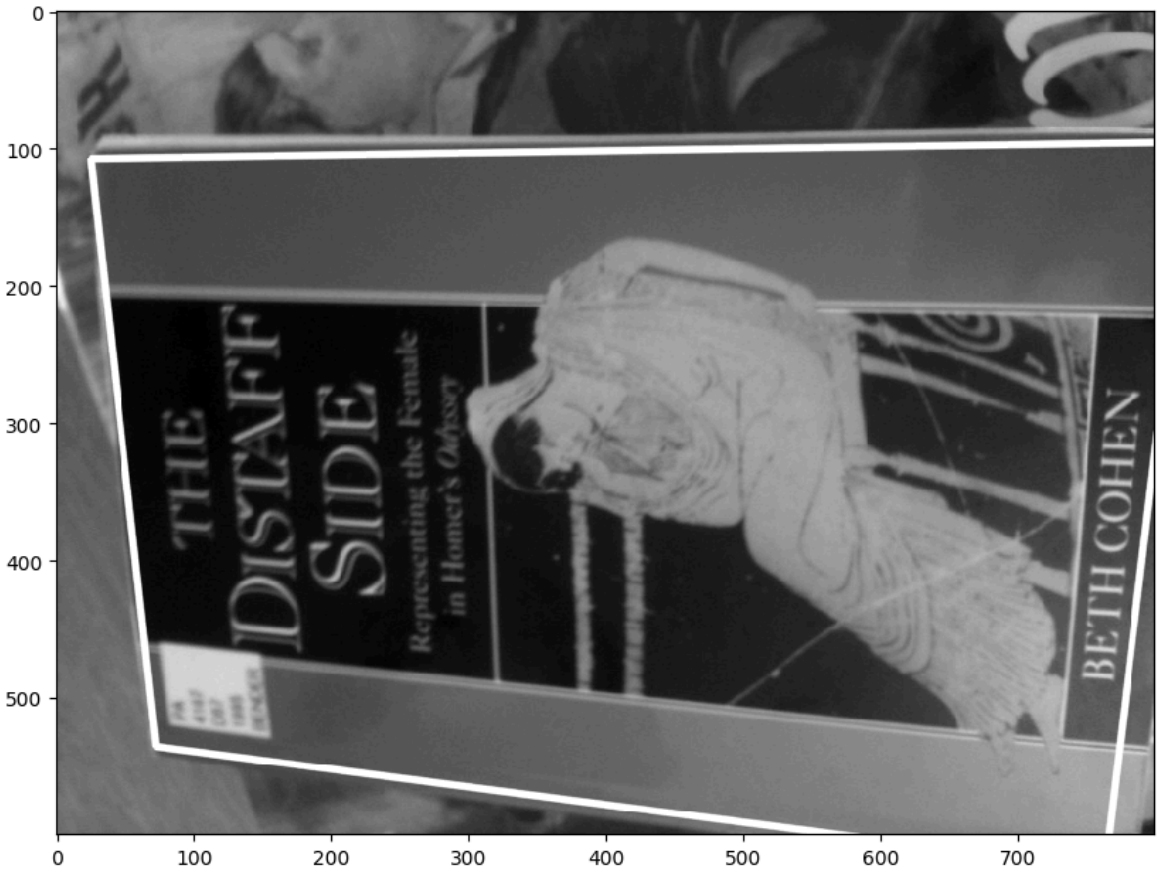
In [10]:
```
# Your code to display book location after RANSAC here
img1 = cv2.imread('A2_smvs/book_covers/Reference/001.jpg', 0)
img2 = cv2.imread("A2_smvs/book_covers/Query/001.jpg", 0)

orb = cv2.ORB_create(nfeatures=1000)
kp1, des1 = orb.detectAndCompute(img1, None)
kp2, des2 = orb.detectAndCompute(img2, None)

bf = cv2.BFMatcher(cv2.NORM_HAMMING)
matches = bf.knnMatch(des1, des2, k=2)

good = []
ratio = 0.8
for m,n in matches:
    if m.distance < ratio * n.distance:
        good.append(m) # append good matches to list

# Create src_pts and dst_pts as float arrays to be passed into cv2.,findH
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)

# using RANSAC
H, mask = cv2.findHomography(src_pts, dst_pts, method=cv2.RANSAC, ransacR

# draw frame
draw_outline(img1, img2, H)

# draw inliers
draw_inliers(img1, img2, kp1, kp2, good, mask)

# inlier number
inliers = mask.ravel().tolist()
print('Number of inliers: ', sum(inliers))

print("----Experiment with different thresholds----")

# experiment with different thresholds
inliers_number_list = []
for threshold in range(1, 21, 2):
    H_i, mask_i = cv2.findHomography(src_pts, dst_pts, method=cv2.RANSAC,
    inliers = mask_i.ravel().tolist()
    inliers_ratio = sum(inliers) / len(good)
    inliers_number_list.append(sum(inliers))
    # plot the outline
    print(f'Number of inliers for threshold {threshold}: ', sum(inliers))
    print("Outline for threshold ", threshold)
    draw_outline(img1, img2, H_i)

# plot the inliers number
plt.plot(range(1, 21, 2), inliers_number_list, 'bo-', label='Inliers Coun
plt.axhline(y=len(good), color='r', linestyle='--', label=f'Number of Goo

plt.xlabel('RANSAC Threshold')
plt.ylabel('Number of Inliers')
plt.title('Number of Inliers vs RANSAC Threshold')
plt.grid(True)
```

```
plt.legend()
plt.show()
```

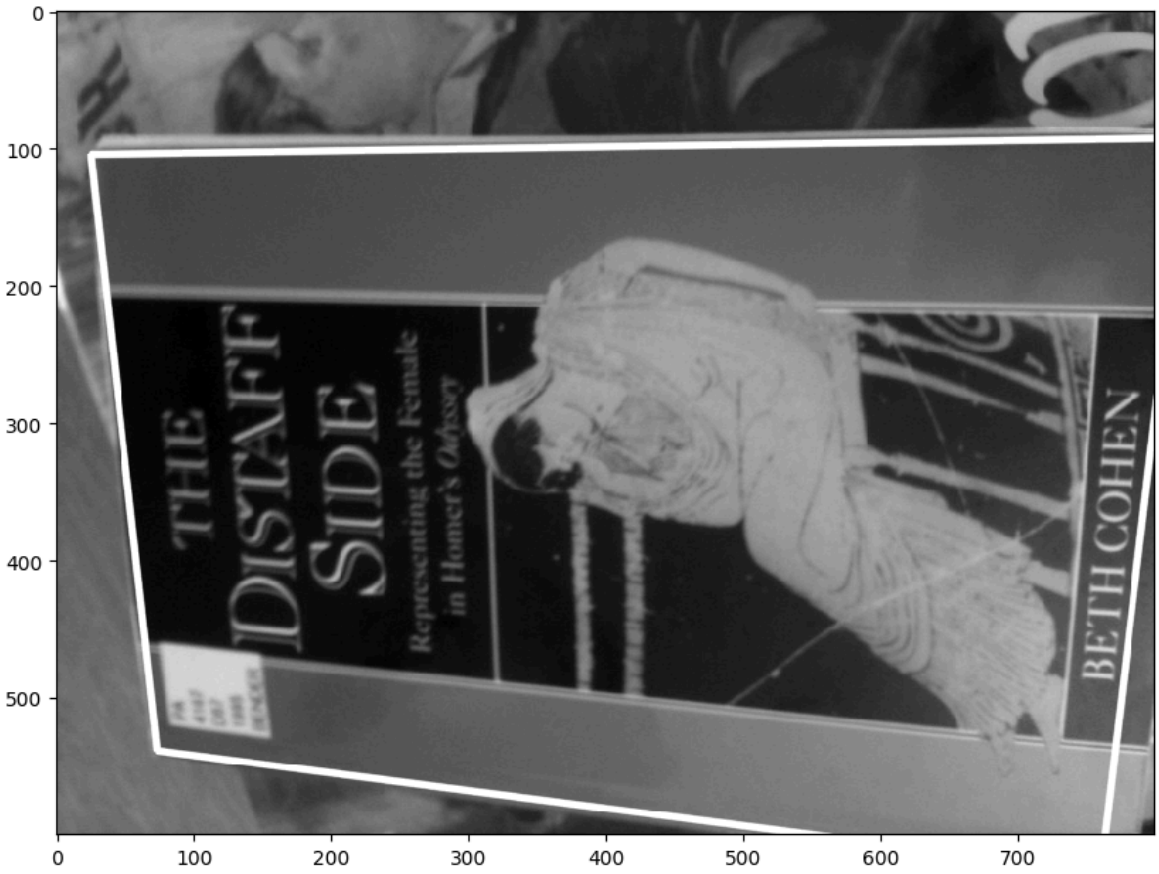




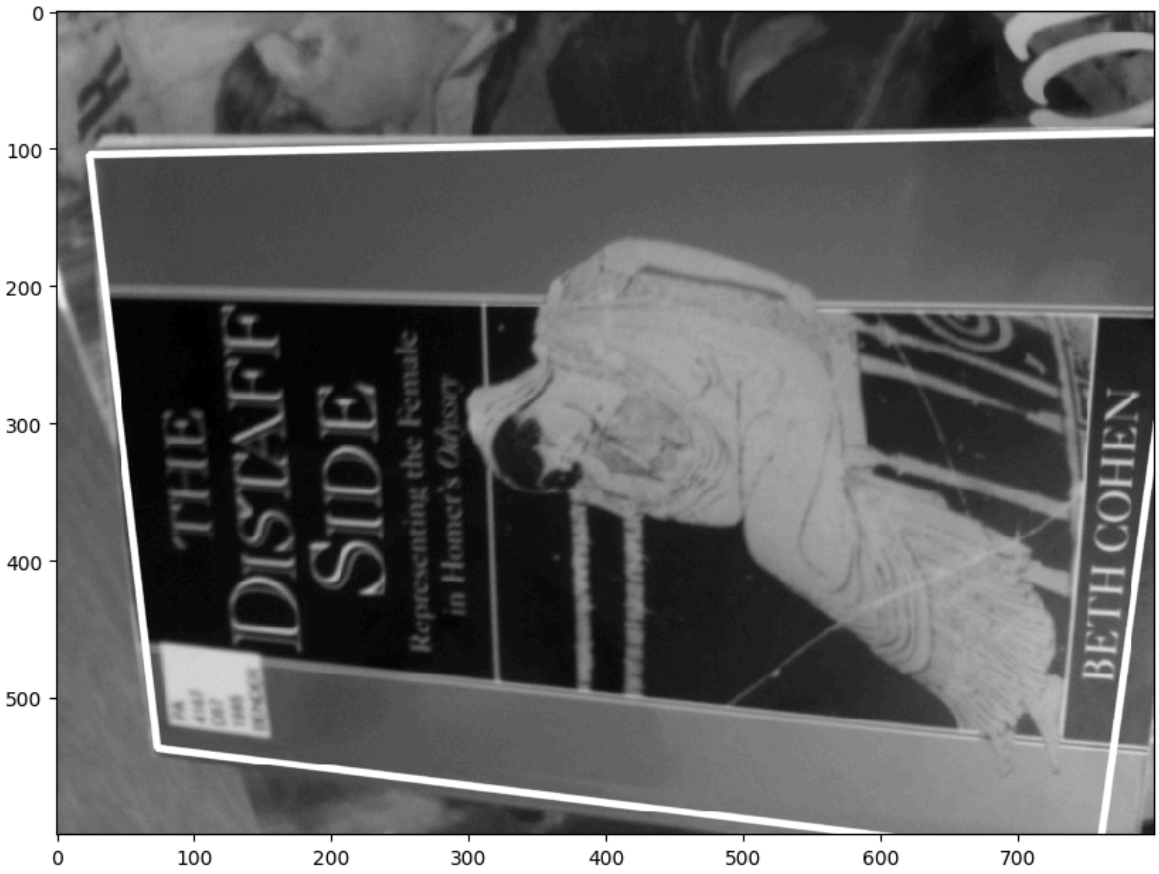

```
Number of inliers:  147
----Experiment with different thresholds----
Number of inliers for threshold 1:  45
Outline for threshold  1
```



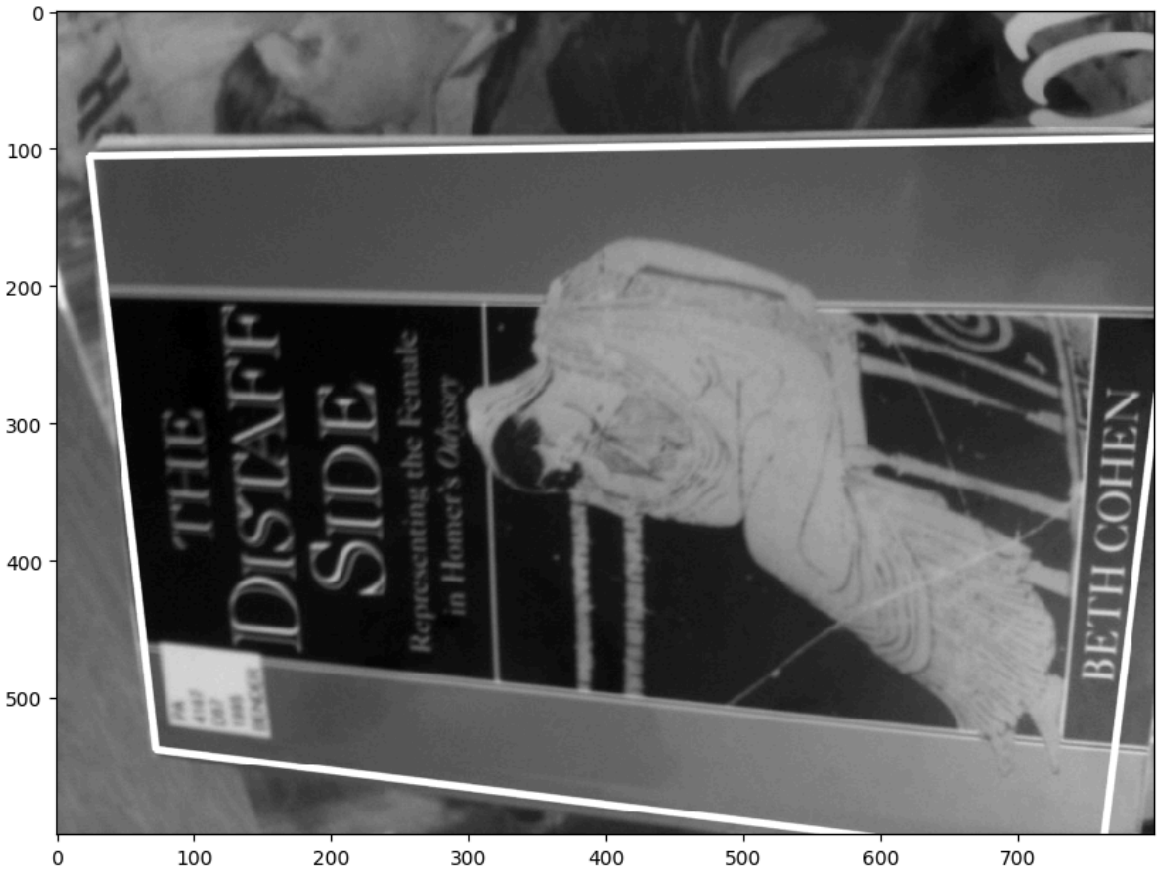
Number of inliers for threshold 3: 141
Outline for threshold 3



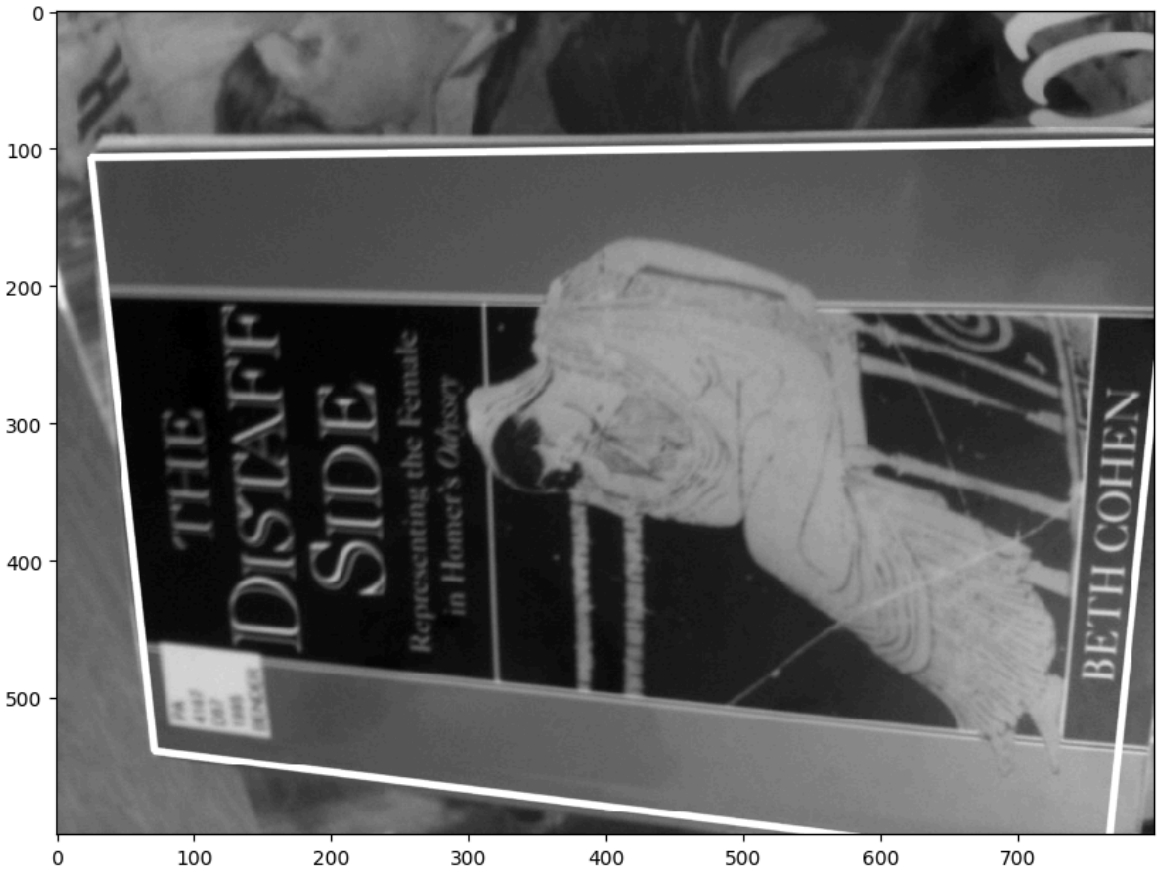
Number of inliers for threshold 5: 147
Outline for threshold 5



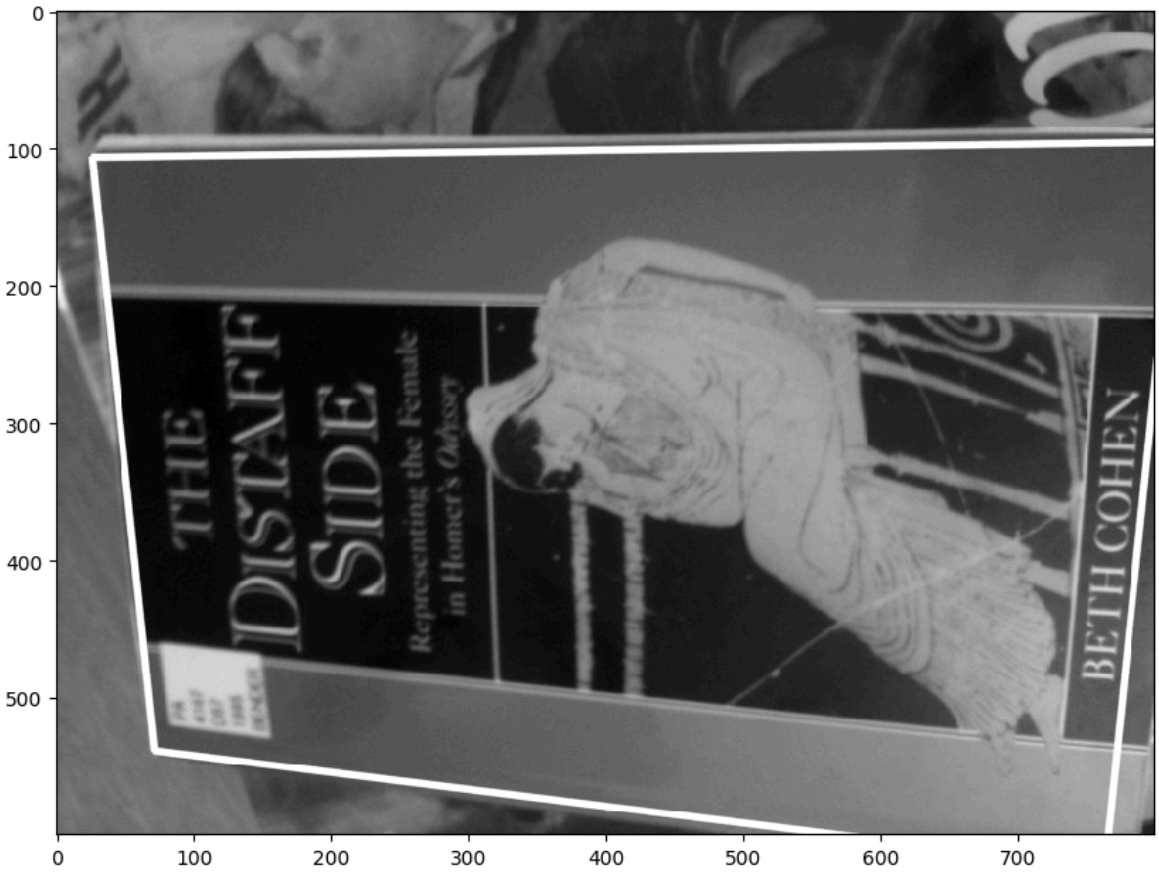
Number of inliers for threshold 7: 174
Outline for threshold 7



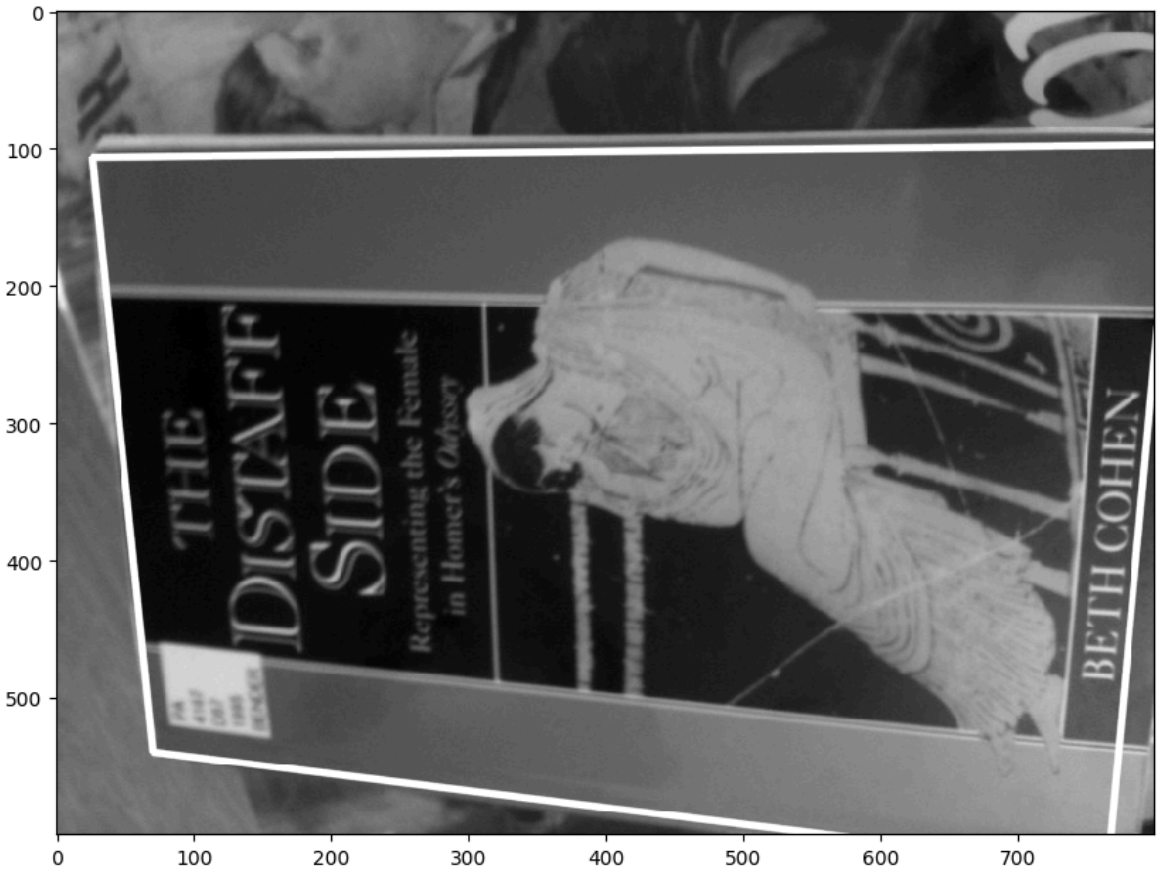
Number of inliers for threshold 9: 185
Outline for threshold 9



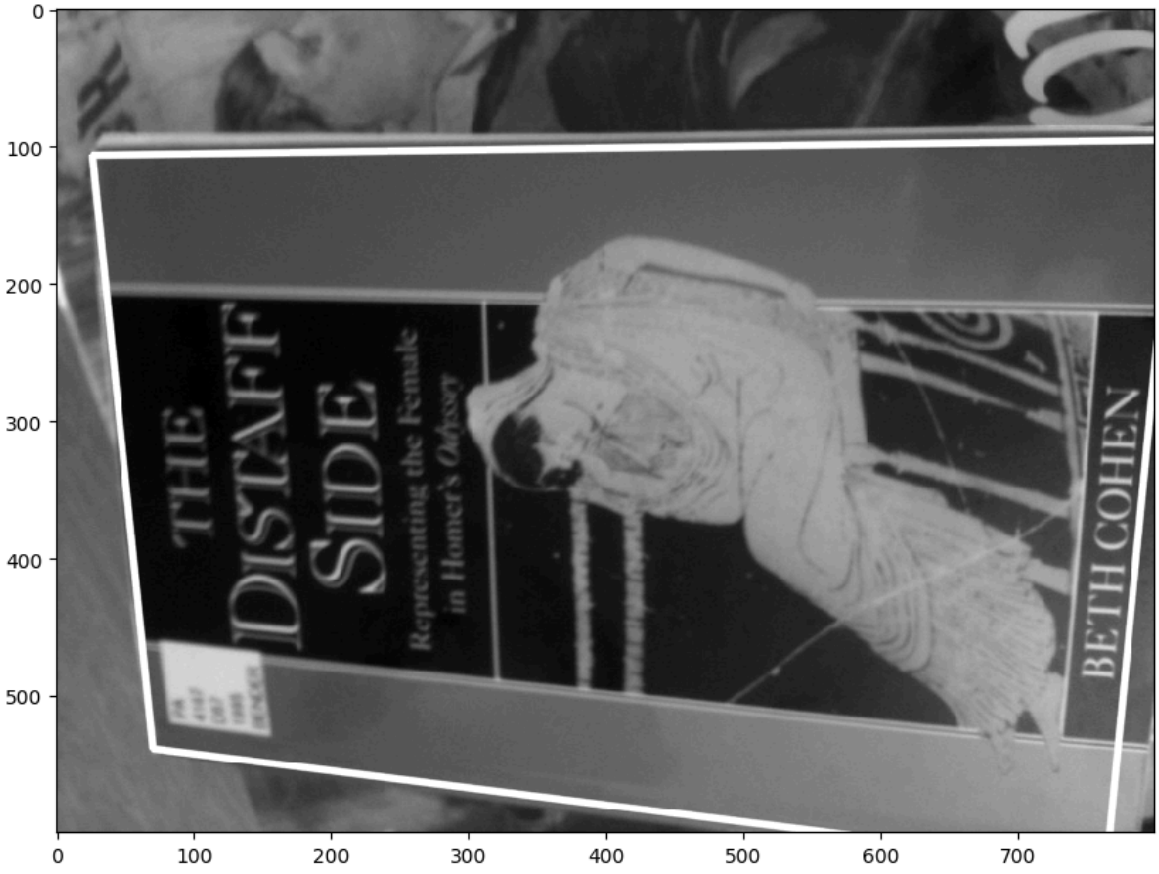

```
Number of inliers for threshold 11:  188
Outline for threshold  11
```



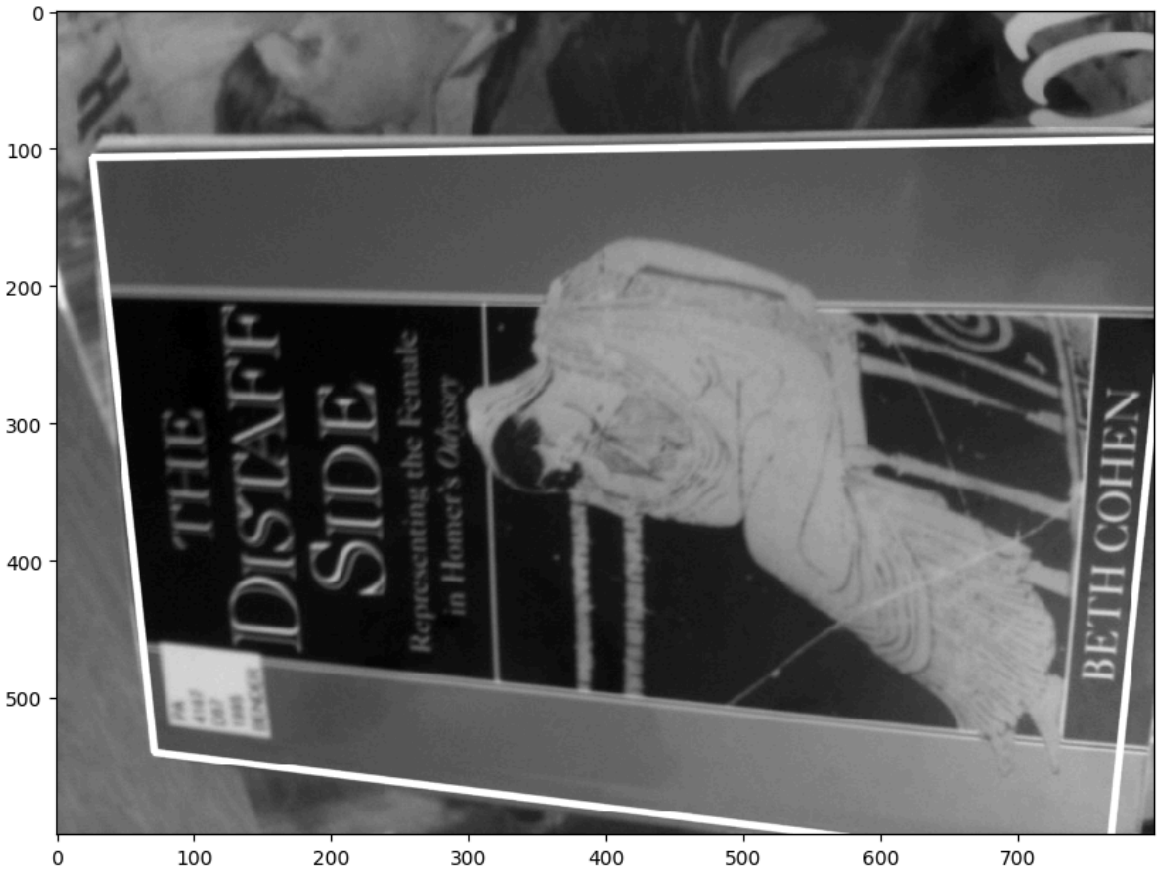

```
Number of inliers for threshold 13:  191
Outline for threshold  13
```



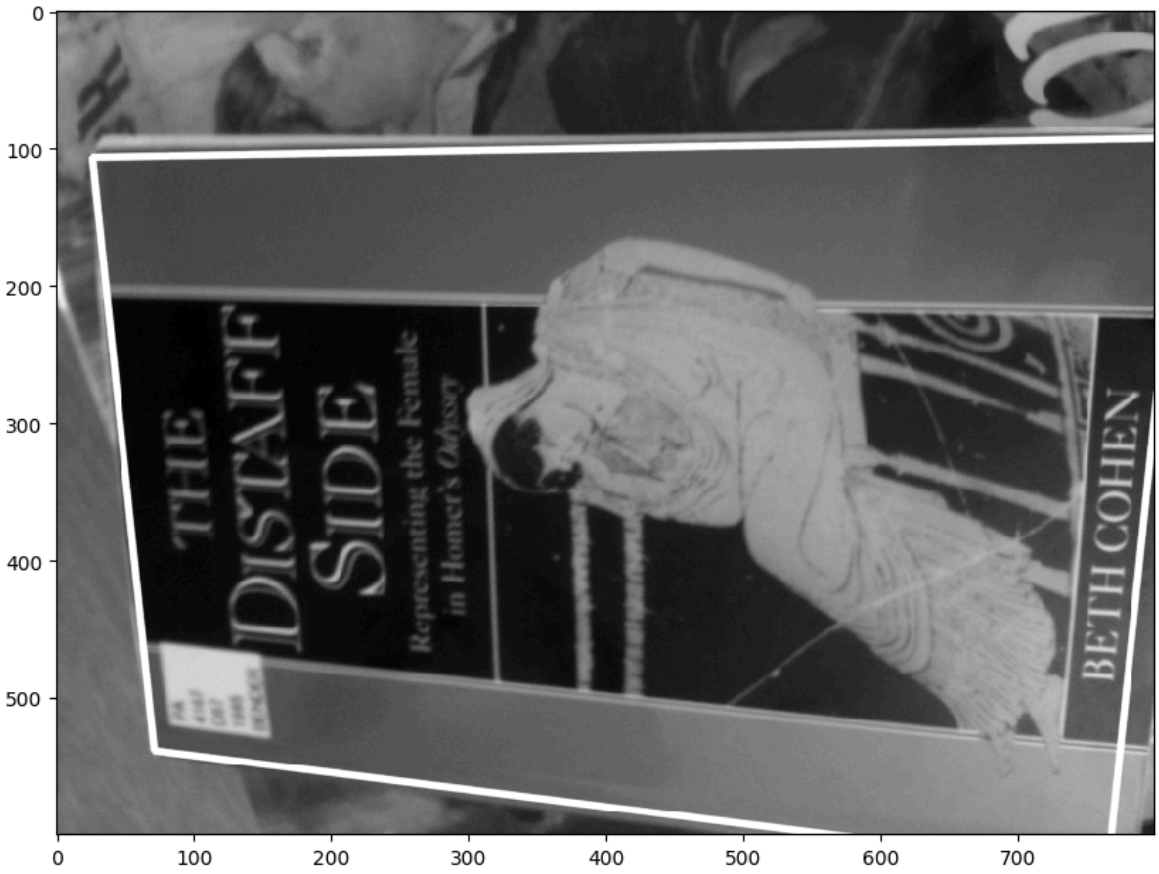
Number of inliers for threshold 15: 194
Outline for threshold 15

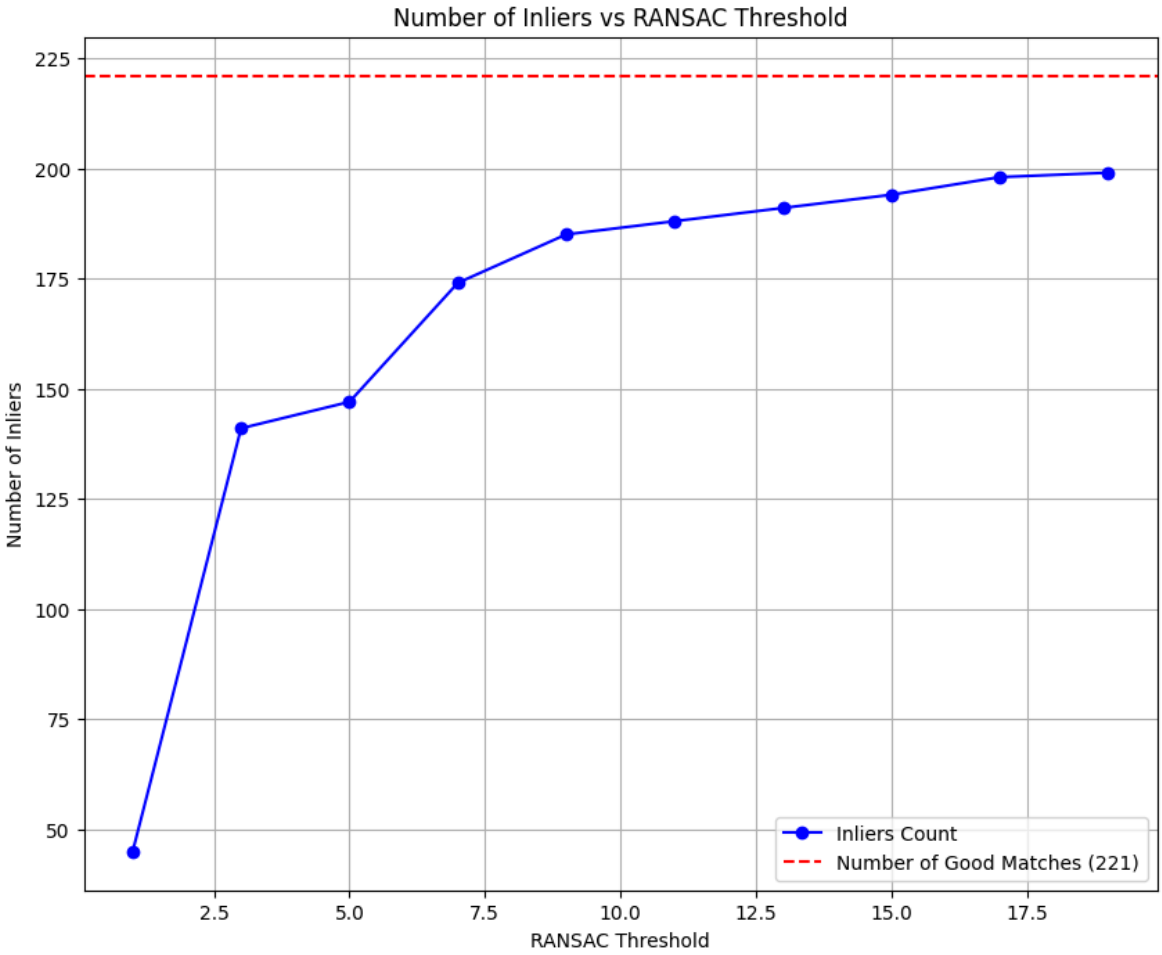


Number of inliers for threshold 17: 198
Outline for threshold 17




```
Number of inliers for threshold 19:  199
Outline for threshold  19
```





Your explanation of what you have tried, and results here

Step 5.2 discussion

Key Observation:

- Despite varying the RANSAC reprojection threshold from 1 to 19, homography estimation was successful at every threshold tested.
- The projected outline of the reference book consistently aligned well with its actual location in the query image.
- This confirms the robustness of RANSAC-based homography estimation in this setup.

Interpretation:

- At lower thresholds (e.g., 1–3), RANSAC is stricter, accepting only very precise matches. This leads to a smaller number of inliers, but these are typically of higher geometric accuracy.
- As the threshold increases, RANSAC becomes more tolerant of error, allowing more matches to be considered inliers — this explains the rising curve.
- However, the rate of increase slows down as the threshold continues to grow. This behavior forms an elbow-shaped curve, where the number of inliers initially increases rapidly, then plateaus.
- This plateau suggests that beyond a certain point, increasing the threshold adds diminishing returns — most valid matches are already included, and further

increases mainly risk admitting poor or noisy correspondences.
- Interestingly, even at higher thresholds (e.g., 15–19), the transformation still remained geometrically stable, likely due to a strong set of dominant correspondences that anchor the homography.

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 [11]:
```
# Function to perform matching and homography estimation for a given pair
def match_and_display(ref_path, query_path, orb_params, ratio_thresh, ran
    print(f"\n--- Matching Pair ---")
    print(f"Reference: {ref_path}")
    print(f"Query:     {query_path}")
    print(f"Parameters: ORB({orb_params}), Ratio={ratio_thresh}, RANSAC T

    img1_gray = cv2.imread(ref_path, cv2.IMREAD_GRAYSCALE)
    img2_gray = cv2.imread(query_path, cv2.IMREAD_GRAYSCALE)
    img1_color = cv2.imread(ref_path)
    img2_color = cv2.imread(query_path)

    if img1_gray is None or img2_gray is None or img1_color is None or im
        print("Error loading images.")
        return False, None

    orb = cv2.ORB_create(**orb_params)
    kp1, des1 = orb.detectAndCompute(img1_gray, None)
    kp2, des2 = orb.detectAndCompute(img2_gray, None)

    print(f"Detected {len(kp1)} (ref) and {len(kp2)} (query) keypoints.")

    if des1 is None or des2 is None or len(kp1) < 2 or len(kp2) < 2:
        print("Not enough keypoints/descriptors found.")
        # Display images anyway
        plt.figure(figsize=(12, 5))
        plt.subplot(121), plt.imshow(cv2.cvtColor(img1_color, cv2.COLOR_
        plt.subplot(122), plt.imshow(cv2.cvtColor(img2_color, cv2.COLOR_
        plt.show()
        return False, None

    bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=False)
    matches = bf.knnMatch(des1, des2, k=2)

    good_matches = []
    valid_matches = [m for m in matches if len(m) == 2]
    for m, n in valid_matches:
        if m.distance < ratio_thresh * n.distance:
```

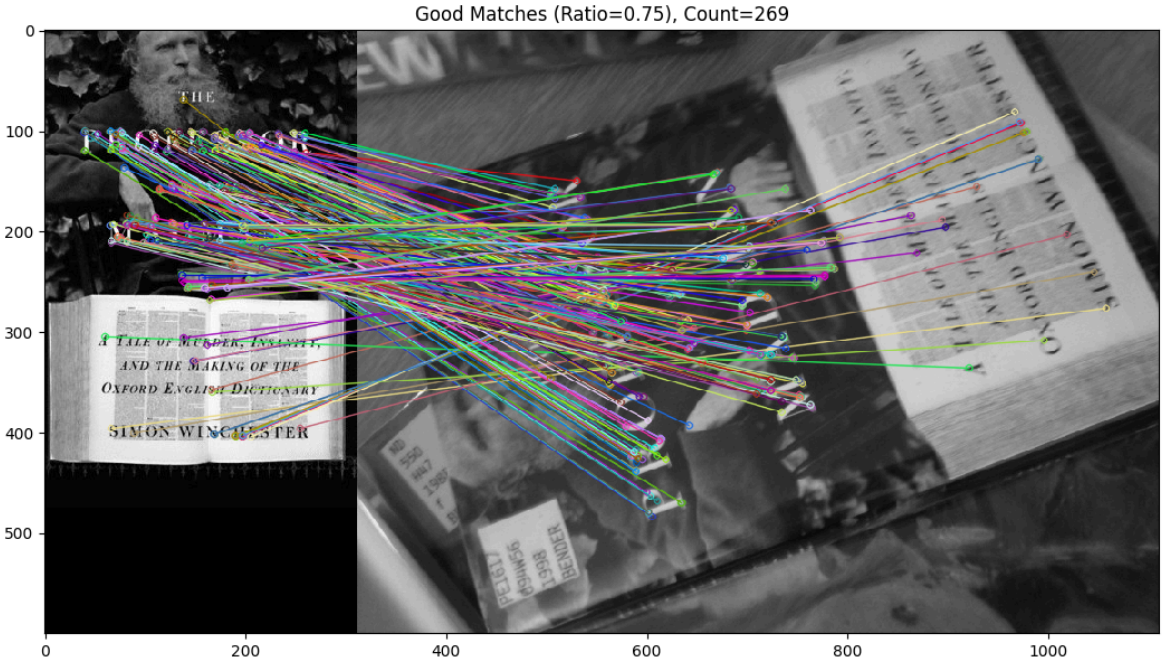
```
            good_matches.append(m)

    print(f"Found {len(good_matches)} good matches after ratio test.")

    # Visualize raw good matches
    img_matches_good = cv2.drawMatches(img1_gray, kp1, img2_gray, kp2, go
    plt.figure(figsize=(15, 7))
    plt.imshow(img_matches_good)
    plt.title(f'Good Matches (Ratio={ratio_thresh}), Count={len(good_matc
    plt.show()


    if len(good_matches) >= 4: # Minimum points for homography
        src_pts = np.float32([ kp1[m.queryIdx].pt for m in good_matches ]
        dst_pts = np.float32([ kp2[m.trainIdx].pt for m in good_matches ]

        H, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, ransac

        if H is not None and mask is not None:
            num_inliers = np.sum(mask)
            print(f"RANSAC Homography successful. Found {num_inliers} inl

            print("Projected outline")
            draw_outline(img1_color, img2_color, H)

            print("RANSAC inlier matches")
            draw_inliers(img1_gray, img2_gray, kp1, kp2, good_matches, ma
            return True, num_inliers
        else:
            print("RANSAC Homography estimation failed.")
            return False, 0
    else:
        print(f"Not enough good matches ({len(good_matches)}) for homogra
        return False, 0
```

In [12]:
```
# Based on previous experiments, these seem reasonable defaults
orb_defaults = {'nfeatures': 1500, 'scaleFactor': 1.2, 'nlevels': 8} # sc
ratio_default = 0.75 # Slightly stricter than 0.8
ransac_thresh_default = 7 # elbow point from previous plot
```

In [13]:
```
# Book cover test 003
print("---------------Example 1: Expected Success (Book Cover 003)-------
ref1 = 'A2_smvs/book_covers/Reference/003.jpg'
query1 = 'A2_smvs/book_covers/Query/003.jpg'
success1, inliers1 = match_and_display(ref1, query1, orb_defaults, ratio_
print(f"Homography estimation: {'Successful' if success1 else 'Failed'},


# Museum painting test 001
print("---------------Example 2: Expected Success (Museum Painting 001)--
ref2 = 'A2_smvs/museum_paintings/Reference/001.jpg'
query2 = 'A2_smvs/museum_paintings/Query/001.jpg'
success2, inliers2 = match_and_display(ref2, query2, orb_defaults, ratio_
print(f"Homography estimation: {'Successful' if success2 else 'Failed'},

# Landmark test 001
print("---------------Example 3: Expected Failure (Landmark 001)---------
ref3 = 'A2_smvs/landmarks/Reference/001.jpg'
```

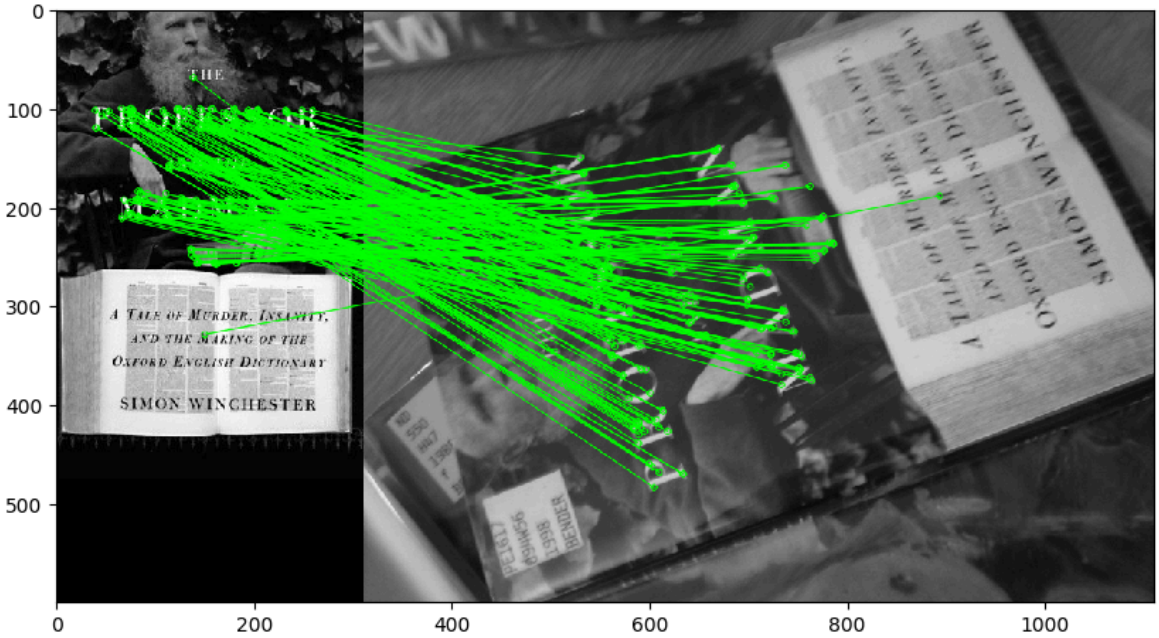
```
query3 = 'A2_smvs/landmarks/Query/001.jpg'
success3, inliers3 = match_and_display(ref3, query3, orb_defaults, ratio_
```

```
---------------Example 1: Expected Success (Book Cover 003)--------------

--- Matching Pair ---
Reference: A2_smvs/book_covers/Reference/003.jpg
Query:     A2_smvs/book_covers/Query/003.jpg
Parameters: ORB({'nfeatures': 1500, 'scaleFactor': 1.2, 'nlevels': 8}), Ra
tio=0.75, RANSAC Thresh=7
Detected 1467 (ref) and 1500 (query) keypoints.
Found 269 good matches after ratio test.
```



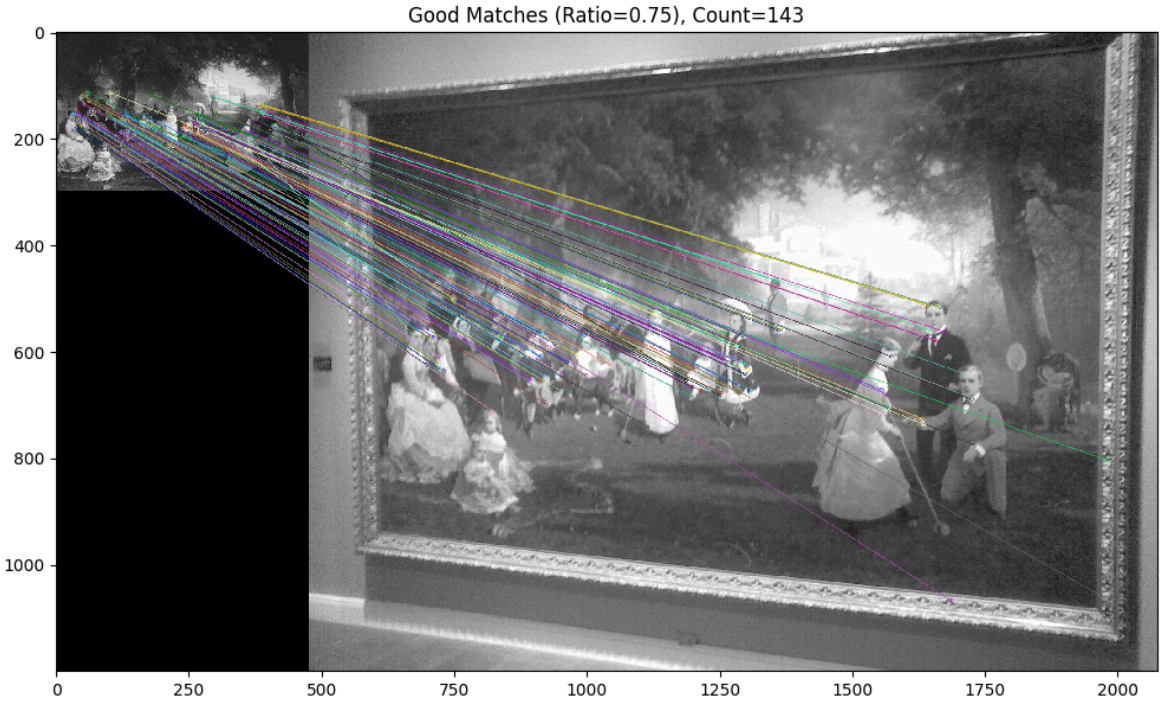

```
RANSAC Homography successful. Found 234 inliers.
Projected outline
```

RANSAC inlier matches



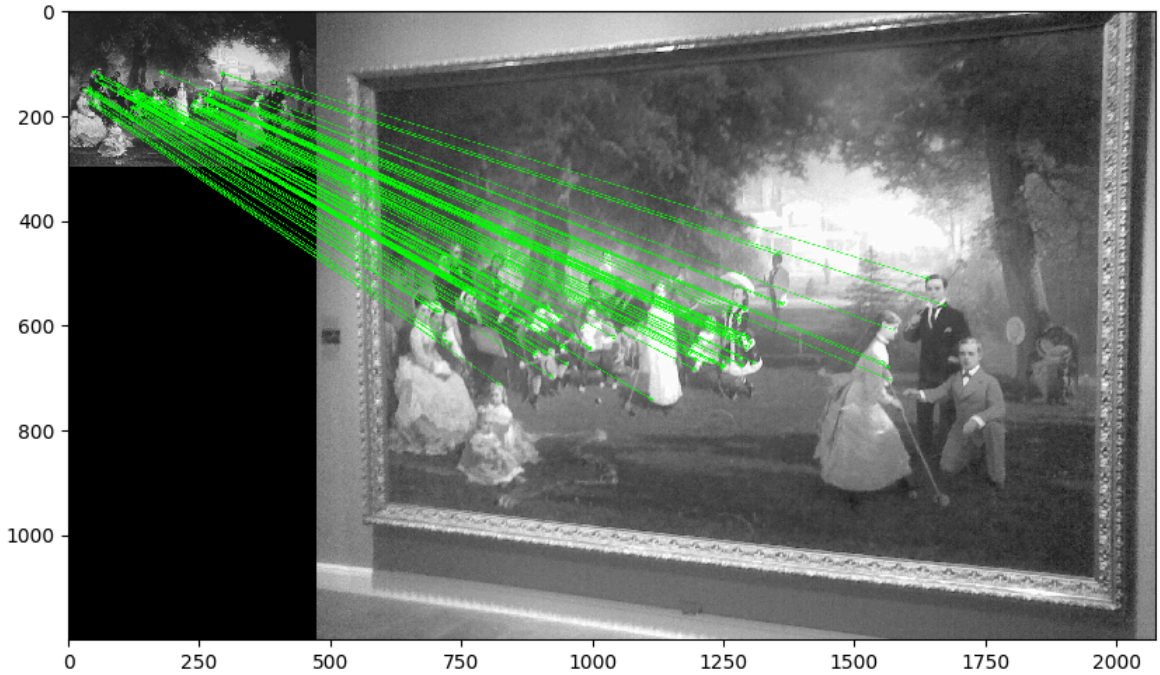
Homography estimation: Successful, Number of inliers: 234
----------Example 2: Expected Success (Museum Painting 001)----------
----

--- Matching Pair ---
Reference: A2_smvs/museum_paintings/Reference/001.jpg
Query: A2_smvs/museum_paintings/Query/001.jpg
Parameters: ORB({'nfeatures': 1500, 'scaleFactor': 1.2, 'nlevels': 8}), Ra
tio=0.75, RANSAC Thresh=7
Detected 1460 (ref) and 1500 (query) keypoints.
Found 143 good matches after ratio test.

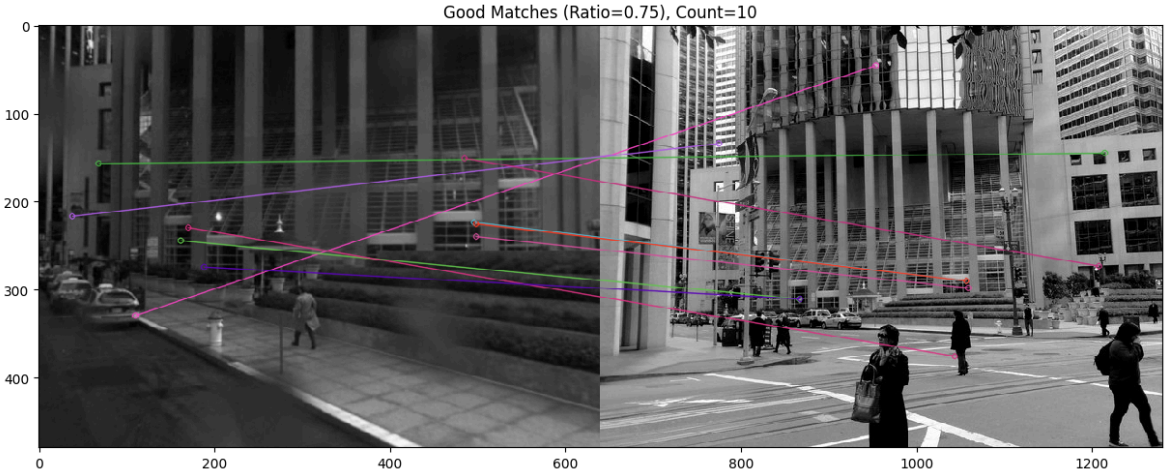


RANSAC Homography successful. Found 119 inliers.
Projected outline

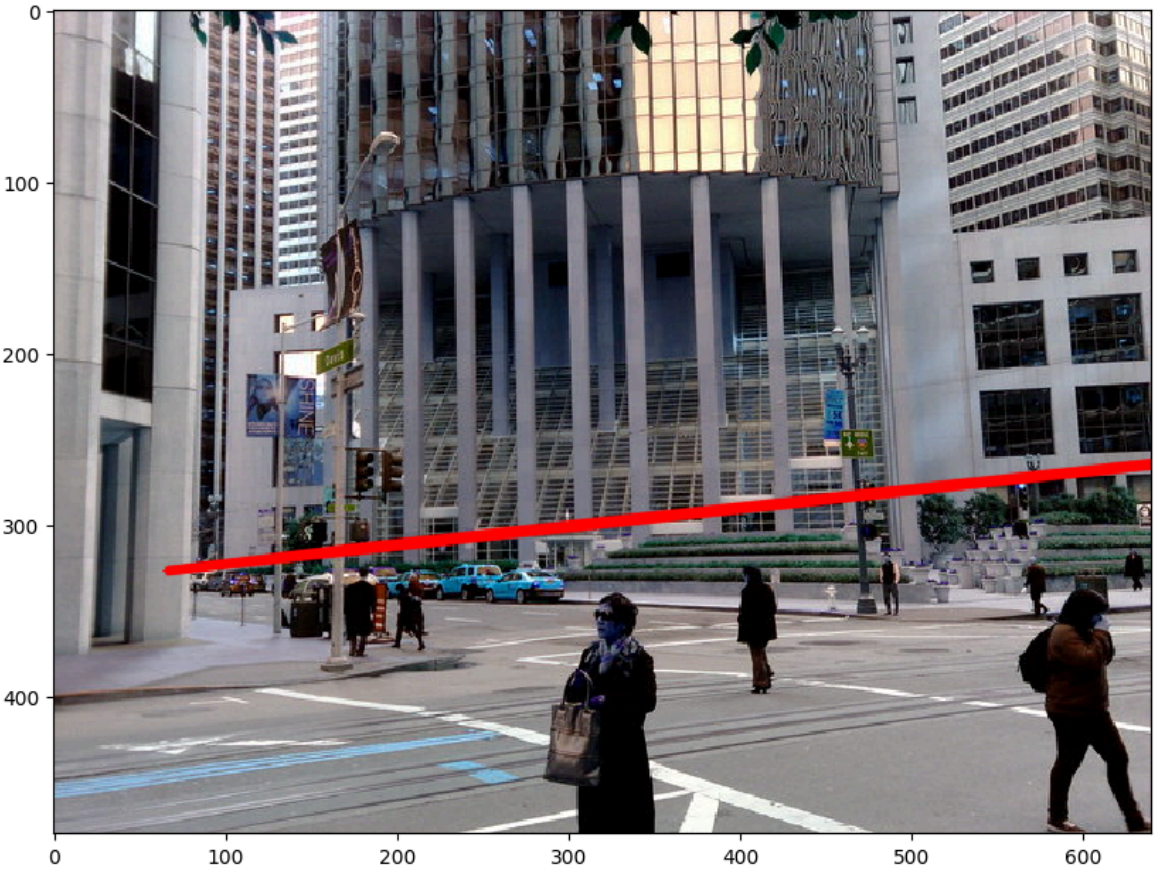
RANSAC inlier matches



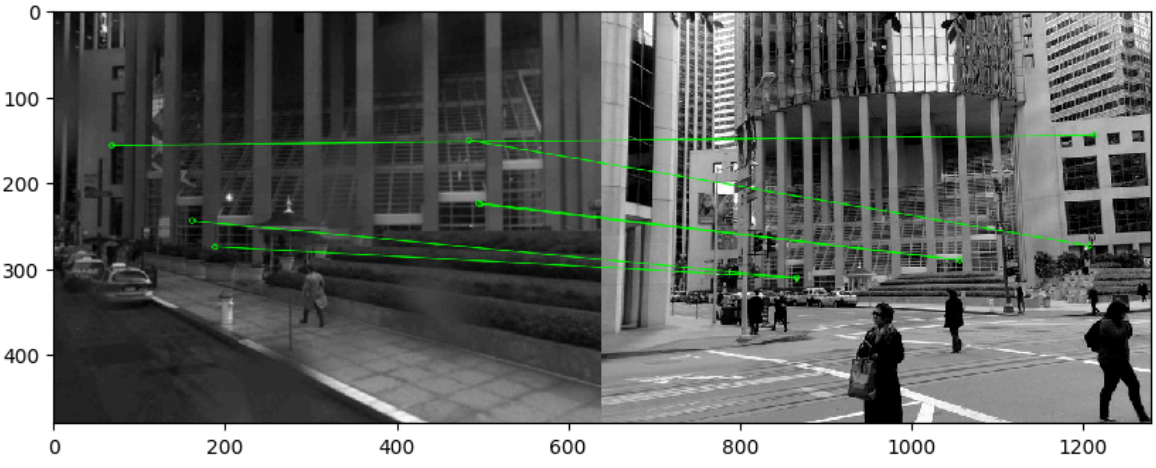

```
Homography estimation: Successful, Number of inliers: 119
---------------Example 3: Expected Failure (Landmark 001)--------------

--- Matching Pair ---
Reference: A2_smvs/landmarks/Reference/001.jpg
Query:     A2_smvs/landmarks/Query/001.jpg
Parameters: ORB({'nfeatures': 1500, 'scaleFactor': 1.2, 'nlevels': 8}), Ra
tio=0.75, RANSAC Thresh=7
Detected 1432 (ref) and 1500 (query) keypoints.
Found 10 good matches after ratio test.
```



RANSAC Homography successful. Found 6 inliers.
Projected outline

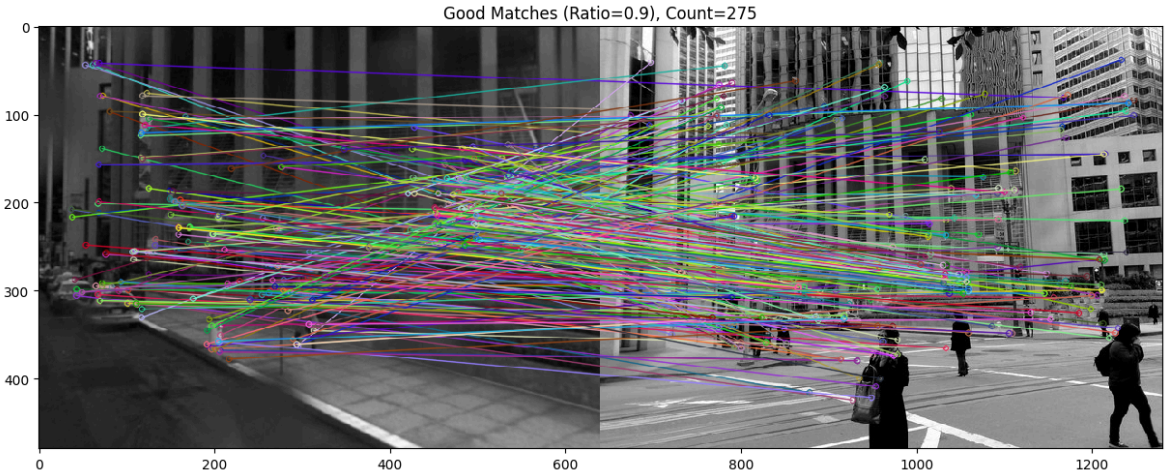


RANSAC inlier matches

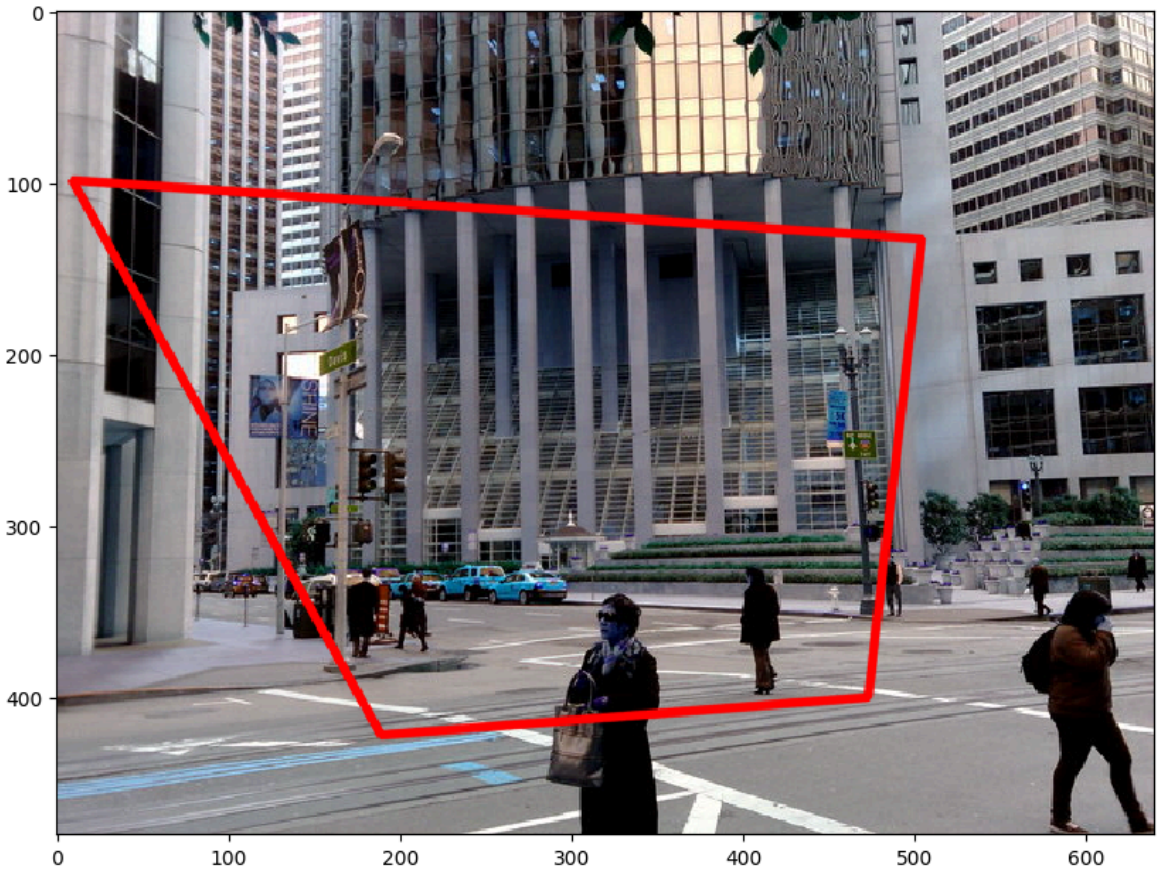


In [14]:
```
# Troubeshooting for landmark test
# Adjusting ORB parameters
orb_params = {'nfeatures': 10000, 'scaleFactor': 1.2, 'nlevels': 8} # Inc
threshold = 0.9 # Ease the ratio test
ransac_thresh = 10 # Ease the RANSAC threshold

print("---------------Troubleshooting Example: Adjusted Parameters-------
ref3 = 'A2_smvs/landmarks/Reference/001.jpg'
query3 = 'A2_smvs/landmarks/Query/001.jpg'
success_troubleshoot, inliers_troubleshoot = match_and_display(ref3, quer
print(f"Adjusted Homography estimation: {'Successful' if success_troubles
```

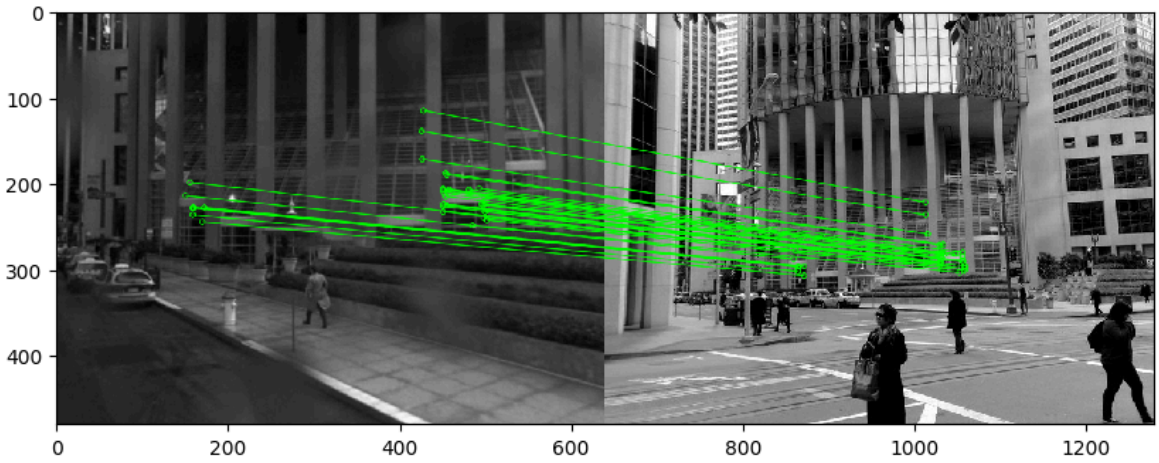
```
---------------Troubleshooting Example: Adjusted Parameters--------------

--- Matching Pair ---
Reference: A2_smvs/landmarks/Reference/001.jpg
Query:     A2_smvs/landmarks/Query/001.jpg
Parameters: ORB({'nfeatures': 10000, 'scaleFactor': 1.2, 'nlevels': 8}), R
atio=0.9, RANSAC Thresh=10
Detected 1698 (ref) and 8881 (query) keypoints.
Found 275 good matches after ratio test.
```




```
RANSAC Homography successful. Found 35 inliers.
Projected outline
```



RANSAC inlier matches



Adjusted Homography estimation: Successful, Number of inliers: 35

Your explanation of results here

Step 6 discussion

Example:

- Example 1 (Book Cover 003):
  - Parameters: ORB(nfeatures=1500), ratio=0.75, RANSAC=7
  - Outcome: Success — found 234 inliers; outline projection and inlier overlay match.
- Example 2 (Museum Painting 001):
  - Parameters: ORB(nfeatures=1500), ratio=0.75, RANSAC=7

    - Outcome: Success — obtained 119 inliers; despite getting less inliers than Example 1, homography remained robust.
- Example 3 (Landmark 001):
    - Parameters: ORB(nfeatures=1500), ratio=0.75, RANSAC=7
    - Outcome: Failure — only 6 inliers, outline warped badly; feature correspondences too sparse/noisy.

Troubleshooting the Failure:

- Increased nfeatures to 10000: generated significantly more keypoints (from 10 to 275), yielding a larger pool of potential correspondences.
- Relaxed ratio to 0.9: admitted more matches, boosting good_matches from 10 to 275.
- Eased RANSAC to 10: inliers rose to 35, enabling a plausible outline, though some background points slipped through.

Key Insights:

- ORB excels on well-textured, planar objects but struggles with low-texture, perspective-rich landmarks.
- Tuning (more features or higher ratio/RANSAC thresholds) can salvage hard cases at the cost of extra outliers.

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. Hint 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?

Step 1-4 combined

Supporting functions for Q2 Part 1-4

In [15]:
```python
# Supporting functions for matching and homography estimation
import numpy as np
import cv2
import os

# load and compute features for reference images
def load_query_reference_images(ref_dir, query_dir, orb):
    """
    Load reference and query images from directories.
    Returns:
        reference_features (dict): Dictionary mapping {ref_path: (keypoin
        query_images (list): List of query image paths.
    """
    reference_features = {}
    for filename in os.listdir(ref_dir):
        if filename.endswith('.jpg') or filename.endswith('.png'):
            ref_path = os.path.join(ref_dir, filename)
            img_ref_gray = cv2.imread(ref_path, cv2.IMREAD_GRAYSCALE)
            if img_ref_gray is None:
                print(f"Warning: Could not load reference image {ref_path
                continue

            kp_ref, des_ref = orb.detectAndCompute(img_ref_gray, None)
            reference_features[ref_path] = (kp_ref, des_ref, img_ref_gray

    query_images = [os.path.join(query_dir, f) for f in os.listdir(query_
    return reference_features, query_images

# match and find best match for a query image against reference features
def find_best_match(query_path, reference_features, orb, bf, min_match_co
    """
    Matches a query image against precomputed reference features.

    Parameters:
        query_path (str): Path to the query image.
        reference_features (dict): Dictionary mapping reference image pat
                                   (keypoints, descriptors, extra_info).
        orb (cv2.ORB): ORB feature extractor.
        bf (cv2.BFMatcher): BFMatcher instance (e.g., created with cv2.NO
        min_match_count (int): Minimum number of good matches needed to a
        ratio_thresh (float): Lowe's ratio test threshold.
```

```
        ransac_thresh (float): RANSAC reprojection threshold.
        top_k (int): Number of top matches to return.

    Returns:
        best_matches (list): List of tuples (ref_path, score) for the top
        scores (dict): Dictionary mapping each reference path to its scor
    """
    # Load and preprocess query image
    img_query_gray = cv2.imread(query_path, cv2.IMREAD_GRAYSCALE)
    if img_query_gray is None:
        print(f"Warning: Could not load query image {query_path}")
        return [], {}

    kp_query, des_query = orb.detectAndCompute(img_query_gray, None)
    if des_query is None:
        return [], {}

    scores = {}  # Dictionary to hold scores: {ref_path: num_inliers}

    # Iterate over each reference image's features
    for ref_path, (kp_ref, des_ref, _) in reference_features.items():
        if des_ref is None:  # Skip if reference image descriptors not co
            scores[ref_path] = 0
            continue

        # Perform KNN matching between the query descriptors and the refe
        matches = bf.knnMatch(des_query, des_ref, k=2)

        # Apply Lowe's ratio test to filter good matches
        good_matches = []
        valid_matches = [m for m in matches if len(m) == 2]
        for m, n in valid_matches:
            if m.distance < ratio_thresh * n.distance:
                good_matches.append(m)

        score = 0  # Default score if not enough matches are found
        # Compute homography only if enough good matches exist
        if len(good_matches) >= min_match_count:
            src_pts = np.float32([kp_query[m.queryIdx].pt for m in good_m
            dst_pts = np.float32([kp_ref[m.trainIdx].pt for m in good_mat

            H, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, ra
            if H is not None and mask is not None:
                score = int(np.sum(mask))  # Use number of inliers as the

        scores[ref_path] = score

    # Sort the references by score in descending order and take the top K
    sorted_scores = sorted(scores.items(), key=lambda item: item[1], reve
    best_matches = sorted_scores[:top_k]

    return best_matches


# calculate accuracy given top k
def calculate_accuracy(result_df: pd.DataFrame, top_k: int = None) -> flo
    correct = 0
    total = len(result_df)

    for _, row in result_df.iterrows():
```

```
        query_path = row['Query Image']
        query_id = query_path.split('/')[-1]  # extract the "number.jpg"
        matches = row['Best Matches']
        max_k = len(matches)
        if top_k is not None and top_k <= max_k:
            k = top_k
        else:
            k = max_k

        # Check if the correct reference image is in the top-k matches
        is_correct = any(query_id in match[0] for match in matches[:k])
        if is_correct:
            correct += 1

    accuracy = correct / total
    return accuracy
```

Experiment with book_covers

In [16]:

```
# Your code to iddntify query objects and measure search accuracy for dat
import os

base_dir = "A2_smvs/book_covers"
query_dir = os.path.join(base_dir, "Query")
reference_dir = os.path.join(base_dir, "Reference")

print(f"Base Directory: {base_dir}")
print(f"Query Directory: {query_dir}")
print(f"Reference Directory: {reference_dir}")

# Use parameters found reasonable in Q1
ORB_PARAMS = {'nfeatures': 1500, 'scaleFactor': 1.2, 'nlevels': 8}
ORB_PARAMS_MORE = {'nfeatures': 3000, 'scaleFactor': 1.2, 'nlevels': 8}
RATIO_THRESH = 0.75
RANSAC_THRESH = 7
MIN_MATCH_COUNT = 10 # minimum number of good matches to attempt homograp
TOP_K = 10 # Top K matches to consider

# Create ORB detector
orb = cv2.ORB_create(**ORB_PARAMS)
orb_more = cv2.ORB_create(**ORB_PARAMS_MORE)
# Create BFMatcher
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=False)

# Load reference images and compute features
reference_features, query_images = load_query_reference_images(reference_
print(f"Loaded {len(reference_features)} reference images and {len(query_

# Find the best K match for each query image
results = []
for query_path in query_images:
    best_ref_path = find_best_match(query_path, reference_features, orb,
    results.append((query_path, best_ref_path))

results_more_feat = []
for query_path in query_images:
    best_ref_path = find_best_match(query_path, reference_features, orb_m
    results_more_feat.append((query_path, best_ref_path))
```

```
# transform to dataframe for easier manipulation
result_df = pd.DataFrame(results, columns=['Query Image', 'Best Matches']
result_df_more_feat = pd.DataFrame(results_more_feat, columns=['Query Ima

# calculate accuracy for deafault param
acc_at_1_book = calculate_accuracy(result_df, top_k=1)
acc_at_3_book = calculate_accuracy(result_df, top_k=3)
acc_at_5_book = calculate_accuracy(result_df, top_k=5)
acc_at_8_book = calculate_accuracy(result_df, top_k=8)
acc_at_10_book = calculate_accuracy(result_df, top_k=10)

# calculate accuracy for more features
acc_at_1_book_more = calculate_accuracy(result_df_more_feat, top_k=1)
acc_at_3_book_more = calculate_accuracy(result_df_more_feat, top_k=3)
acc_at_5_book_more = calculate_accuracy(result_df_more_feat, top_k=5)
acc_at_8_book_more = calculate_accuracy(result_df_more_feat, top_k=8)
acc_at_10_book_more = calculate_accuracy(result_df_more_feat, top_k=10)

print("Results (1500 features):")
print(f"Accuracy at top 1: {acc_at_1_book:.2%}")
print(f"Accuracy at top 3: {acc_at_3_book:.2%}")
print(f"Accuracy at top 5: {acc_at_5_book:.2%}")
print(f"Accuracy at top 8: {acc_at_8_book:.2%}")
print(f"Accuracy at top 10: {acc_at_10_book:.2%}")

print("\nResults (3000 features):")
print(f"Accuracy at top 1: {acc_at_1_book_more:.2%}")
print(f"Accuracy at top 3: {acc_at_3_book_more:.2%}")
print(f"Accuracy at top 5: {acc_at_5_book_more:.2%}")
print(f"Accuracy at top 8: {acc_at_8_book_more:.2%}")
print(f"Accuracy at top 10: {acc_at_10_book_more:.2%}")
```

```
Base Directory: A2_smvs/book_covers
Query Directory: A2_smvs/book_covers/Query
Reference Directory: A2_smvs/book_covers/Reference
Loaded 101 reference images and 101 query images.
Results (1500 features):
Accuracy at top 1: 74.26%
Accuracy at top 3: 85.15%
Accuracy at top 5: 88.12%
Accuracy at top 8: 94.06%
Accuracy at top 10: 94.06%

Results (3000 features):
Accuracy at top 1: 75.25%
Accuracy at top 3: 87.13%
Accuracy at top 5: 91.09%
Accuracy at top 8: 93.07%
Accuracy at top 10: 95.05%
```

Experiment with museum_paintings

In [17]:
```
# Your code to iddntify query objects and measure search accuracy for dat
import os

base_dir = "A2_smvs/museum_paintings"
query_dir = os.path.join(base_dir, "Query")
reference_dir = os.path.join(base_dir, "Reference")
```

```
print(f"Base Directory: {base_dir}")
print(f"Query Directory: {query_dir}")
print(f"Reference Directory: {reference_dir}")

# Use parameters found reasonable in Q1
ORB_PARAMS = {'nfeatures': 1500, 'scaleFactor': 1.2, 'nlevels': 8}
ORB_PARAMS_MORE = {'nfeatures': 10000, 'scaleFactor': 1.2, 'nlevels': 8}
RATIO_THRESH = 0.75
RANSAC_THRESH = 7
MIN_MATCH_COUNT = 10 # minimum number of good matches to attempt homograp
TOP_K = 10 # Top K matches to consider

# Create ORB detector
orb = cv2.ORB_create(**ORB_PARAMS)
orb_more = cv2.ORB_create(**ORB_PARAMS_MORE)

# Create BFMatcher
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=False)

# Load reference images and compute features
reference_features, query_images = load_query_reference_images(reference_
print(f"Loaded {len(reference_features)} reference images and {len(query_

# Find the best K match for each query image
results = []
for query_path in query_images:
    best_ref_path = find_best_match(query_path, reference_features, orb,
    results.append((query_path, best_ref_path))

results_more_feat = []
for query_path in query_images:
    best_ref_path = find_best_match(query_path, reference_features, orb_m
    results_more_feat.append((query_path, best_ref_path))

# transform to dataframe for easier manipulation
result_df = pd.DataFrame(results, columns=['Query Image', 'Best Matches']
result_df_more_feat = pd.DataFrame(results_more_feat, columns=['Query Ima

# calculate accuracy for default params
acc_at_1_painting = calculate_accuracy(result_df, top_k=1)
acc_at_3_painting = calculate_accuracy(result_df, top_k=3)
acc_at_5_painting = calculate_accuracy(result_df, top_k=5)
acc_at_8_painting = calculate_accuracy(result_df, top_k=8)
acc_at_10_painting = calculate_accuracy(result_df, top_k=10)

# calculate accuracy for more features
acc_at_1_painting_more_feat = calculate_accuracy(result_df_more_feat, top
acc_at_3_painting_more_feat = calculate_accuracy(result_df_more_feat, top
acc_at_5_painting_more_feat = calculate_accuracy(result_df_more_feat, top
acc_at_8_painting_more_feat = calculate_accuracy(result_df_more_feat, top
acc_at_10_painting_more_feat = calculate_accuracy(result_df_more_feat, to

print("Results (1500 features):")
print(f"Accuracy at top 1: {acc_at_1_painting:.2%}")
print(f"Accuracy at top 3: {acc_at_3_painting:.2%}")
print(f"Accuracy at top 5: {acc_at_5_painting:.2%}")
print(f"Accuracy at top 8: {acc_at_8_painting:.2%}")
print(f"Accuracy at top 10: {acc_at_10_painting:.2%}")

print("\nResults (3000 features):")
```

```
print(f"Accuracy at top 1: {acc_at_1_painting_more_feat:.2%}")
print(f"Accuracy at top 3: {acc_at_3_painting_more_feat:.2%}")
print(f"Accuracy at top 5: {acc_at_5_painting_more_feat:.2%}")
print(f"Accuracy at top 8: {acc_at_8_painting_more_feat:.2%}")
print(f"Accuracy at top 10: {acc_at_10_painting_more_feat:.2%}")
```

```
Base Directory: A2_smvs/museum_paintings
Query Directory: A2_smvs/museum_paintings/Query
Reference Directory: A2_smvs/museum_paintings/Reference
Loaded 91 reference images and 91 query images.
Results (1500 features):
Accuracy at top 1: 49.45%
Accuracy at top 3: 52.75%
Accuracy at top 5: 53.85%
Accuracy at top 8: 54.95%
Accuracy at top 10: 54.95%

Results (3000 features):
Accuracy at top 1: 51.65%
Accuracy at top 3: 54.95%
Accuracy at top 5: 56.04%
Accuracy at top 8: 58.24%
Accuracy at top 10: 59.34%
```

Experiment with landmarks

In [18]:

```
# Your code to iddntify query objects and measure search accuracy for dat
import os

base_dir = "A2_smvs/landmarks"
query_dir = os.path.join(base_dir, "Query")
reference_dir = os.path.join(base_dir, "Reference")

print(f"Base Directory: {base_dir}")
print(f"Query Directory: {query_dir}")
print(f"Reference Directory: {reference_dir}")

# Use parameters found reasonable in Q1
ORB_PARAMS = {'nfeatures': 1500, 'scaleFactor': 1.2, 'nlevels': 8}
ORB_PARAMS_MORE = {'nfeatures': 10000, 'scaleFactor': 1.2, 'nlevels': 8}
RATIO_THRESH = 0.75
RANSAC_THRESH = 7
MIN_MATCH_COUNT = 10 # minimum number of good matches to attempt homograp
TOP_K = 10 # Top K matches to consider

# Create ORB detector
orb = cv2.ORB_create(**ORB_PARAMS)
orb_more = cv2.ORB_create(**ORB_PARAMS_MORE)
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=False)

# Load reference images and compute features
reference_features, query_images = load_query_reference_images(reference_
print(f"Loaded {len(reference_features)} reference images and {len(query_

# Find the best K match for each query image
results = []
for query_path in query_images:
    best_ref_path = find_best_match(query_path, reference_features, orb,
    results.append((query_path, best_ref_path))
```

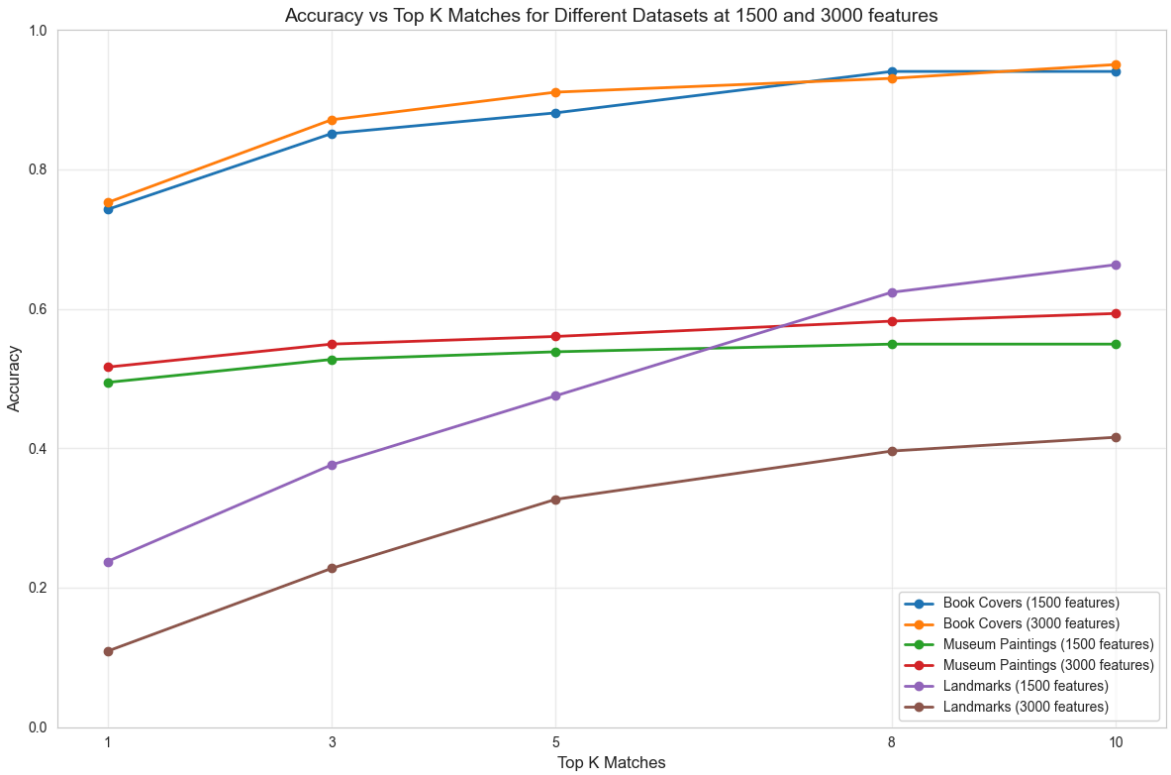
```
results_more_feat = []
for query_path in query_images:
    best_ref_path = find_best_match(query_path, reference_features, orb_m
    results_more_feat.append((query_path, best_ref_path))

# transform to dataframe for easier manipulation
result_df = pd.DataFrame(results, columns=['Query Image', 'Best Matches']
result_df_more_feat = pd.DataFrame(results_more_feat, columns=['Query Ima

# calculate accuracy for default params
acc_at_1_landmarks = calculate_accuracy(result_df, top_k=1)
acc_at_3_landmarks = calculate_accuracy(result_df, top_k=3)
acc_at_5_landmarks = calculate_accuracy(result_df, top_k=5)
acc_at_8_landmarks = calculate_accuracy(result_df, top_k=8)
acc_at_10_landmarks = calculate_accuracy(result_df, top_k=10)

# calculate accuracy for more features
acc_at_1_landmarks_more_feat = calculate_accuracy(result_df_more_feat, to
acc_at_3_landmarks_more_feat = calculate_accuracy(result_df_more_feat, to
acc_at_5_landmarks_more_feat = calculate_accuracy(result_df_more_feat, to
acc_at_8_landmarks_more_feat = calculate_accuracy(result_df_more_feat, to
acc_at_10_landmarks_more_feat = calculate_accuracy(result_df_more_feat, t

print("Results (1500 features):")
print(f"Accuracy at top 1: {acc_at_1_landmarks:.2%}")
print(f"Accuracy at top 3: {acc_at_3_landmarks:.2%}")
print(f"Accuracy at top 5: {acc_at_5_landmarks:.2%}")
print(f"Accuracy at top 8: {acc_at_8_landmarks:.2%}")
print(f"Accuracy at top 10: {acc_at_10_landmarks:.2%}")

print("\nResults (3000 features):")
print(f"Accuracy at top 1: {acc_at_1_landmarks_more_feat:.2%}")
print(f"Accuracy at top 3: {acc_at_3_landmarks_more_feat:.2%}")
print(f"Accuracy at top 5: {acc_at_5_landmarks_more_feat:.2%}")
print(f"Accuracy at top 8: {acc_at_8_landmarks_more_feat:.2%}")
print(f"Accuracy at top 10: {acc_at_10_landmarks_more_feat:.2%}")
```

```
Base Directory: A2_smvs/landmarks
Query Directory: A2_smvs/landmarks/Query
Reference Directory: A2_smvs/landmarks/Reference
Loaded 101 reference images and 101 query images.
Results (1500 features):
Accuracy at top 1: 23.76%
Accuracy at top 3: 37.62%
Accuracy at top 5: 47.52%
Accuracy at top 8: 62.38%
Accuracy at top 10: 66.34%

Results (3000 features):
Accuracy at top 1: 10.89%
Accuracy at top 3: 22.77%
Accuracy at top 5: 32.67%
Accuracy at top 8: 39.60%
Accuracy at top 10: 41.58%
```

In [19]:
```
# visualize accuracy difference @ different K levels
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
```

```python
# Accuracy data for each dataset
accuracy_data = {
    'Book Covers (1500 features)': {
        'top_1': acc_at_1_book,
        'top_3': acc_at_3_book,
        'top_5': acc_at_5_book,
        'top_8': acc_at_8_book,
        'top_10': acc_at_10_book
    },
    'Book Covers (3000 features)': {
        'top_1': acc_at_1_book_more,
        'top_3': acc_at_3_book_more,
        'top_5': acc_at_5_book_more,
        'top_8': acc_at_8_book_more,
        'top_10': acc_at_10_book_more
    },
    'Museum Paintings (1500 features)': {
        'top_1': acc_at_1_painting,
        'top_3': acc_at_3_painting,
        'top_5': acc_at_5_painting,
        'top_8': acc_at_8_painting,
        'top_10': acc_at_10_painting
    },
    'Museum Paintings (3000 features)': {
        'top_1': acc_at_1_painting_more_feat,
        'top_3': acc_at_3_painting_more_feat,
        'top_5': acc_at_5_painting_more_feat,
        'top_8': acc_at_8_painting_more_feat,
        'top_10': acc_at_10_painting_more_feat
    },
    'Landmarks (1500 features)': {
        'top_1': acc_at_1_landmarks,
        'top_3': acc_at_3_landmarks,
        'top_5': acc_at_5_landmarks,
        'top_8': acc_at_8_landmarks,
        'top_10': acc_at_10_landmarks
    },
    'Landmarks (3000 features)': {
        'top_1': acc_at_1_landmarks_more_feat,
        'top_3': acc_at_3_landmarks_more_feat,
        'top_5': acc_at_5_landmarks_more_feat,
        'top_8': acc_at_8_landmarks_more_feat,
        'top_10': acc_at_10_landmarks_more_feat
    }
}

# Set up the figure
plt.figure(figsize=(12, 8))
sns.set_style("whitegrid")

# Create x-axis values
k_values = [1, 3, 5, 8, 10]

# Plot each dataset's accuracy
for dataset, accuracies in accuracy_data.items():
    y_values = [accuracies[f'top_{k}'] for k in k_values]
    plt.plot(k_values, y_values, marker='o', label=dataset, linewidth=2)

# Customize the plot
plt.xlabel('Top K Matches', fontsize=12)
```

```
plt.ylabel('Accuracy', fontsize=12)
plt.title('Accuracy vs Top K Matches for Different Datasets at 1500 and 3
plt.xticks(k_values)
plt.ylim(0, 1)  # Set y-axis from 0 to 1 for percentage
plt.legend(fontsize=10)
plt.grid(True, alpha=0.3)

# Show the plot
plt.tight_layout()
plt.show()
```



Your explanation of what you have done, and your results, here

Step 1-4 Discussion

Step 1–4 retrieval discussion

- Top K trends:
  - For book covers and landmarks, raising K steadily improves retrieval rates (books nearly reach 100% by K = 8; landmarks climb from about 24% at K = 1 to around 66% at K = 10).
  - In contrast, museum paintings show almost no gain as K increases—most correct matches never make the shortlist.
- Landmarks benefit most from longer shortlists:
  - Sparse and noisy matches mean the true landmark often ranks low; allowing K ≥ 5 recovers many of these correct references that would otherwise be missed.
- Effect of increasing ORB features from 1500 to 3000:
  - Unlike our homography experiments in Q1 (where boosting up to 10000 features helped), here more features generally hurt retrieval.

    - Book covers see only small changes, but paintings and landmarks suffer drops at nearly every K level.
    - Landmarks showing the greatest decline.

      -> Extra weak keypoints introduce more confusion than useful correspondences.
- Interpretation:
    - High-contrast objects tolerate extra features and shortlist length, but complex scenes (paintings under varied lighting or 3D landmarks) generate descriptor noise when feature count is raised, degrading match quality.
- Conclusion:
    - Using a moderate K is an easy way to improve retrieval for difficult queries, especially landmarks.
    - Feature tuning must be tailored to the task. Adding more ORB features does not always translate to better retrieval in real-world images.

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.

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.

Step 5 and 6 combined

Supporting functions for Q2 Part 5

In [20]:
```
import pandas as pd

def calculate_accurate_occurance(result_df: pd.DataFrame, inlier_threshol
    """
    Computes retrieval accuracy from a DataFrame.

    DataFrame columns:
      - 'Query Image': query image path.
      - 'Best Matches': list of (ref_path, score) tuples.
      - 'Ground Truth' (optional): correct reference path or "not in data

    For unknown queries, correct if no matches exceed the inlier threshol
```

```
    For known queries, correct if the ground truth match exceeds the thre

    Parameters:
        result_df (pd.DataFrame): Evaluation results.
        inlier_threshold (int): Minimum inliers for a valid match.
        unknown_label (str): Label for queries not in the dataset.

    Returns:
        accuracy (float): Proportion of correctly identified queries.
    """
    correct = 0
    total = len(result_df)

    for _, row in result_df.iterrows():
        # Use the "Ground Truth" column if it exists; otherwise, assume t
        # reference image is based on the query filename
        if 'Ground Truth' in row:
            gt = row['Ground Truth']
        else:
            gt = row['Query Image'].split('/')[-1]  # default assumption

        matches = row['Best Matches']

        # For unknown objects, correct if no matches have enough inliers
        if gt.lower() == unknown_label.lower():
            is_correct = all(match[1] < inlier_threshold for match in mat
        else:
            # For known objects, correct if ground truth match has enough
            gt_matches = [match for match in matches if gt in match[0]]
            is_correct = any(match[1] >= inlier_threshold for match in gt

        if is_correct:
            correct += 1

    accuracy = correct / total if total > 0 else 0.0
    return accuracy
```

Experiment for extra image with books dataset

In [21]:

```
import os
import cv2
import pandas as pd

additional_query_dir = "A2_smvs/extra_images/book"
base_dir             = "A2_smvs/book_covers"
query_dir            = os.path.join(base_dir, "Query")
reference_dir        = os.path.join(base_dir, "Reference")

print(f"Additional Query Directory: {additional_query_dir}")
print(f"Query Directory:            {query_dir}")
print(f"Reference Directory:        {reference_dir}")

# Parameters
ORB_PARAMS      = {'nfeatures': 1500, 'scaleFactor': 1.2, 'nlevels': 8}
RATIO_THRESH    = 0.75
RANSAC_THRESH   = 7
MIN_MATCH_COUNT = 10   # min good matches to attempt homography
TOP_K           = 1    # only need the top-1 match to decide in/out
```

```python
# Create ORB and BFMatcher
orb = cv2.ORB_create(**ORB_PARAMS)
bf  = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=False)

# Load reference features and known queries
reference_features, query_images = load_query_reference_images(reference_

# Load 'unknown' queries (extra images)
_, additional_query_images = load_query_reference_images(reference_dir, a

print(f"Loaded {len(reference_features)} reference images and {len(query_
print(f"Loaded {len(additional_query_images)} extra queries (unknown).")

# Match known queries
known_results = []
for q in query_images:
    best = find_best_match(q, reference_features, orb, bf,
                           MIN_MATCH_COUNT, RATIO_THRESH, RANSAC_THRESH,
    known_results.append((q, best))

# Match unknown queries
unknown_results = []
for q in additional_query_images:
    best = find_best_match(q, reference_features, orb, bf,
                           MIN_MATCH_COUNT, RATIO_THRESH, RANSAC_THRESH,
    unknown_results.append((q, best))

# Build DataFrames
df_known = pd.DataFrame(known_results, columns=['Query Image','Best Match
df_known['Ground Truth'] = df_known['Query Image'].apply(lambda p: os.pat

df_unknown = pd.DataFrame(unknown_results, columns=['Query Image','Best M
df_unknown['Ground Truth'] = "not in dataset"

# Sweep inlier thresholds and compute accuracy
thresholds  = [5, 10, 15, 20, 25, 30]
records     = []
for thr in thresholds:
    acc_known    = calculate_accurate_occurance(df_known,   inlier_thresh
    acc_unknown  = calculate_accurate_occurance(df_unknown, inlier_thresh
    df_combined  = pd.concat([df_known, df_unknown], ignore_index=True)
    acc_combined = calculate_accurate_occurance(df_combined, inlier_thres
    records.append({
        'Category': 'Book Covers',
        'Threshold': thr,
        'Accuracy Known': acc_known,
        'Accuracy Unknown': acc_unknown,
        'Accuracy Combined': acc_combined
    })

acc_df_book = pd.DataFrame(records)
print(acc_df_book)
```

```
Additional Query Directory: A2_smvs/extra_images/book
Query Directory:            A2_smvs/book_covers/Query
Reference Directory:        A2_smvs/book_covers/Reference
Loaded 101 reference images and 101 known queries.
Loaded 10 extra queries (unknown).
      Category  Threshold  Accuracy Known  Accuracy Unknown  Accuracy Comb
ined
0  Book Covers          5        0.742574               0.2           0.69
3694
1  Book Covers         10        0.742574               0.9           0.75
6757
2  Book Covers         15        0.732673               0.9           0.74
7748
3  Book Covers         20        0.702970               1.0           0.72
9730
4  Book Covers         25        0.623762               1.0           0.65
7658
5  Book Covers         30        0.594059               1.0           0.63
0631
```

Experiment for extra image with museum paintings dataset

In [22]:

```python
import os
import cv2
import pandas as pd

additional_query_dir = "A2_smvs/extra_images/painting"
base_dir             = "A2_smvs/museum_paintings"
query_dir            = os.path.join(base_dir, "Query")
reference_dir        = os.path.join(base_dir, "Reference")

print(f"Additional Query Directory: {additional_query_dir}")
print(f"Query Directory:            {query_dir}")
print(f"Reference Directory:        {reference_dir}")

# Parameters
ORB_PARAMS      = {'nfeatures': 1500, 'scaleFactor': 1.2, 'nlevels': 8}
RATIO_THRESH    = 0.75
RANSAC_THRESH   = 7
MIN_MATCH_COUNT = 10
TOP_K           = 1

# Create ORB and BFMatcher
orb = cv2.ORB_create(**ORB_PARAMS)
bf  = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=False)

# Load reference features and known queries
reference_features, query_images = load_query_reference_images(reference_

# Load 'unknown' queries (extra images)
_, additional_query_images = load_query_reference_images(reference_dir, a

print(f"Loaded {len(reference_features)} reference images and {len(query_
print(f"Loaded {len(additional_query_images)} extra queries (unknown).")

# Match known queries
known_results = []
for q in query_images:
    best = find_best_match(q, reference_features, orb, bf,
```

```
                            MIN_MATCH_COUNT, RATIO_THRESH, RANSAC_THRESH,
    known_results.append((q, best))

# Match unknown queries
unknown_results = []
for q in additional_query_images:
    best = find_best_match(q, reference_features, orb, bf,
                            MIN_MATCH_COUNT, RATIO_THRESH, RANSAC_THRESH,
    unknown_results.append((q, best))

# Build DataFrames
df_known = pd.DataFrame(known_results, columns=['Query Image','Best Match
df_known['Ground Truth'] = df_known['Query Image'].apply(lambda p: os.pat

df_unknown = pd.DataFrame(unknown_results, columns=['Query Image','Best M
df_unknown['Ground Truth'] = "not in dataset"

# Sweep inlier thresholds and compute accuracy
thresholds  = [5, 10, 15, 20, 25, 30]
records     = []
for thr in thresholds:
    acc_known    = calculate_accurate_occurance(df_known,   inlier_thresh
    acc_unknown  = calculate_accurate_occurance(df_unknown, inlier_thresh
    df_combined  = pd.concat([df_known, df_unknown], ignore_index=True)
    acc_combined = calculate_accurate_occurance(df_combined, inlier_thres
    records.append({
        'Category': 'Museum Paintings',
        'Threshold': thr,
        'Accuracy Known': acc_known,
        'Accuracy Unknown': acc_unknown,
        'Accuracy Combined': acc_combined
    })

acc_df_painting = pd.DataFrame(records)
print(acc_df_painting)
```

```
Additional Query Directory: A2_smvs/extra_images/painting
Query Directory:            A2_smvs/museum_paintings/Query
Reference Directory:        A2_smvs/museum_paintings/Reference
Loaded 91 reference images and 91 known queries.
Loaded 10 extra queries (unknown).
           Category  Threshold  Accuracy Known  Accuracy Unknown  \
0  Museum Paintings          5        0.494505               0.0
1  Museum Paintings         10        0.483516               0.6
2  Museum Paintings         15        0.439560               0.9
3  Museum Paintings         20        0.417582               1.0
4  Museum Paintings         25        0.406593               1.0
5  Museum Paintings         30        0.384615               1.0

   Accuracy Combined
0           0.445545
1           0.495050
2           0.485149
3           0.475248
4           0.465347
5           0.445545
```

Experiment for extra image with landmarks dataset

In [23]:

```
import os
import cv2
import pandas as pd

additional_query_dir = "A2_smvs/extra_images/landmarks"
base_dir             = "A2_smvs/landmarks"
query_dir            = os.path.join(base_dir, "Query")
reference_dir        = os.path.join(base_dir, "Reference")

print(f"Additional Query Directory: {additional_query_dir}")
print(f"Query Directory:            {query_dir}")
print(f"Reference Directory:        {reference_dir}")

# Parameters
ORB_PARAMS      = {'nfeatures': 1500, 'scaleFactor': 1.2, 'nlevels': 8}
RATIO_THRESH    = 0.75
RANSAC_THRESH   = 7
MIN_MATCH_COUNT = 10
TOP_K           = 1

# Create ORB and BFMatcher
orb = cv2.ORB_create(**ORB_PARAMS)
bf  = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=False)

# Load reference features and known queries
reference_features, query_images = load_query_reference_images(reference_

# Load 'unknown' queries (extra images)
_, additional_query_images = load_query_reference_images(reference_dir, a

print(f"Loaded {len(reference_features)} reference images and {len(query_
print(f"Loaded {len(additional_query_images)} extra queries (unknown).")

# Match known queries
known_results = []
for q in query_images:
    best = find_best_match(q, reference_features, orb, bf,
                           MIN_MATCH_COUNT, RATIO_THRESH, RANSAC_THRESH,
    known_results.append((q, best))

# Match unknown queries
unknown_results = []
for q in additional_query_images:
    best = find_best_match(q, reference_features, orb, bf,
                           MIN_MATCH_COUNT, RATIO_THRESH, RANSAC_THRESH,
    unknown_results.append((q, best))

# Build DataFrames
df_known = pd.DataFrame(known_results, columns=['Query Image','Best Match
df_known['Ground Truth'] = df_known['Query Image'].apply(lambda p: os.pat

df_unknown = pd.DataFrame(unknown_results, columns=['Query Image','Best M
df_unknown['Ground Truth'] = "not in dataset"

# Sweep inlier thresholds and compute accuracy
thresholds  = [5, 10, 15, 20, 25, 30]
records     = []
for thr in thresholds:
    acc_known    = calculate_accurate_occurance(df_known,   inlier_thresh
```

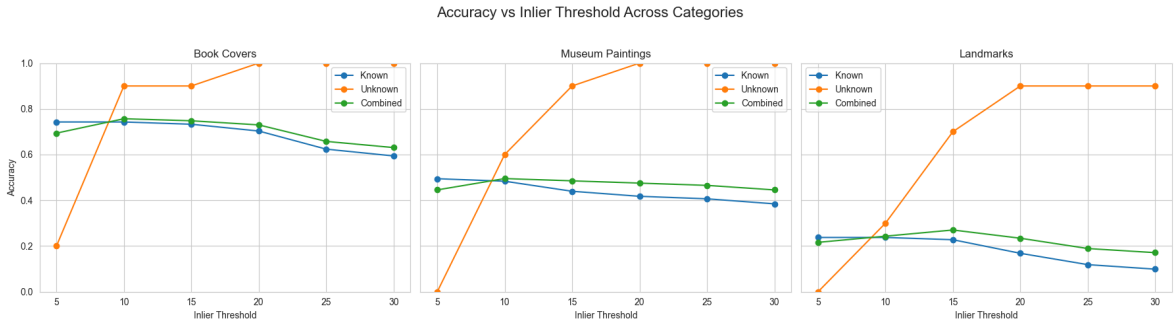
```
    acc_unknown  = calculate_accurate_occurance(df_unknown, inlier_thresh
    df_combined  = pd.concat([df_known, df_unknown], ignore_index=True)
    acc_combined = calculate_accurate_occurance(df_combined, inlier_thres
    records.append({
        'Category': 'Landmarks',
        'Threshold': thr,
        'Accuracy Known': acc_known,
        'Accuracy Unknown': acc_unknown,
        'Accuracy Combined': acc_combined
    })

acc_df_landmark = pd.DataFrame(records)
print(acc_df_landmark)
```

```
Additional Query Directory: A2_smvs/extra_images/landmarks
Query Directory:            A2_smvs/landmarks/Query
Reference Directory:        A2_smvs/landmarks/Reference
Loaded 101 reference images and 101 known queries.
Loaded 10 extra queries (unknown).
    Category  Threshold  Accuracy Known  Accuracy Unknown  Accuracy Combin
ed
0  Landmarks          5        0.237624               0.0           0.2162
16
1  Landmarks         10        0.237624               0.3           0.2432
43
2  Landmarks         15        0.227723               0.7           0.2702
70
3  Landmarks         20        0.168317               0.9           0.2342
34
4  Landmarks         25        0.118812               0.9           0.1891
89
5  Landmarks         30        0.099010               0.9           0.1711
71
```

Discussion and Possible Improvement

In [24]:

```
# Plotting
import matplotlib.pyplot as plt
import seaborn as sns

fig, axes = plt.subplots(1, 3, figsize=(18, 5), sharey=True)

for ax, df, title in zip(axes,
                         [acc_df_book, acc_df_painting, acc_df_landmark],
                         ['Book Covers', 'Museum Paintings', 'Landmarks']
    ax.plot(df['Threshold'], df['Accuracy Known'], marker='o', label='Kno
    ax.plot(df['Threshold'], df['Accuracy Unknown'], marker='o', label='U
    ax.plot(df['Threshold'], df['Accuracy Combined'], marker='o', label='
    ax.set_title(title)
    ax.set_xlabel('Inlier Threshold')
    ax.set_ylim(0, 1)
    ax.grid(True)
    ax.legend()

axes[0].set_ylabel('Accuracy')
plt.suptitle('Accuracy vs Inlier Threshold Across Categories', fontsize=1
plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.show()
```



Your description of what you have done, and explanation of results, here

Step 5 discussion

Trade off between known vs unknown when changing inlier threshold for decision:

- As we raise the inlier threshold, "Unknown" accuracy (orange) climbs steeply—fewer false positives—because only very strong match sets remain.
- But "Known" accuracy (blue) falls off at the same time—some genuine objects now have too few inliers and get rejected.
- The "Combined" curve (green) peaks where these two effects balance, giving the highest overall correct ID rate.

Category-specific behavior:

- Book covers are easiest: combined accuracy peaks at threshold=10 at 0.7567. Above that, you start losing too many true covers.
- Museum paintings are in the middle: less textured, so even genuine matches seldom exceed ~0.5 combined. The best threshold is 10, beyond which known accuracy drops faster than unknown gains help.
- Landmarks are hardest: combined accuracy maxed out at 0.27. Genuine matches are so sparse that raising the threshold to catch unknowns almost kills known recognition.

Discussion:

- A single global inlier threshold cannot simultaneously satisfy all categories. What reliably flags "not in dataset" for books is too strict for paintings or landmarks.
- To improve, we could:
  - Use category-specific thresholds and parameters.
  - Enhance descriptor robustness (such as switch from ORB to SIFT).

Experiment with nfeatures on Landmarks dataset

To test if category-specific thresholds and parameters is true, I test it on the most difficult dataset to find a more optimal solution compared to the previous. I will test the following:

- Adjusting nfeatures
- Grid search to find the optimal threshold

In [39]:

```
import os
import cv2
import pandas as pd

additional_query_dir = "A2_smvs/extra_images/landmarks"
base_dir             = "A2_smvs/landmarks"
query_dir            = os.path.join(base_dir, "Query")
reference_dir        = os.path.join(base_dir, "Reference")

print(f"Additional Query Directory: {additional_query_dir}")
print(f"Query Directory:            {query_dir}")
print(f"Reference Directory:        {reference_dir}")

# Parameters
ORB_PARAMS      = {'nfeatures': 2000, 'scaleFactor': 1.2, 'nlevels': 8}
RATIO_THRESH    = 0.75
RANSAC_THRESH   = 7
MIN_MATCH_COUNT = 10
TOP_K           = 1

# Create ORB and BFMatcher
orb = cv2.ORB_create(**ORB_PARAMS)
bf  = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=False)

# Load reference features and known queries
reference_features, query_images = load_query_reference_images(reference_

# Load 'unknown' queries (extra images)
_, additional_query_images = load_query_reference_images(reference_dir, a

print(f"Loaded {len(reference_features)} reference images and {len(query_
print(f"Loaded {len(additional_query_images)} extra queries (unknown).")

# Match known queries
known_results = []
for q in query_images:
    best = find_best_match(q, reference_features, orb, bf,
                           MIN_MATCH_COUNT, RATIO_THRESH, RANSAC_THRESH,
    known_results.append((q, best))

# Match unknown queries
unknown_results = []
for q in additional_query_images:
    best = find_best_match(q, reference_features, orb, bf,
                           MIN_MATCH_COUNT, RATIO_THRESH, RANSAC_THRESH,
    unknown_results.append((q, best))

# Build DataFrames
df_known = pd.DataFrame(known_results, columns=['Query Image','Best Match
df_known['Ground Truth'] = df_known['Query Image'].apply(lambda p: os.pat

df_unknown = pd.DataFrame(unknown_results, columns=['Query Image','Best M
df_unknown['Ground Truth'] = "not in dataset"

# Sweep inlier thresholds and compute accuracy
from sklearn.model_selection import ParameterGrid

# Define a range of thresholds to search over
param_grid = {'threshold': range(10, 32, 2)}  # chosen based on repeated
```

```
records = []

# Use ParameterGrid to iterate over all combinations of parameters
for params in ParameterGrid(param_grid):
    thr = params['threshold']
    acc_known = calculate_accurate_occurance(df_known, inlier_threshold=t
    acc_unknown = calculate_accurate_occurance(df_unknown, inlier_thresho
    df_combined = pd.concat([df_known, df_unknown], ignore_index=True)
    acc_combined = calculate_accurate_occurance(df_combined, inlier_thres
    records.append({
        'Category': 'Landmarks',
        'Threshold': thr,
        'Accuracy Known': acc_known,
        'Accuracy Unknown': acc_unknown,
        'Accuracy Combined': acc_combined
    })


acc_df_landmark = pd.DataFrame(records)
print(acc_df_landmark)
```

```
Additional Query Directory: A2_smvs/extra_images/landmarks
Query Directory:            A2_smvs/landmarks/Query
Reference Directory:        A2_smvs/landmarks/Reference
Loaded 101 reference images and 101 known queries.
Loaded 10 extra queries (unknown).
     Category  Threshold  Accuracy Known  Accuracy Unknown  Accuracy Combi
ned
0   Landmarks         10        0.237624               0.1           0.225
225
1   Landmarks         12        0.237624               0.2           0.234
234
2   Landmarks         14        0.237624               0.5           0.261
261
3   Landmarks         16        0.227723               0.6           0.261
261
4   Landmarks         18        0.217822               0.8           0.270
270
5   Landmarks         20        0.207921               0.8           0.261
261
6   Landmarks         22        0.198020               0.8           0.252
252
7   Landmarks         24        0.178218               0.9           0.243
243
8   Landmarks         26        0.178218               0.9           0.243
243
9   Landmarks         28        0.158416               0.9           0.225
225
10  Landmarks         30        0.148515               0.9           0.216
216
```

I observe that the combined accuracy when increasing nfeatures did not increase. Therefore, I will test lowering nfeatures.

In [36]:
```
import os
import cv2
import pandas as pd

additional_query_dir = "A2_smvs/extra_images/landmarks"
base_dir             = "A2_smvs/landmarks"
```

```
query_dir             = os.path.join(base_dir, "Query")
reference_dir         = os.path.join(base_dir, "Reference")

print(f"Additional Query Directory: {additional_query_dir}")
print(f"Query Directory:            {query_dir}")
print(f"Reference Directory:        {reference_dir}")

# Parameters
ORB_PARAMS      = {'nfeatures': 1000, 'scaleFactor': 1.2, 'nlevels': 8}
RATIO_THRESH    = 0.75
RANSAC_THRESH   = 7
MIN_MATCH_COUNT = 10
TOP_K           = 1

# Create ORB and BFMatcher
orb = cv2.ORB_create(**ORB_PARAMS)
bf  = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=False)

# Load reference features and known queries
reference_features, query_images = load_query_reference_images(reference_

# Load 'unknown' queries (extra images)
_, additional_query_images = load_query_reference_images(reference_dir, a

print(f"Loaded {len(reference_features)} reference images and {len(query_
print(f"Loaded {len(additional_query_images)} extra queries (unknown).")

# Match known queries
known_results = []
for q in query_images:
    best = find_best_match(q, reference_features, orb, bf,
                           MIN_MATCH_COUNT, RATIO_THRESH, RANSAC_THRESH,
    known_results.append((q, best))

# Match unknown queries
unknown_results = []
for q in additional_query_images:
    best = find_best_match(q, reference_features, orb, bf,
                           MIN_MATCH_COUNT, RATIO_THRESH, RANSAC_THRESH,
    unknown_results.append((q, best))

# Build DataFrames
df_known = pd.DataFrame(known_results, columns=['Query Image','Best Match
df_known['Ground Truth'] = df_known['Query Image'].apply(lambda p: os.pat

df_unknown = pd.DataFrame(unknown_results, columns=['Query Image','Best M
df_unknown['Ground Truth'] = "not in dataset"

# Sweep inlier thresholds and compute accuracy
from sklearn.model_selection import ParameterGrid

# Define a range of thresholds to search over
param_grid = {'threshold': range(1, 21, 2)}  # Example range from 10 to 1
records = []

# Use ParameterGrid to iterate over all combinations of parameters
for params in ParameterGrid(param_grid):
    thr = params['threshold']
    acc_known = calculate_accurate_occurance(df_known, inlier_threshold=t
    acc_unknown = calculate_accurate_occurance(df_unknown, inlier_thresho
```

```
    df_combined = pd.concat([df_known, df_unknown], ignore_index=True)
    acc_combined = calculate_accurate_occurance(df_combined, inlier_thres
    records.append({
        'Category': 'Landmarks',
        'Threshold': thr,
        'Accuracy Known': acc_known,
        'Accuracy Unknown': acc_unknown,
        'Accuracy Combined': acc_combined
    })


acc_df_landmark = pd.DataFrame(records)
print(acc_df_landmark)
```

```
Additional Query Directory: A2_smvs/extra_images/landmarks
Query Directory:            A2_smvs/landmarks/Query
Reference Directory:        A2_smvs/landmarks/Reference
Loaded 101 reference images and 101 known queries.
Loaded 10 extra queries (unknown).
    Category  Threshold  Accuracy Known  Accuracy Unknown  Accuracy Combin
ed
0  Landmarks          1        0.297030               0.0           0.2702
70
1  Landmarks          3        0.297030               0.0           0.2702
70
2  Landmarks          5        0.297030               0.0           0.2702
70
3  Landmarks          7        0.297030               0.0           0.2702
70
4  Landmarks          9        0.277228               0.7           0.3153
15
5  Landmarks         11        0.247525               0.8           0.2972
97
6  Landmarks         13        0.207921               0.8           0.2612
61
7  Landmarks         15        0.168317               0.9           0.2342
34
8  Landmarks         17        0.138614               0.9           0.2072
07
9  Landmarks         19        0.118812               0.9           0.1891
89
```

By lowering the nfeatures to 1000, we observed that the maximum combined accuracy we could achieve is 0.3153. I will further test if lowering the nfeatures even more result in diminishing return for the accuracy.

In [40]:

```
import os
import cv2
import pandas as pd

additional_query_dir = "A2_smvs/extra_images/landmarks"
base_dir             = "A2_smvs/landmarks"
query_dir            = os.path.join(base_dir, "Query")
reference_dir        = os.path.join(base_dir, "Reference")

print(f"Additional Query Directory: {additional_query_dir}")
print(f"Query Directory:            {query_dir}")
print(f"Reference Directory:        {reference_dir}")
```

```python
# Parameters
ORB_PARAMS      = {'nfeatures': 700, 'scaleFactor': 1.2, 'nlevels': 8}
RATIO_THRESH    = 0.75
RANSAC_THRESH   = 7
MIN_MATCH_COUNT = 10
TOP_K           = 1

# Create ORB and BFMatcher
orb = cv2.ORB_create(**ORB_PARAMS)
bf  = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=False)

# Load reference features and known queries
reference_features, query_images = load_query_reference_images(reference_

# Load 'unknown' queries (extra images)
_, additional_query_images = load_query_reference_images(reference_dir, a

print(f"Loaded {len(reference_features)} reference images and {len(query_
print(f"Loaded {len(additional_query_images)} extra queries (unknown).")

# Match known queries
known_results = []
for q in query_images:
    best = find_best_match(q, reference_features, orb, bf,
                           MIN_MATCH_COUNT, RATIO_THRESH, RANSAC_THRESH,
    known_results.append((q, best))

# Match unknown queries
unknown_results = []
for q in additional_query_images:
    best = find_best_match(q, reference_features, orb, bf,
                           MIN_MATCH_COUNT, RATIO_THRESH, RANSAC_THRESH,
    unknown_results.append((q, best))

# Build DataFrames
df_known = pd.DataFrame(known_results, columns=['Query Image','Best Match
df_known['Ground Truth'] = df_known['Query Image'].apply(lambda p: os.pat

df_unknown = pd.DataFrame(unknown_results, columns=['Query Image','Best M
df_unknown['Ground Truth'] = "not in dataset"

# Sweep inlier thresholds and compute accuracy
from sklearn.model_selection import ParameterGrid

# Define a range of thresholds to search over
param_grid = {'threshold': range(1, 21, 2)}  # Example range from 10 to 1
records = []

# Use ParameterGrid to iterate over all combinations of parameters
for params in ParameterGrid(param_grid):
    thr = params['threshold']
    acc_known = calculate_accurate_occurance(df_known, inlier_threshold=t
    acc_unknown = calculate_accurate_occurance(df_unknown, inlier_thresho
    df_combined = pd.concat([df_known, df_unknown], ignore_index=True)
    acc_combined = calculate_accurate_occurance(df_combined, inlier_thres
    records.append({
        'Category': 'Landmarks',
        'Threshold': thr,
        'Accuracy Known': acc_known,
        'Accuracy Unknown': acc_unknown,
```

```
        'Accuracy Combined': acc_combined
    })


acc_df_landmark = pd.DataFrame(records)
print(acc_df_landmark)
```

```
Additional Query Directory: A2_smvs/extra_images/landmarks
Query Directory:            A2_smvs/landmarks/Query
Reference Directory:        A2_smvs/landmarks/Reference
Loaded 101 reference images and 101 known queries.
Loaded 10 extra queries (unknown).
    Category  Threshold  Accuracy Known  Accuracy Unknown  Accuracy Combin
ed
0  Landmarks          1        0.297030               0.1           0.2792
79
1  Landmarks          3        0.297030               0.1           0.2792
79
2  Landmarks          5        0.297030               0.1           0.2792
79
3  Landmarks          7        0.277228               0.3           0.2792
79
4  Landmarks          9        0.247525               0.7           0.2882
88
5  Landmarks         11        0.188119               0.8           0.2432
43
6  Landmarks         13        0.138614               0.9           0.2072
07
7  Landmarks         15        0.099010               1.0           0.1801
80
8  Landmarks         17        0.099010               1.0           0.1801
80
9  Landmarks         19        0.089109               1.0           0.1711
71
```

Step 6 discussion

- Experiment setup
  We varied ORB nfeatures (2000, 1000, 700) and swept the inlier-count threshold used to decide whether a match is “in-dataset.” For each combination, we computed:
  - Known accuracy: fraction of true landmarks correctly accepted (inliers >= threshold)
  - Unknown accuracy: fraction of extra (non-landmark) queries correctly rejected (inliers < threshold)
  - Combined accuracy: overall correct decision rate
- Results summary
  - nfeatures = 2000
    - Best combined accuracy ≈ 0.27 at threshold = 18
    - Known accuracy drops as threshold rises; unknown accuracy reaches 0.8–0.9 only at high thresholds
  - nfeatures = 1000
    - Best combined accuracy ≈ 0.315 at threshold = 9

        - Known accuracy stays high for thresholds up to 9; unknown accuracy climbs sharply between 7–11
    - nfeatures = 700
        - Best combined accuracy ≃ 0.288 at threshold = 9
        - Further reducing features hurts known-query recall without much gain on unknowns
- Interpretation
  Raising ORB feature count to 2000 added noisy descriptors that boosted false positives, forcing a higher inlier cutoff and reducing true-positive recall. Cutting features too much (700) reduced genuine matches and also lowered recall. The sweet spot was at 1000 features with an inlier threshold of 9, where combined accuracy was maximized by balancing acceptance of true landmarks and rejection of unknowns.

- Conclusion:
  Image retrieval requires per-category tuning of both detector parameters and inlier-count thresholds. Adjusting ORB feature budget and selecting an appropriate decision threshold can significantly improve overall identification performance.

Question 3: FUndametal Matrix, Epilines and Retrival (optional, assesed for granting up to 25% bonus marks for the A2)

In this question, the aim is to accuractely estimate the fundamental matrix given two views of a scene, visualise the corresponding epipolar lines and use the inlier count of fundametal matrix for retrival.

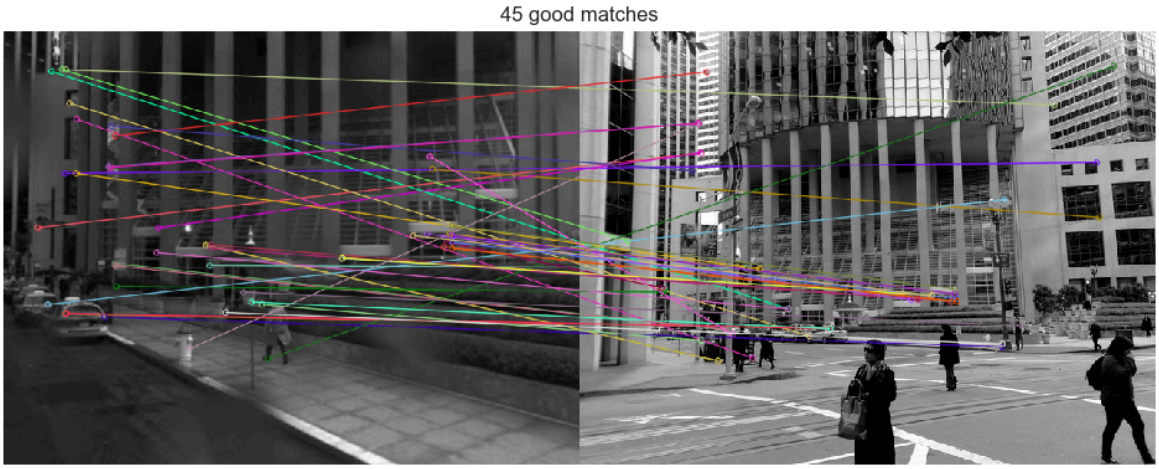
The steps are as follows:

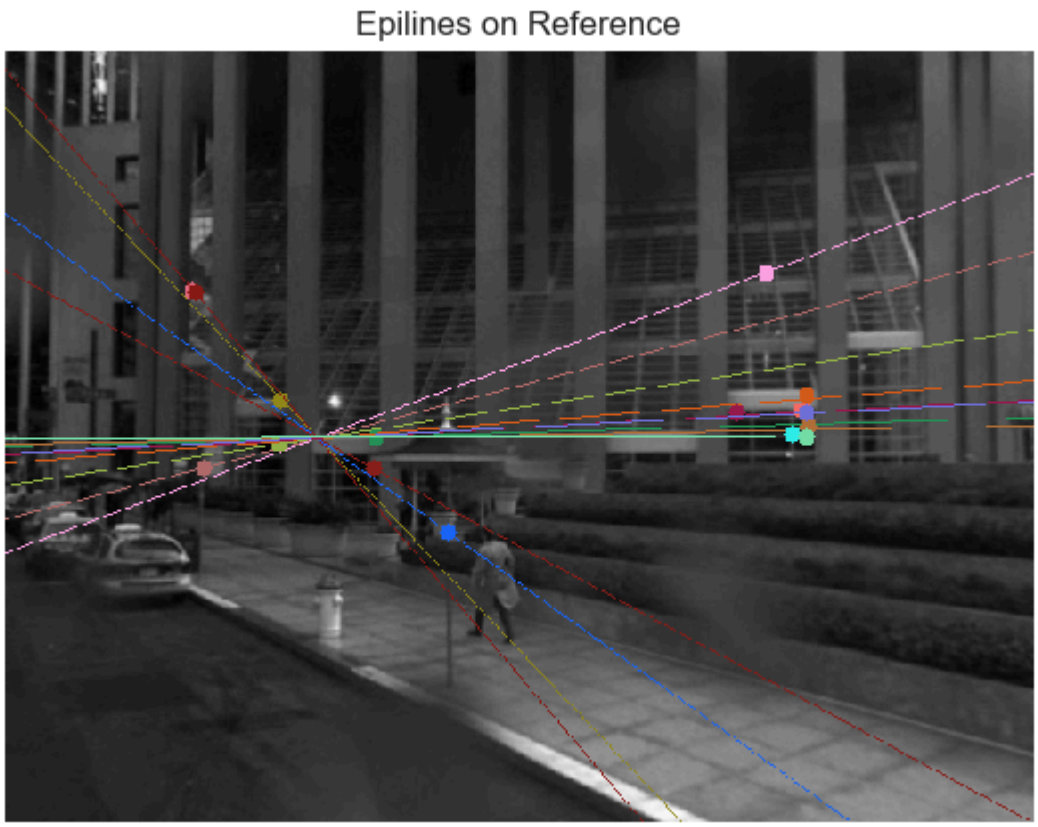
1. Select two images of the same scene (query and reference) from the landmark dataset and find the matches as you have done in Question 1 (1.1-1.4).
2. Compute fundametal metrix with good matches (after appying ratio test) using the opencv function cv.findFundamentalMat(). Use both 8 point algorithm and RANSAC assisted 8 point algorithm to compute fundamental matrix.
3. Hint: You need minimum 8 matches to be able to use the function. Ignore pairs where 8 mathes are not found.
4. Visualise the epipolar lines for the matched features and analyse the results. You can use openCV function cv.computeCorrespondEpilines() to estimate the epilines. We have provided the code for drawing these epilines in function drawlines() that you can modify as required.

In [68]:
```
import cv2, numpy as np
import matplotlib.pyplot as plt

# Load and find features
```

```python
ref = cv2.imread('A2_smvs/landmarks/Reference/001.jpg',0)
qry = cv2.imread('A2_smvs/landmarks/Query/001.jpg',0)
orb = cv2.ORB_create(nfeatures=3000)
kp1, d1 = orb.detectAndCompute(ref, None)
kp2, d2 = orb.detectAndCompute(qry, None)

bf = cv2.BFMatcher(cv2.NORM_HAMMING)
knn = bf.knnMatch(d1, d2, k=2)
good = [m for m,n in knn if m.distance < 0.8*n.distance]

# show matches
plt.figure(figsize=(12,5))
imgm = cv2.drawMatches(ref,kp1,qry,kp2,good,None,flags=2)
plt.imshow(imgm); plt.title(f'{len(good)} good matches'); plt.axis('off')

# Compute F with and without RANSAC
if len(good) >= 8:
    pts1 = np.float32([kp1[m.queryIdx].pt for m in good]).reshape(-1,1,2)
    pts2 = np.float32([kp2[m.trainIdx].pt for m in good]).reshape(-1,1,2)

    F8, _       = cv2.findFundamentalMat(pts1, pts2, cv2.FM_8POINT)
    F_ransac, mask = cv2.findFundamentalMat(
        pts1, pts2, cv2.FM_RANSAC, ransacReprojThreshold=3.0, confidence=
    )
    inliers1 = pts1[mask.ravel()==1]
    inliers2 = pts2[mask.ravel()==1]
    print("8-point F:\n", F8)
    print("RANSAC F (inliers):", int(mask.sum()))

# Visualize epilines for RANSAC inliers
if F_ransac is not None:
    lines1 = cv2.computeCorrespondEpilines(inliers2.reshape(-1,1,2), 2, F
    lines1 = lines1.reshape(-1,3)
    epi1, _ = drawlines(ref, qry, lines1, inliers1.squeeze().astype(int),
    plt.figure(figsize=(12,5)); plt.imshow(epi1); plt.title('Epilines on
```

```
8-point F:
 [[ 2.61867697e-05 -1.08941721e-05 -1.26866410e-03]
 [ 1.52242310e-04 -9.79003850e-05 -6.88919003e-04]
 [-5.36587726e-02  3.05915926e-02  1.00000000e+00]]
RANSAC F (inliers): 16
```



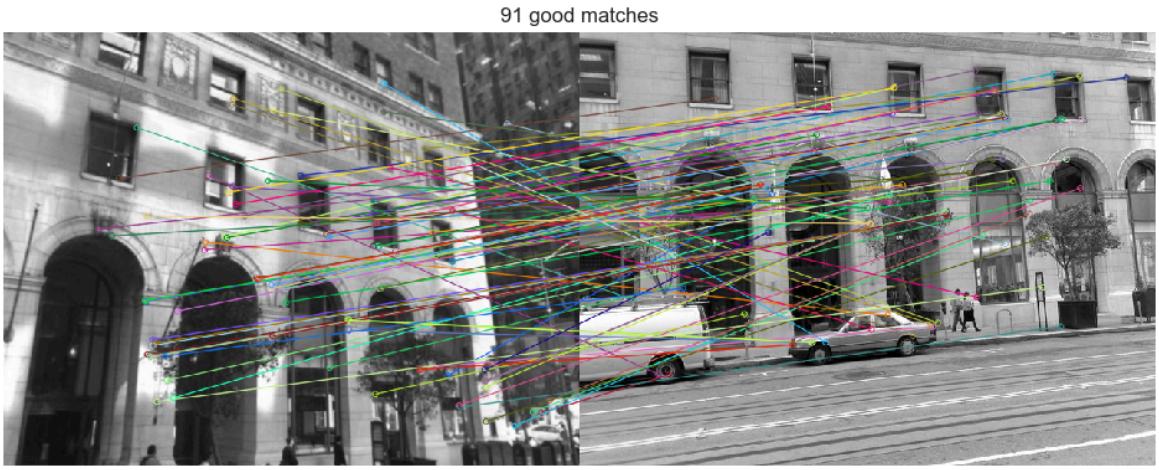


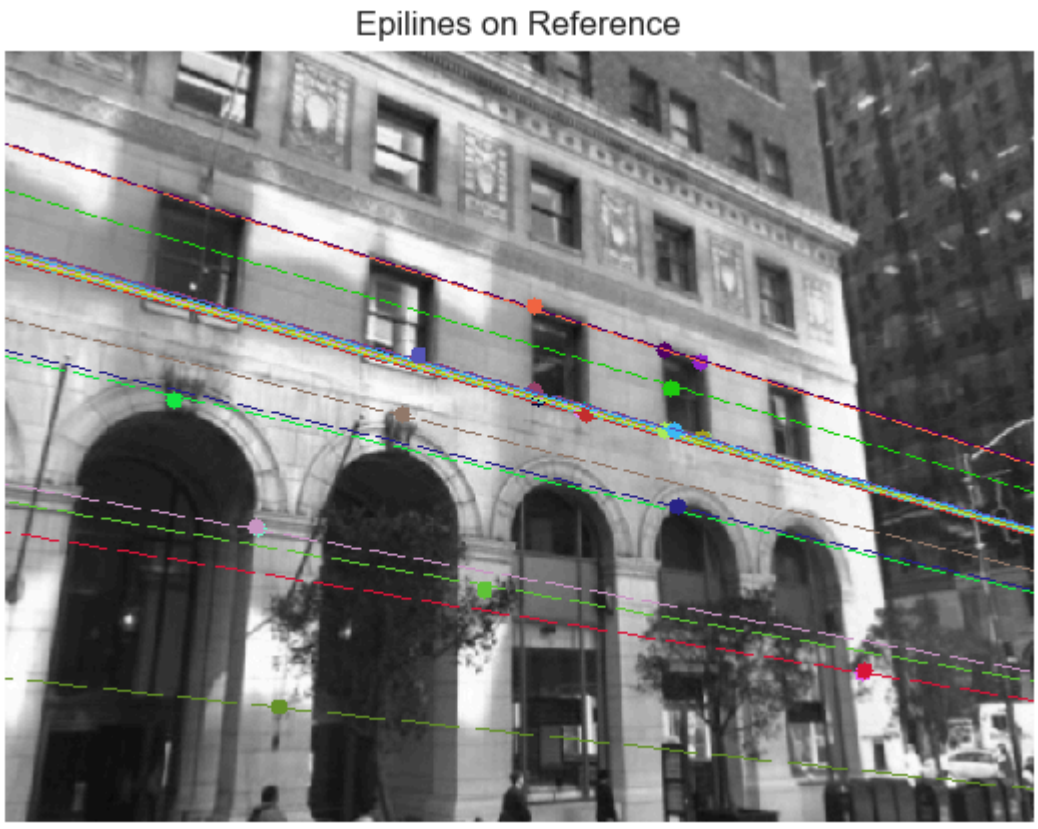
4. Repeat the steps for some examples from the landmarks datasets.

In [67]:
```
# Your code for additional images goes here
import cv2, numpy as np
import matplotlib.pyplot as plt

# Load and find features
ref = cv2.imread('A2_smvs/landmarks/Reference/008.jpg',0)
qry = cv2.imread('A2_smvs/landmarks/Query/008.jpg',0)
orb = cv2.ORB_create(nfeatures=3000)
kp1, d1 = orb.detectAndCompute(ref, None)
kp2, d2 = orb.detectAndCompute(qry, None)

bf = cv2.BFMatcher(cv2.NORM_HAMMING)
knn = bf.knnMatch(d1, d2, k=2)
good = [m for m,n in knn if m.distance < 0.8*n.distance]

# show matches
plt.figure(figsize=(12,5))
imgm = cv2.drawMatches(ref,kp1,qry,kp2,good,None,flags=2)
plt.imshow(imgm); plt.title(f'{len(good)} good matches'); plt.axis('off')

# Compute F with and without RANSAC
if len(good) >= 8:
    pts1 = np.float32([kp1[m.queryIdx].pt for m in good]).reshape(-1,1,2)
    pts2 = np.float32([kp2[m.trainIdx].pt for m in good]).reshape(-1,1,2)

    F8, _ = cv2.findFundamentalMat(pts1, pts2, cv2.FM_8POINT)
    F_ransac, mask = cv2.findFundamentalMat(
        pts1, pts2, cv2.FM_RANSAC, ransacReprojThreshold=3.0, confidence=
    )
    inliers1 = pts1[mask.ravel()==1]
    inliers2 = pts2[mask.ravel()==1]
```

```
    print("8-point F:\n", F8)
    print("RANSAC F (inliers):", int(mask.sum()))

# Visualize epilines for RANSAC inliers
if F_ransac is not None:
    lines1 = cv2.computeCorrespondEpilines(inliers2.reshape(-1,1,2), 2, F
    lines1 = lines1.reshape(-1,3)
    epi1, _ = drawlines(ref, qry, lines1, inliers1.squeeze().astype(int),
    plt.figure(figsize=(12,5)); plt.imshow(epi1); plt.title('Epilines on
```

```
8-point F:
 [[-6.92677695e-07  4.37848652e-06 -7.14298617e-04]
 [-8.33959666e-06  2.38319500e-05 -1.74919569e-03]
 [ 1.78077642e-03 -7.73026267e-03  1.00000000e+00]]
RANSAC F (inliers): 26
```

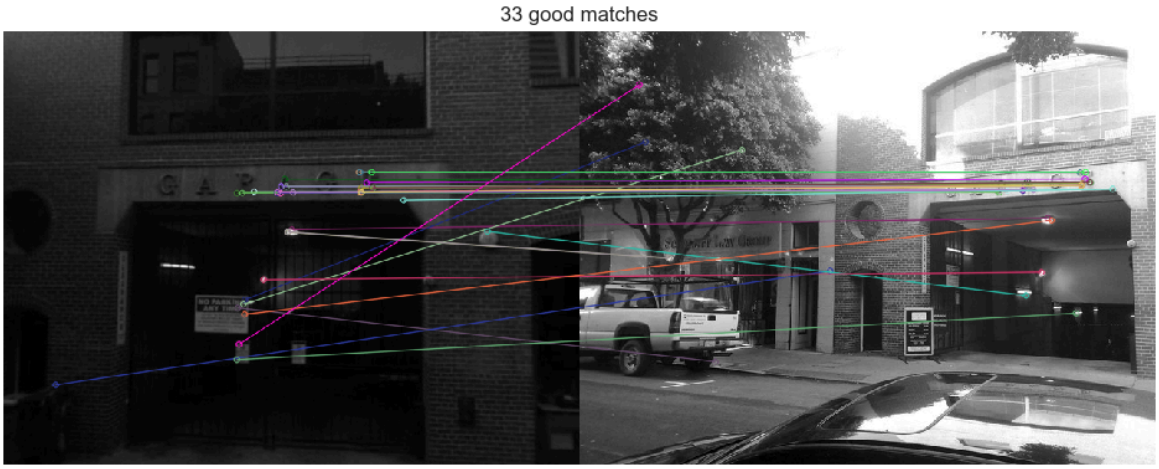


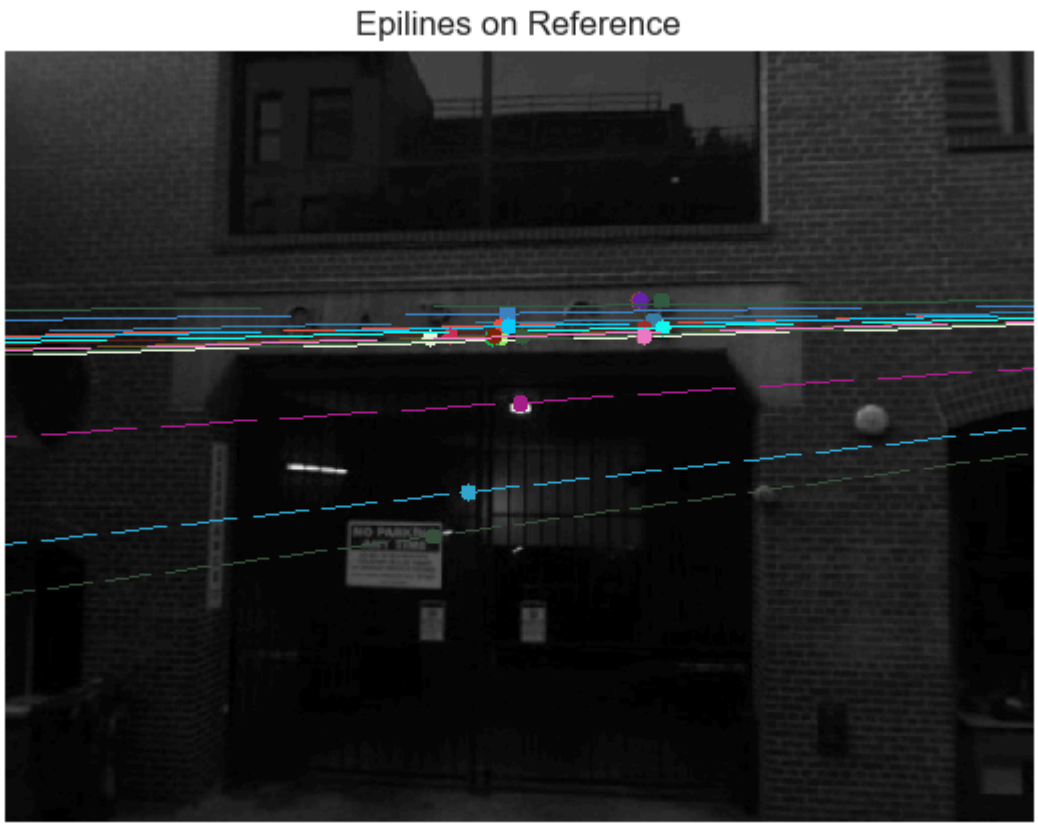


In [ ]:
```
# Your code for additional images goes here
import cv2, numpy as np
import matplotlib.pyplot as plt

# Load and find features
```

```python
ref = cv2.imread('A2_smvs/landmarks/Reference/060.jpg',0)
qry = cv2.imread('A2_smvs/landmarks/Query/060.jpg',0)
orb = cv2.ORB_create(nfeatures=3000)
kp1, d1 = orb.detectAndCompute(ref, None)
kp2, d2 = orb.detectAndCompute(qry, None)

bf = cv2.BFMatcher(cv2.NORM_HAMMING)
knn = bf.knnMatch(d1, d2, k=2)
good = [m for m,n in knn if m.distance < 0.8*n.distance]

# show matches
plt.figure(figsize=(12,5))
imgm = cv2.drawMatches(ref,kp1,qry,kp2,good,None,flags=2)
plt.imshow(imgm); plt.title(f'{len(good)} good matches'); plt.axis('off')

# Compute F with and without RANSAC
if len(good) >= 8:
    pts1 = np.float32([kp1[m.queryIdx].pt for m in good]).reshape(-1,1,2)
    pts2 = np.float32([kp2[m.trainIdx].pt for m in good]).reshape(-1,1,2)

    F8, _       = cv2.findFundamentalMat(pts1, pts2, cv2.FM_8POINT)
    F_ransac, mask = cv2.findFundamentalMat(
        pts1, pts2, cv2.FM_RANSAC, ransacReprojThreshold=3.0, confidence=
    )
    inliers1 = pts1[mask.ravel()==1]
    inliers2 = pts2[mask.ravel()==1]
    print("8-point F:\n", F8)
    print("RANSAC F (inliers):", int(mask.sum()))

# Visualize epilines for RANSAC inliers
if F_ransac is not None:
    lines1 = cv2.computeCorrespondEpilines(inliers2.reshape(-1,1,2), 2, F
    lines1 = lines1.reshape(-1,3)
    epi1, _ = drawlines(ref, qry, lines1, inliers1.squeeze().astype(int),
    plt.figure(figsize=(12,5)); plt.imshow(epi1); plt.title('Epilines on
```

```
8-point F:
 [[ 3.73435844e-07  8.99865248e-07 -4.57503851e-04]
 [ 3.80035732e-06  1.03632205e-05 -4.34259438e-03]
 [-8.64125115e-04 -2.30797887e-03  1.00000000e+00]]
RANSAC F (inliers): 23
```

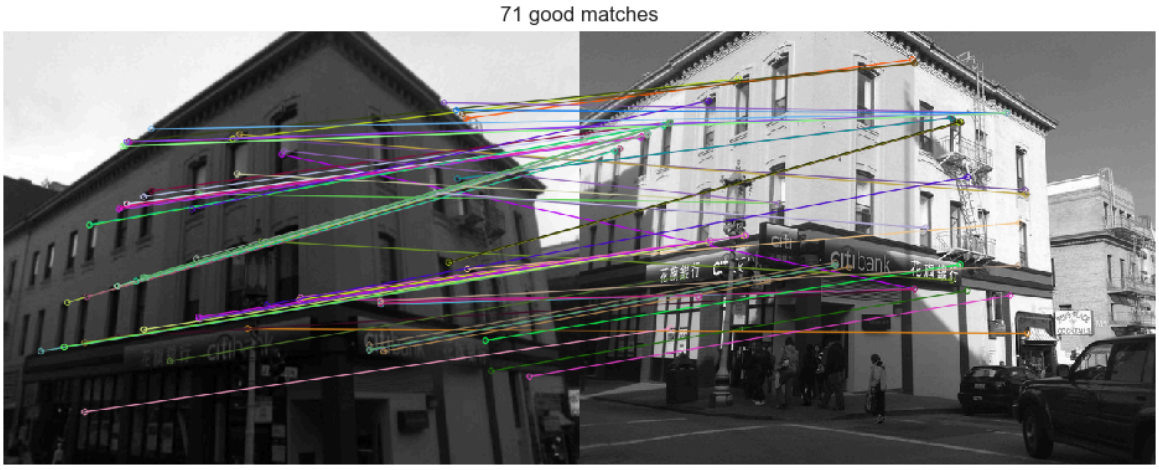


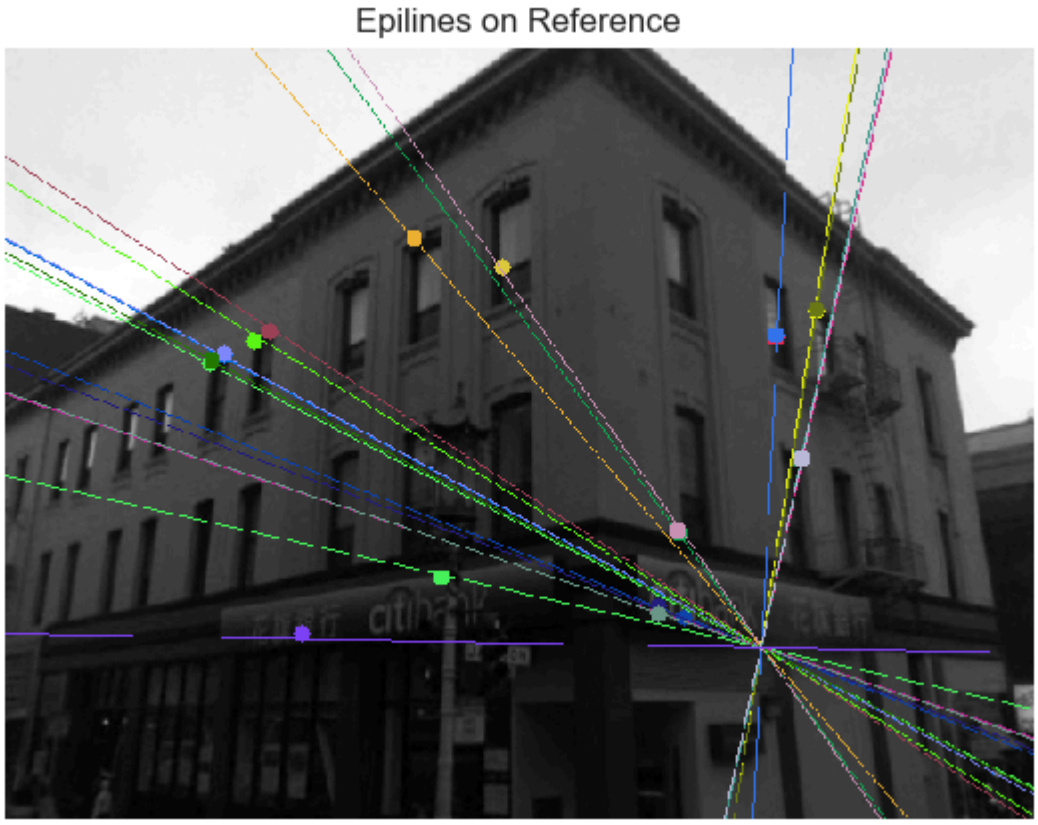


In [69]:
```python
# Your code for additional images goes here
import cv2, numpy as np
import matplotlib.pyplot as plt

# Load and find features
ref = cv2.imread('A2_smvs/landmarks/Reference/082.jpg',0)
qry = cv2.imread('A2_smvs/landmarks/Query/082.jpg',0)
orb = cv2.ORB_create(nfeatures=3000)
kp1, d1 = orb.detectAndCompute(ref, None)
kp2, d2 = orb.detectAndCompute(qry, None)

bf = cv2.BFMatcher(cv2.NORM_HAMMING)
knn = bf.knnMatch(d1, d2, k=2)
good = [m for m,n in knn if m.distance < 0.8*n.distance]

# show matches
plt.figure(figsize=(12,5))
imgm = cv2.drawMatches(ref,kp1,qry,kp2,good,None,flags=2)
plt.imshow(imgm); plt.title(f'{len(good)} good matches'); plt.axis('off')

# Compute F with and without RANSAC
if len(good) >= 8:
    pts1 = np.float32([kp1[m.queryIdx].pt for m in good]).reshape(-1,1,2)
    pts2 = np.float32([kp2[m.trainIdx].pt for m in good]).reshape(-1,1,2)

    F8, _       = cv2.findFundamentalMat(pts1, pts2, cv2.FM_8POINT)
    F_ransac, mask = cv2.findFundamentalMat(
        pts1, pts2, cv2.FM_RANSAC, ransacReprojThreshold=3.0, confidence=
    )
    inliers1 = pts1[mask.ravel()==1]
    inliers2 = pts2[mask.ravel()==1]
    print("8-point F:\n", F8)
    print("RANSAC F (inliers):", int(mask.sum()))
```

```python
# Visualize epilines for RANSAC inliers
if F_ransac is not None:
    lines1 = cv2.computeCorrespondEpilines(inliers2.reshape(-1,1,2), 2, F
    lines1 = lines1.reshape(-1,3)
    epi1, _ = drawlines(ref, qry, lines1, inliers1.squeeze().astype(int),
    plt.figure(figsize=(12,5)); plt.imshow(epi1); plt.title('Epilines on
```

```
8-point F:
 [[ 4.82058737e-07  1.21640167e-06 -9.78373582e-04]
 [ 6.56967416e-07  1.22405248e-05  6.94506837e-05]
 [-7.39260545e-04 -5.64028203e-03  1.00000000e+00]]
RANSAC F (inliers): 24
```





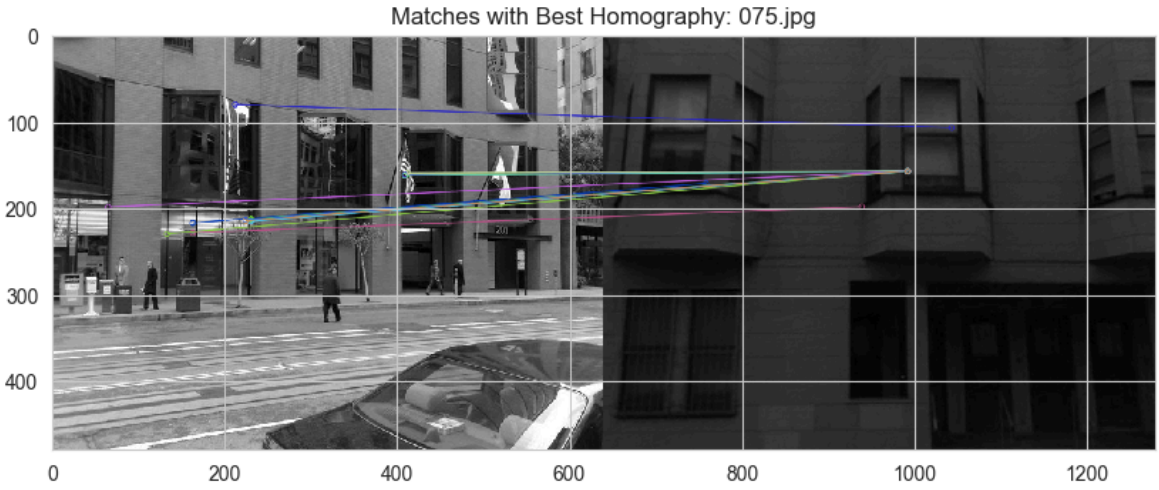
5. Find a query from landmarks data for which the retrival in Q2 failed. Attempt the retrival with replacing the Homography + RANSAC method of Q2 to Fundamental Matrix + RANSAC method using code written above. Does the change of the model makes retraival successful? Analyse and comment.

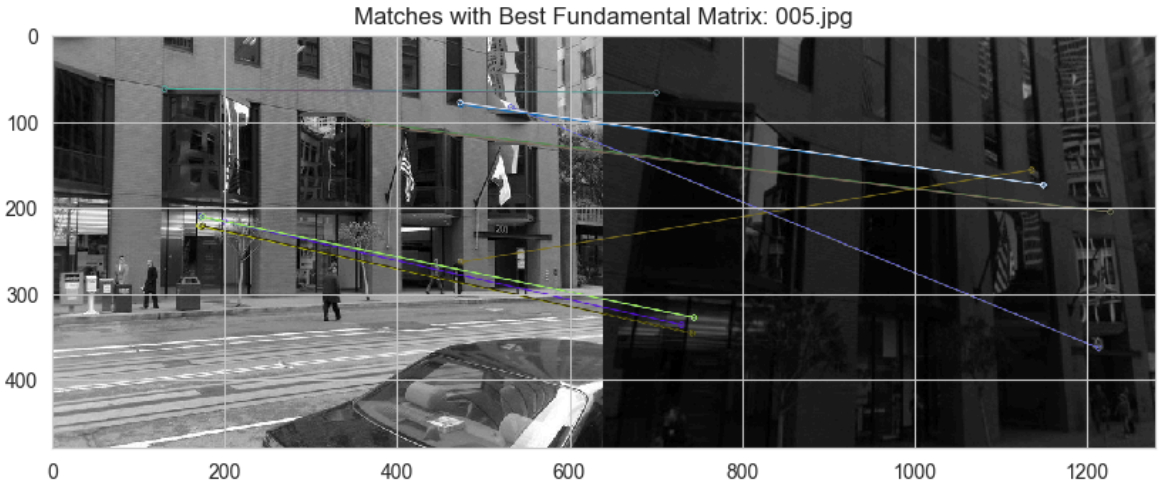
In [64]:

```
import os
import cv2
import numpy as np

# known false matches of homography
query_path = 'A2_smvs/landmarks/Query/005.jpg'
query_name = os.path.basename(query_path)

# setup orb and bfmatcher
orb = cv2.ORB_create(nfeatures=1500)
bf  = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=False)

# load and find keypoints and descriptors
img_q = cv2.imread(query_path, cv2.IMREAD_GRAYSCALE)
kp_q, des_q = orb.detectAndCompute(img_q, None)

# find best homography match
best_homo = ('<none>', 0)
ref_dir = 'A2_smvs/landmarks/Reference'
for fn in os.listdir(ref_dir):
    if not fn.lower().endswith('.jpg'):
        continue
    ref_path = os.path.join(ref_dir, fn)
    img_r = cv2.imread(ref_path, cv2.IMREAD_GRAYSCALE)
    kp_r, des_r = orb.detectAndCompute(img_r, None)
    if des_q is None or des_r is None:
        continue

    # KNN + ratio test
    knn = bf.knnMatch(des_q, des_r, k=2)
    good = [m for m,n in knn if m.distance < 0.75 * n.distance]
    if len(good) < 10:
        continue

    # build point arrays
    pts_q = np.float32([ kp_q[m.queryIdx].pt for m in good ]).reshape(-1,
    pts_r = np.float32([ kp_r[m.trainIdx].pt for m in good ]).reshape(-1,

    # estimate homography + count inliers
    H, mask = cv2.findHomography(pts_q, pts_r, cv2.RANSAC, 7.0)
    if mask is None:
        continue
    inliers = int(mask.sum())
    if inliers > best_homo[1]:
        best_homo = (fn, inliers)

# find best fundamental matrix match
best_fm = ('<none>', 0)
for fn in os.listdir(ref_dir):
    if not fn.lower().endswith('.jpg'):
        continue
    ref_path = os.path.join(ref_dir, fn)
    img_r = cv2.imread(ref_path, cv2.IMREAD_GRAYSCALE)
    kp_r, des_r = orb.detectAndCompute(img_r, None)
    if des_q is None or des_r is None:
        continue

    # same ratio-test filtering
    knn = bf.knnMatch(des_q, des_r, k=2)
```

```
    good = [m for m,n in knn if m.distance < 0.75 * n.distance]
    if len(good) < 10:
        continue

    pts_q = np.float32([ kp_q[m.queryIdx].pt for m in good ]).reshape(-1,
    pts_r = np.float32([ kp_r[m.trainIdx].pt for m in good ]).reshape(-1,

    # estimate fundamental matrix + count inliers
    F, mask = cv2.findFundamentalMat(pts_q, pts_r, cv2.FM_RANSAC, 3.0, 0.
    if mask is None:
        continue
    inliers = int(mask.sum())
    if inliers > best_fm[1]:
        best_fm = (fn, inliers)

# print results
print(f"Query image: {query_name}")
print(f"Top-1 by homography: {best_homo[0]} (inliers = {best_homo[1]})")
print(f"Top-1 by fundamental: {best_fm[0]} (inliers = {best_fm[1]})")

# matches between the query image and the best homography match
if best_homo[0] != '<none>':
    best_homo_path = os.path.join(ref_dir, best_homo[0])
    img_best_homo = cv2.imread(best_homo_path, cv2.IMREAD_GRAYSCALE)
    kp_best_homo, des_best_homo = orb.detectAndCompute(img_best_homo, Non
    knn_homo = bf.knnMatch(des_q, des_best_homo, k=2)
    good_homo = [m for m, n in knn_homo if m.distance < 0.75 * n.distance
    pts_q_homo = np.float32([kp_q[m.queryIdx].pt for m in good_homo]).res
    pts_r_homo = np.float32([kp_best_homo[m.trainIdx].pt for m in good_ho
    H, mask_homo = cv2.findHomography(pts_q_homo, pts_r_homo, cv2.RANSAC,
    inlier_matches_homo = [good_homo[i] for i in range(len(good_homo)) if
    img_matches_homo = cv2.drawMatches(img_q, kp_q, img_best_homo, kp_bes
    plt.figure(figsize=(10, 8))
    plt.title(f"Matches with Best Homography: {best_homo[0]}")
    plt.imshow(img_matches_homo, cmap='gray')
    plt.show()

# matches between the query image and the best fundamental matrix match
if best_fm[0] != '<none>':
    best_fm_path = os.path.join(ref_dir, best_fm[0])
    img_best_fm = cv2.imread(best_fm_path, cv2.IMREAD_GRAYSCALE)
    kp_best_fm, des_best_fm = orb.detectAndCompute(img_best_fm, None)
    knn_fm = bf.knnMatch(des_q, des_best_fm, k=2)
    good_fm = [m for m, n in knn_fm if m.distance < 0.75 * n.distance]
    pts_q_fm = np.float32([kp_q[m.queryIdx].pt for m in good_fm]).reshape
    pts_r_fm = np.float32([kp_best_fm[m.trainIdx].pt for m in good_fm]).r
    F, mask_fm = cv2.findFundamentalMat(pts_q_fm, pts_r_fm, cv2.FM_RANSAC
    inlier_matches_fm = [good_fm[i] for i in range(len(good_fm)) if mask_
    img_matches_fm = cv2.drawMatches(img_q, kp_q, img_best_fm, kp_best_fm
    plt.figure(figsize=(10, 8))
    plt.title(f"Matches with Best Fundamental Matrix: {best_fm[0]}")
    plt.imshow(img_matches_fm, cmap='gray')
    plt.show()
```

```
Query image: 005.jpg
Top-1 by homography: 075.jpg (inliers = 13)
Top-1 by fundamental: 005.jpg (inliers = 17)
```





Your analysis goes here

Step 5 discussion

Experiment setup:

Query image 005.jpg (a known fail homography example) was matched against all reference images using ORB+BFMatcher. Two geometric verification methods were applied:

- Homography+RANSAC
- FundamentalMat+RANSAC

Results summary

- Homography verification returned 075.jpg as the top-1 match (incorrect).
- Fundamental-matrix returned 005.jpg as the top-1 match (correct).

Interpretation

- Homography enforces a single planar warp, which can overfit subsets of correspondences in a richly 3D scene and admit false positives. The fundamental matrix enforces epipolar constraints between two arbitrary camera

views, filtering out matches that violate true 3D geometry and thus recovering the correct correspondence.

Conclusion:

- In environments with substantial depth variation (example: street-level landmarks), fundamental-matrix RANSAC provides more reliable retrieval than planar homography. Appropriate tuning of ratio-test thresholds and RANSAC parameters is essential, and combining both verification models can further improve robustness.