Computer Vision 2022 Assignment 2: Image matching and retrieval

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

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

This assignment will have a minimum hurdle of 40%. You will fail if you can not reach the minimum hurdle.

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.re
move_cell_tags 'remove-cell'
# Or
jupyter nbconvert Assignment2.ipynb --TagRemovePreprocessor.remove_cell_tags='{"re
move_cell"}'
```

Please do try this ccommond early before the last day! As the commend may be a little bit difference depending on your computer and the environment.

This will also remove the preamble text from each question. 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.

This being the second assignment, we have provided less strict direction this time and you have more flexibility to choose how you answer each question. However you still need to ensure the outputs and report are clear and easy to read. 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. Within each category, 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 (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]:

```
# Numpy is the main package for scientific computing with Python.
import numpy as np
import cv2

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

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

In [2]:

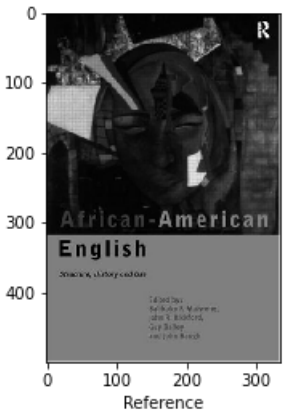
```
def draw_outline(ref, query, model):
    """
        Draw outline of reference image in the query image.
        This is just an example to show the steps involved.
        You can modify to suit your needs.
        Inputs:
            ref: reference image
            query: query image
        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

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

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

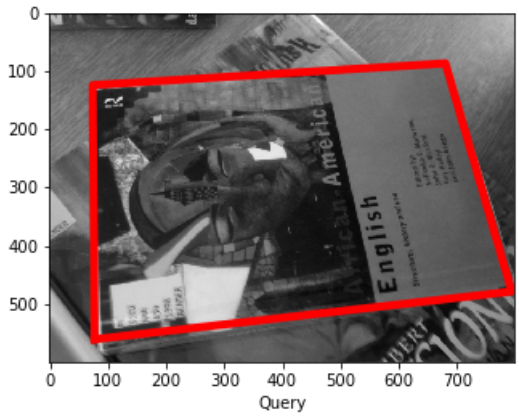
Question 1: Matching an object in a pair of images (45%)

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

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 (https://www.researchgate.net/publication/221111151_ORB_an_efficient_alternative_to_SIFT_or_SURF)](https://www.researchgate.net/publication/221111151_ORB_an_efficient_alternative_to_SIFT_or_SURF) by Rublee et al. You don't need to understand all the details, but try to get an idea of how it works. ORB combines the FAST corner detector (covered in week 4) and the BRIEF descriptor. BRIEF is based on similar ideas to the SIFT descriptor we covered week 4, but with some changes for efficiency.
1. [Load images] Load the first (reference, query) image pair from the "book_covers" category using opencv (e.g. `img=cv2.imread()`). Check the parameter option in " cv2.imread()" to ensure that you read the gray scale image, since it is necessary for computing ORB features.
2. [Detect features] Create opencv ORB feature extractor by `orb=cv2.ORB_create()` . Then you can detect keypoints by `kp = orb.detect(img,None)` , and compute descriptors by `kp, des = orb.compute(img, kp)` . You need to do this for each image, and then you can use `cv2.drawKeypoints()` for visualization.
3. [Match features] As ORB is a binary feature, you need to use HAMMING distance for matching, e.g., `bf = cv2.BFMatcher(cv2.NORM_HAMMING)` . Then you are requried to do KNN matching (k=2) by using `bf.knnMatch()` . After that, you are required to use "ratio_test" to find good matches. By default, you can set `ratio=0.8` .
4. [Plot and analyze] You need to visualize the matches by using the `cv2.drawMatches()` function. Also you can change the ratio values, parameters in `cv2.ORB_create()` , and distance functions in `cv2.BFMatcher()` . Please discuss how these changes influence the match numbers.

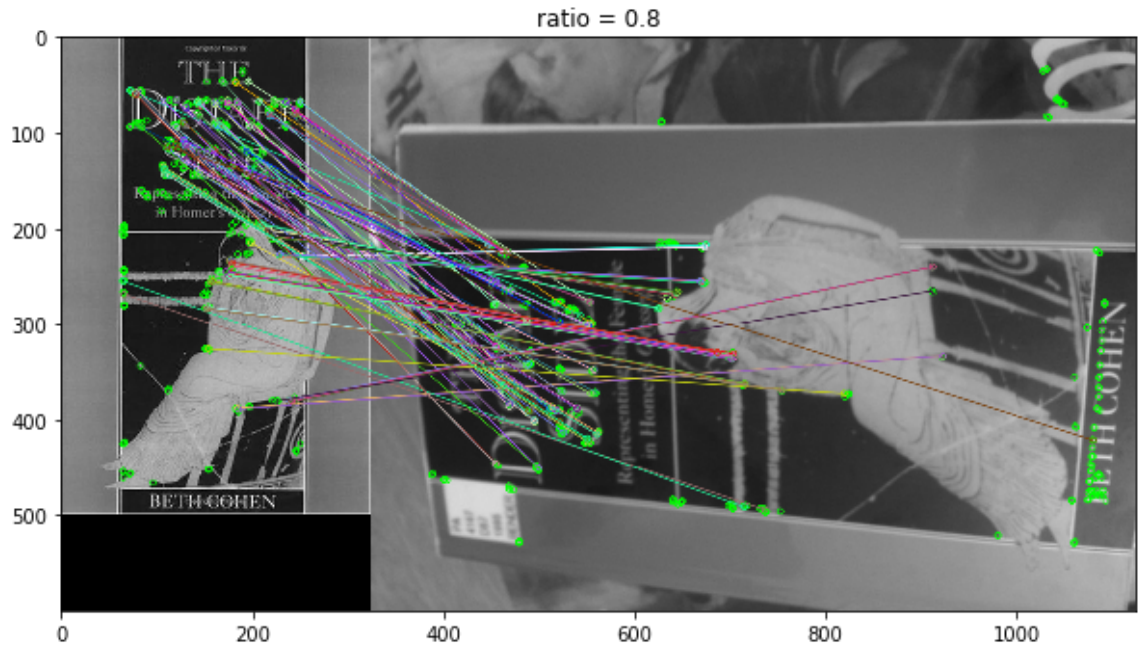
In [3]:

```
# Your code for descriptor matching tests here
import numpy as np
from matplotlib import pyplot as plt

# load image at gray scale
ref = cv2.imread('book_covers/Reference/001.jpg',0)
que = cv2.imread('book_covers/Query/001.jpg',0)

# compute detector and descriptor
orb = cv2.ORB_create()
# find the keypoints and descriptors with ORB
kp_ref = orb.detect(ref,None)
kp_que = orb.detect(que,None)

kp_ref,des_ref = orb.compute(ref,kp_ref)
kp_que,des_que = orb.compute(que,kp_que)

# draw keypoints
ref_img2 = cv2.drawKeypoints(ref,kp_ref,None,color=(0,255,0),flags=0)
que_img2 = cv2.drawKeypoints(que,kp_que,None,color=(0,255,0),flags=0)

# create BFMatcher object
bf = cv2.BFMatcher(cv2.NORM_HAMMING)

# Match descriptors.
matches = bf.knnMatch(des_ref,des_que,k=2)

# Apply ratio test
good = []
good2 = []
for m,n in matches:
    if m.distance < 0.8*n.distance:
        good.append([m])
        good2.append(m)
img = cv2.drawMatchesKnn(ref_img2,kp_ref,que_img2,kp_que,good,None,flags=cv2.DrawMatchesFla

# draw matches
plt.title("ratio = 0.8")
plt.imshow(img),plt.show()

# different ratio test values
good = []
good2 = []
for m,n in matches:
    if m.distance < 0.5*n.distance:
        good.append([m])
        good2.append(m)
img = cv2.drawMatchesKnn(ref_img2,kp_ref,que_img2,kp_que,good,None,flags=cv2.DrawMatchesFla

# draw matches
plt.title("ratio = 0.5")
plt.imshow(img),plt.show()

good = []
good2 = []
for m,n in matches:
    if m.distance < 1.5*n.distance:
        good.append([m])
        good2.append(m)
```

```
img = cv2.drawMatchesKnn(ref_img2,kp_ref,que_img2,kp_que,good,None,flags=cv2.DrawMatchesFla

# draw matches
plt.title("ratio = 1.5")
plt.imshow(img),plt.show()

# different orb parameters
orb2 = cv2.ORB_create(nfeatures=100)

kp_ref = orb2.detect(ref,None)
kp_que = orb2.detect(que,None)

kp_ref,des_ref = orb2.compute(ref,kp_ref)
kp_que,des_que = orb2.compute(que,kp_que)

# draw keypoints
ref_img2 = cv2.drawKeypoints(ref,kp_ref,None,color=(0,255,0),flags=0)
que_img2 = cv2.drawKeypoints(que,kp_que,None,color=(0,255,0),flags=0)

# create BFMatcher object
bf = cv2.BFMatcher(cv2.NORM_HAMMING)

# Match descriptors.
matches = bf.knnMatch(des_ref,des_que,k=2)

# Apply ratio test
good = []
good2 = []
for m,n in matches:
    if m.distance < 0.8*n.distance:
        good.append([m])
        good2.append(m)
img = cv2.drawMatchesKnn(ref_img2,kp_ref,que_img2,kp_que,good,None,flags=cv2.DrawMatchesFla

# draw matches
plt.title("ratio = 0.8, nfeatures = 100")
plt.imshow(img),plt.show()

# different orb parameters
orb3 = cv2.ORB_create(scaleFactor=2)

kp_ref = orb3.detect(ref,None)
kp_que = orb3.detect(que,None)

kp_ref,des_ref = orb3.compute(ref,kp_ref)
kp_que,des_que = orb3.compute(que,kp_que)

# draw keypoints
ref_img2 = cv2.drawKeypoints(ref,kp_ref,None,color=(0,255,0),flags=0)
que_img2 = cv2.drawKeypoints(que,kp_que,None,color=(0,255,0),flags=0)

# create BFMatcher object
bf = cv2.BFMatcher(cv2.NORM_HAMMING)

# Match descriptors.
matches = bf.knnMatch(des_ref,des_que,k=2)

# Apply ratio test
good = []
good2 = []
for m,n in matches:
```

```
    if m.distance < 0.8*n.distance:
        good.append([m])
        good2.append(m)
img = cv2.drawMatchesKnn(ref_img2,kp_ref,que_img2,kp_que,good,None,flags=cv2.DrawMatchesFla

# draw matches
plt.title("ratio = 0.8, scaleFactor = 2")
plt.imshow(img),plt.show()

# different orb parameters
orb4 = cv2.ORB_create(edgeThreshold = 100)

kp_ref = orb4.detect(ref,None)
kp_que = orb4.detect(que,None)

kp_ref,des_ref = orb4.compute(ref,kp_ref)
kp_que,des_que = orb4.compute(que,kp_que)

# draw keypoints
ref_img2 = cv2.drawKeypoints(ref,kp_ref,None,color=(0,255,0),flags=0)
que_img2 = cv2.drawKeypoints(que,kp_que,None,color=(0,255,0),flags=0)

# create BFMatcher object
bf = cv2.BFMatcher(cv2.NORM_HAMMING)

# Match descriptors.
matches = bf.knnMatch(des_ref,des_que,k=2)

# Apply ratio test
good = []
good2 = []
for m,n in matches:
    if m.distance < 0.8*n.distance:
        good.append([m])
        good2.append(m)
img = cv2.drawMatchesKnn(ref_img2,kp_ref,que_img2,kp_que,good,None,flags=cv2.DrawMatchesFla

# draw matches
plt.title("ratio = 0.8, edgeThreshold = 100")
plt.imshow(img),plt.show()

# different BFMatcher parameters
# find the keypoints and descriptors with ORB
kp_ref = orb.detect(ref,None)
kp_que = orb.detect(que,None)

kp_ref,des_ref = orb.compute(ref,kp_ref)
kp_que,des_que = orb.compute(que,kp_que)

# draw keypoints
ref_img2 = cv2.drawKeypoints(ref,kp_ref,None,color=(0,255,0),flags=0)
que_img2 = cv2.drawKeypoints(que,kp_que,None,color=(0,255,0),flags=0)

# create BFMatcher object
bf = cv2.BFMatcher(cv2.NORM_L1)

# Match descriptors.
matches = bf.knnMatch(des_ref,des_que,k=2)

# Apply ratio test
good = []
```

```
good2 = []
for m,n in matches:
    if m.distance < 0.8*n.distance:
        good.append([m])
        good2.append(m)
img = cv2.drawMatchesKnn(ref_img2,kp_ref,que_img2,kp_que,good,None,flags=cv2.DrawMatchesFla

# draw matches
plt.title("ratio = 0.8, BFMatcher = NORM_L1 (manhattan)")
plt.imshow(img),plt.show()

# different BFMatcher parameter
bf = cv2.BFMatcher(cv2.NORM_L2)

# Match descriptors.
matches = bf.knnMatch(des_ref,des_que,k=2)

# Apply ratio test
good = []
good2 = []
for m,n in matches:
    if m.distance < 0.8*n.distance:
        good.append([m])
        good2.append(m)
img = cv2.drawMatchesKnn(ref_img2,kp_ref,que_img2,kp_que,good,None,flags=cv2.DrawMatchesFla

# draw matches
plt.title("ratio = 0.8, BFMatcher = NORM_L2 (euclidean)")
plt.imshow(img),plt.show()
```



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

1. Changing ratio values

By changing the value of the ratio used in the ratio test, the number of matches that are classified as 'good' can be influenced. When the ratio is lowered, only matches that have a greater degree of similarity are drawn, and conversely, when the ratio is increased, some matches with less similarity are drawn. Dependent on the ratio value chosen, there can be certain drawbacks. For example, when a ratio value of 0.5 is used, many matches are dropped, due to the difference in lighting condition. This can negatively impact the confidence of a match in

images. When a ratio value of 1.5 is used however, many bad matches are allowed, such as some parts of the query image which do not contain the book cover being matched to the top of the reference image. Therefore, it is important to choose a ratio value that allows for a reasonable number of matches, without compromising the confidence or accuracy of the feature matching.

2. Changing parameters in cv2.ORB_create()

The default parameters of the ORB_create function are nfeatures=500, scaleFactor = 1.2, edgeThreshold = 31. By adjusting these parameters, it is possible to modify the matches drawn. For example, by modifying nfeatures, the maximum number of features retained is changed, and by lowering it to 100, less matches are formed. The scale factor determines the decimation ratio of the image pyramid, and by increasing it to 2, each level of the pyramid will have 4 times less pixels, thus the feature matching scores will be affected. The edge threshold determines the edges of the image where features are not detected. Increasing this value to 100 results in a border of 100 pixels where features are not being matched.

3. Changing distance functions in cv2.BFMatcher()

Instead of using Hamming distance, the Manhattan distance (NORM_L1) or the Euclidean distance (NORM_L2) can be used instead. These will result in slightly different matches being found, but are both relatively similar.

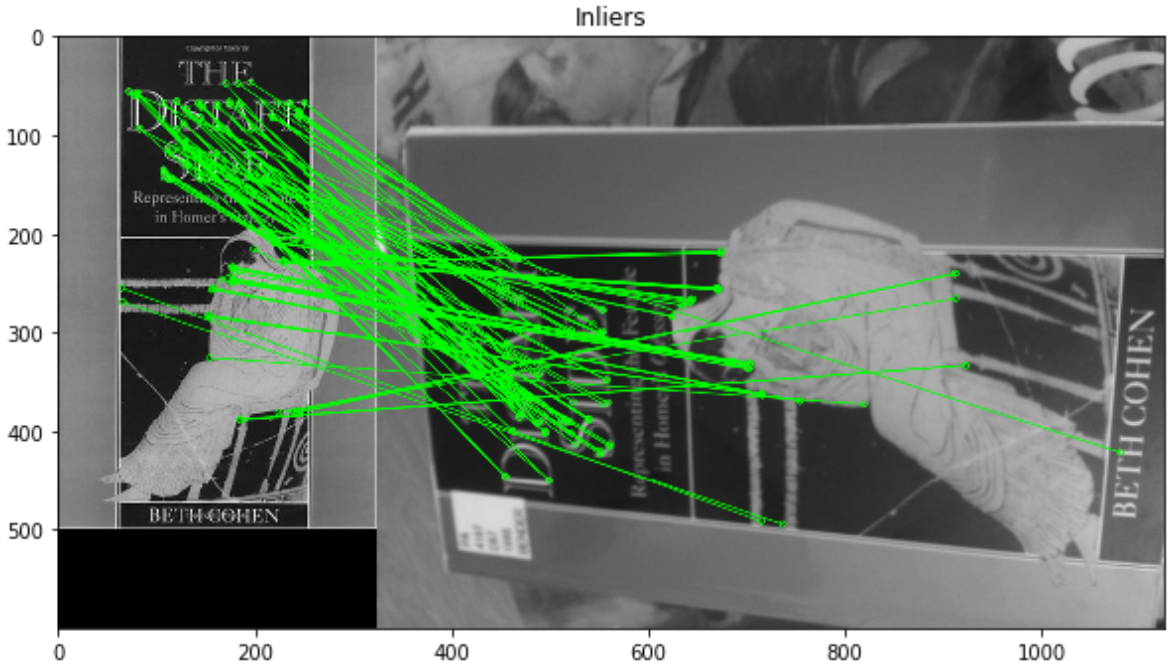
3. Estimate a homography transformation based on the matches, using `cv2.findHomography()`. Display the transformed outline of the first reference book cover image on the query image, to see how well they match.
    - We provide a function `draw_outline()` to help with the display, but you may need to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

In [4]:

```
# return to ratio = 0.8, hamming distance, default ORB
kp_ref = orb.detect(ref,None)
kp_que = orb.detect(que,None)

kp_ref,des_ref = orb.compute(ref,kp_ref)
kp_que,des_que = orb.compute(que,kp_que)

# draw keypoints
ref_img2 = cv2.drawKeypoints(ref,kp_ref,None,color=(0,255,0),flags=0)
que_img2 = cv2.drawKeypoints(que,kp_que,None,color=(0,255,0),flags=0)

bf = cv2.BFMatcher(cv2.NORM_HAMMING)

# Match descriptors.
matches = bf.knnMatch(des_ref,des_que,k=2)

# Apply ratio test
good = []
good2 = []
for m,n in matches:
    if m.distance < 0.8*n.distance:
        good.append([m])
        good2.append(m)
img = cv2.drawMatchesKnn(ref_img2,kp_ref,que_img2,kp_que,good,None,flags=cv2.DrawMatchesFla
```

In [5]:

```
# Your code to display book location here
que_pts = np.float32([kp_ref[m.queryIdx].pt for m in good2 ]).reshape(-1,1,2)
ref_pts = np.float32([kp_que[m.trainIdx].pt for m in good2 ]).reshape(-1,1,2)

# using regular method (cv2.findHomography)
matrix, mask = cv2.findHomography(que_pts, ref_pts, 0)

# Draw in and outliners
# draw inliers
plt.title("Inliers")
draw_inliers(ref, que, kp_ref, kp_que, good2, mask)
# draw outline
plt.title("Outline")
draw_outline(ref, que, matrix)
```



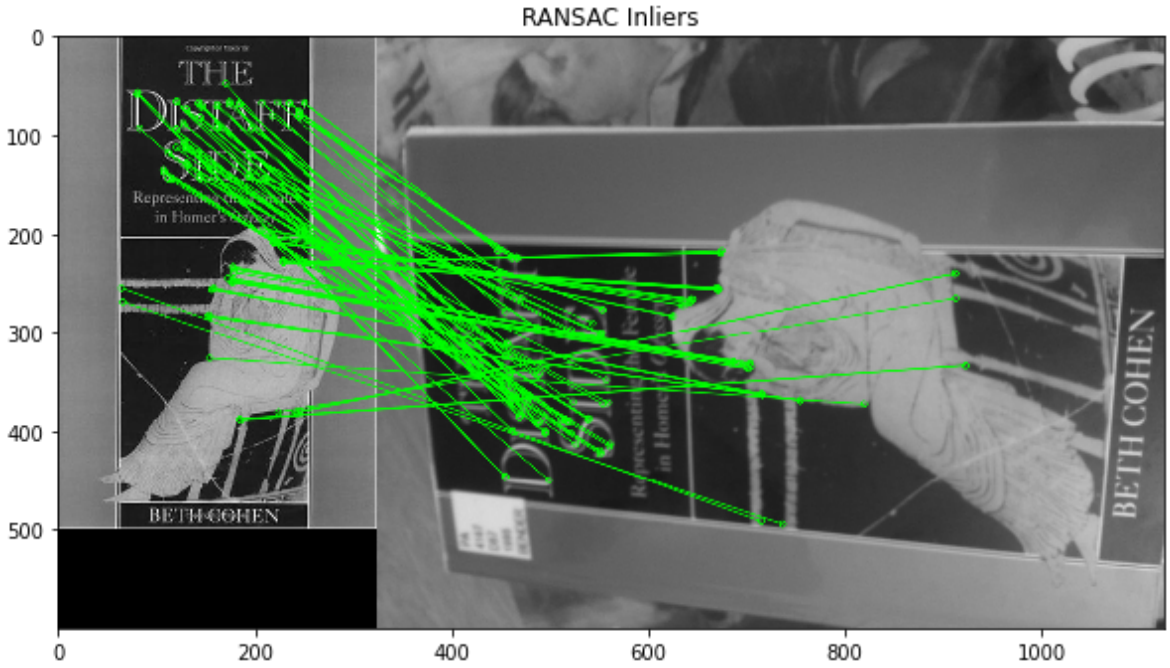
Your explanation of results here

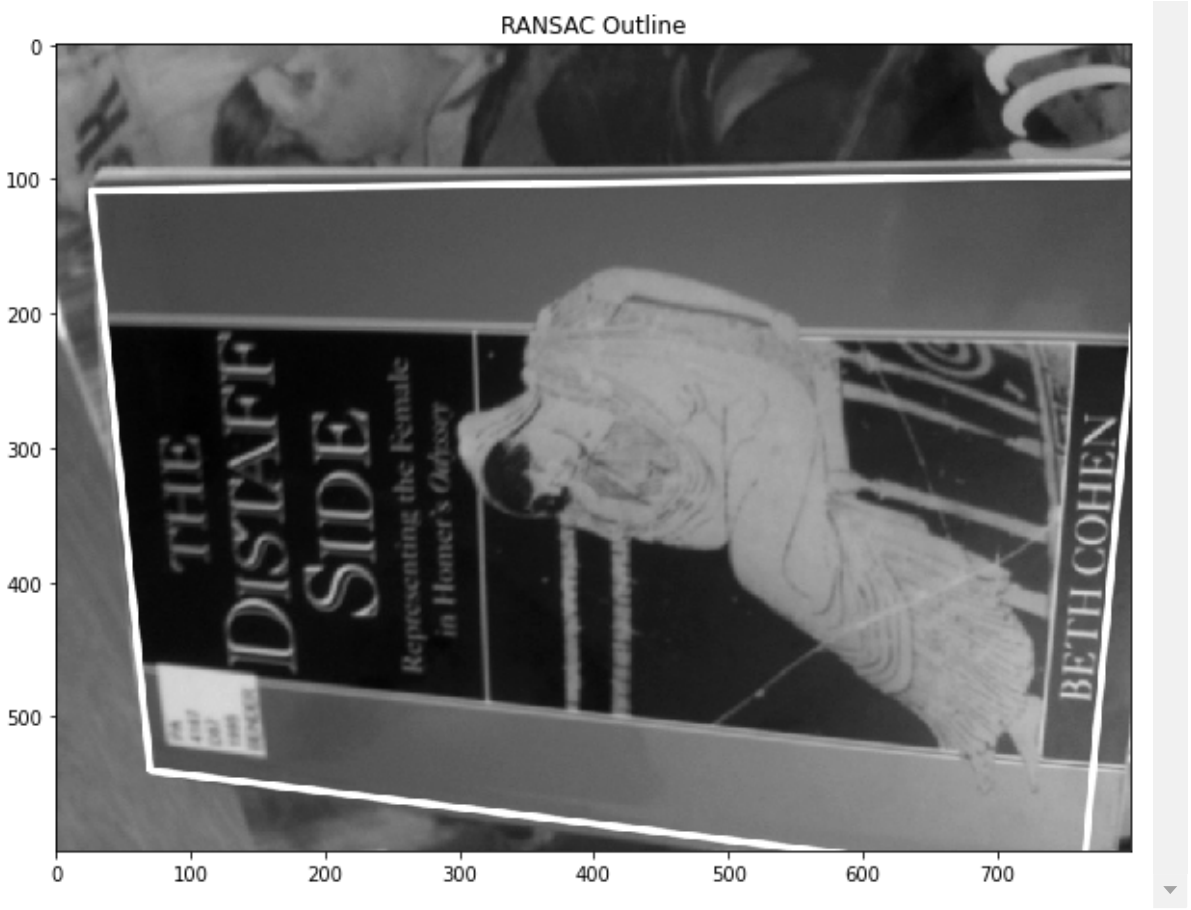
Using cv2.findHomography, the outline is fitted around the inliers, but does not account for possible outliers which may be in the image but are not found as a match.

Try the RANSAC option to compute homography. Change the RANSAC parameters, and explain your results. Print and analyze the inlier numbers. Hint: use cv2.RANSAC with cv2.findHomography.

In [6]:

```
# Your code to display book location after RANSAC here
que_pts = np.float32([kp_ref[m.queryIdx].pt for m in good2 ]).reshape(-1,1,2)
ref_pts = np.float32([kp_que[m.trainIdx].pt for m in good2 ]).reshape(-1,1,2)

matrix, mask = cv2.findHomography(que_pts, ref_pts, cv2.RANSAC, 5.0)

# draw outline and inliers
plt.title("RANSAC Inliers")
draw_inliers(ref, que, kp_ref, kp_que, good2, mask)
# draw outline
plt.title("RANSAC Outline")
draw_outline(ref, que, matrix)
```



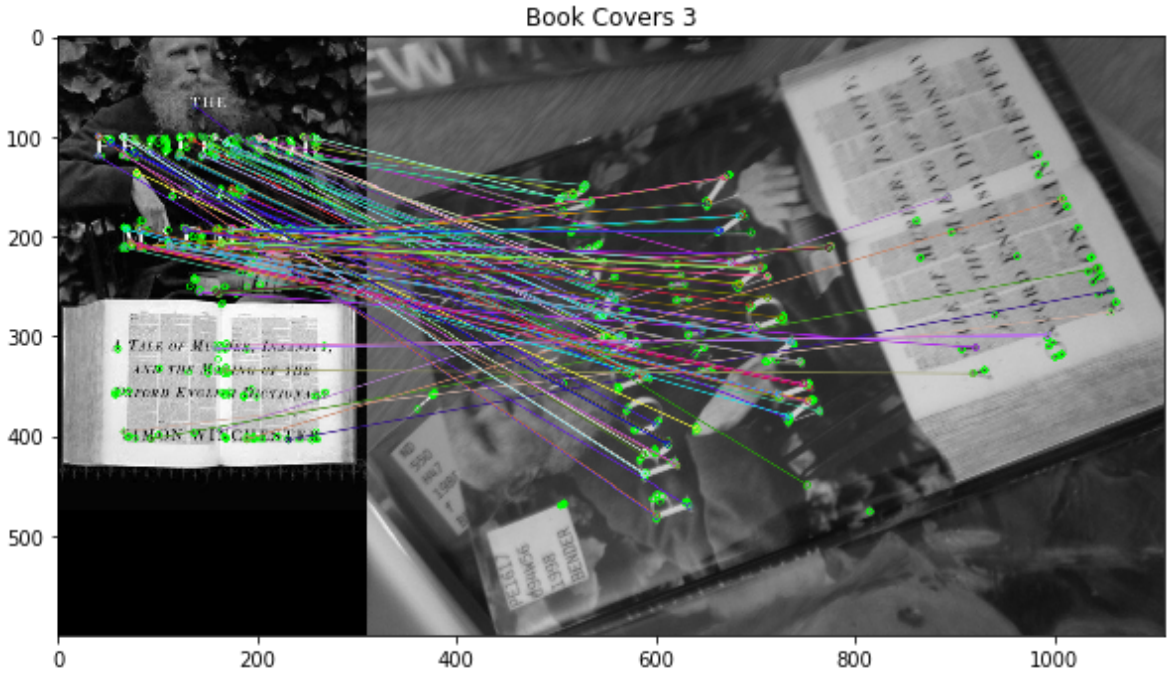


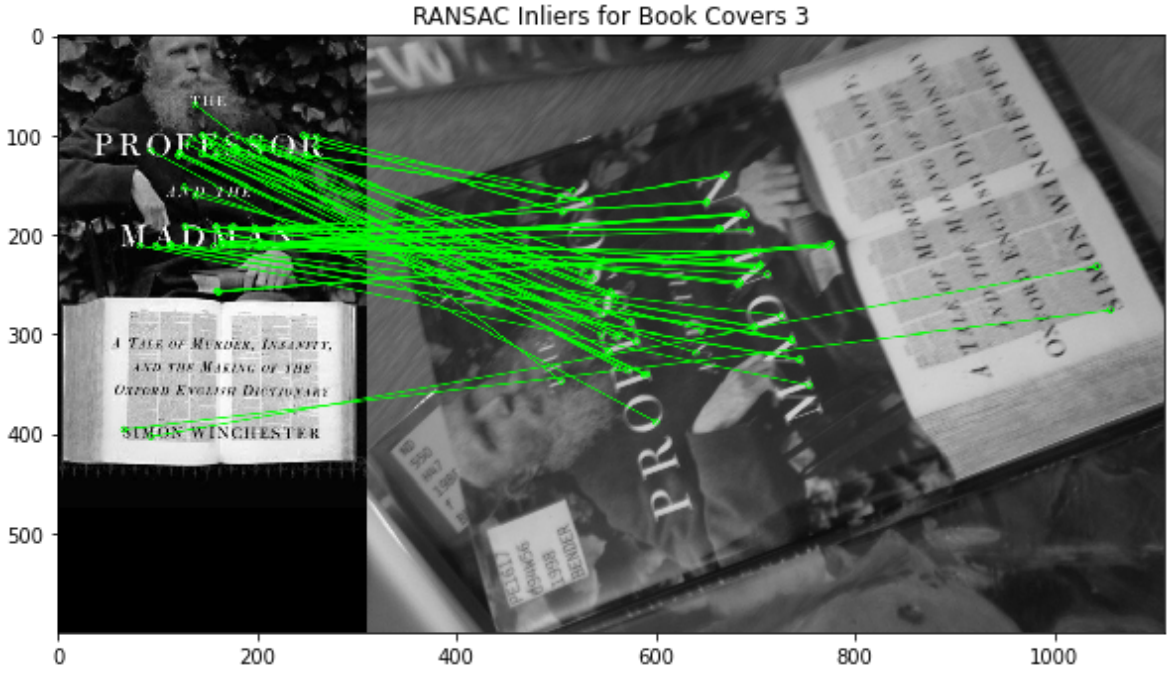
Your explanation of what you have tried, and results here Using the Random Sample Consensus (RANSAC), the outline of the image in the query is estimated based on both the inliers and the outliers, iteratively repeating the process of drawing the outline until the consensus is reached that there are enoguh inliers in the frame.

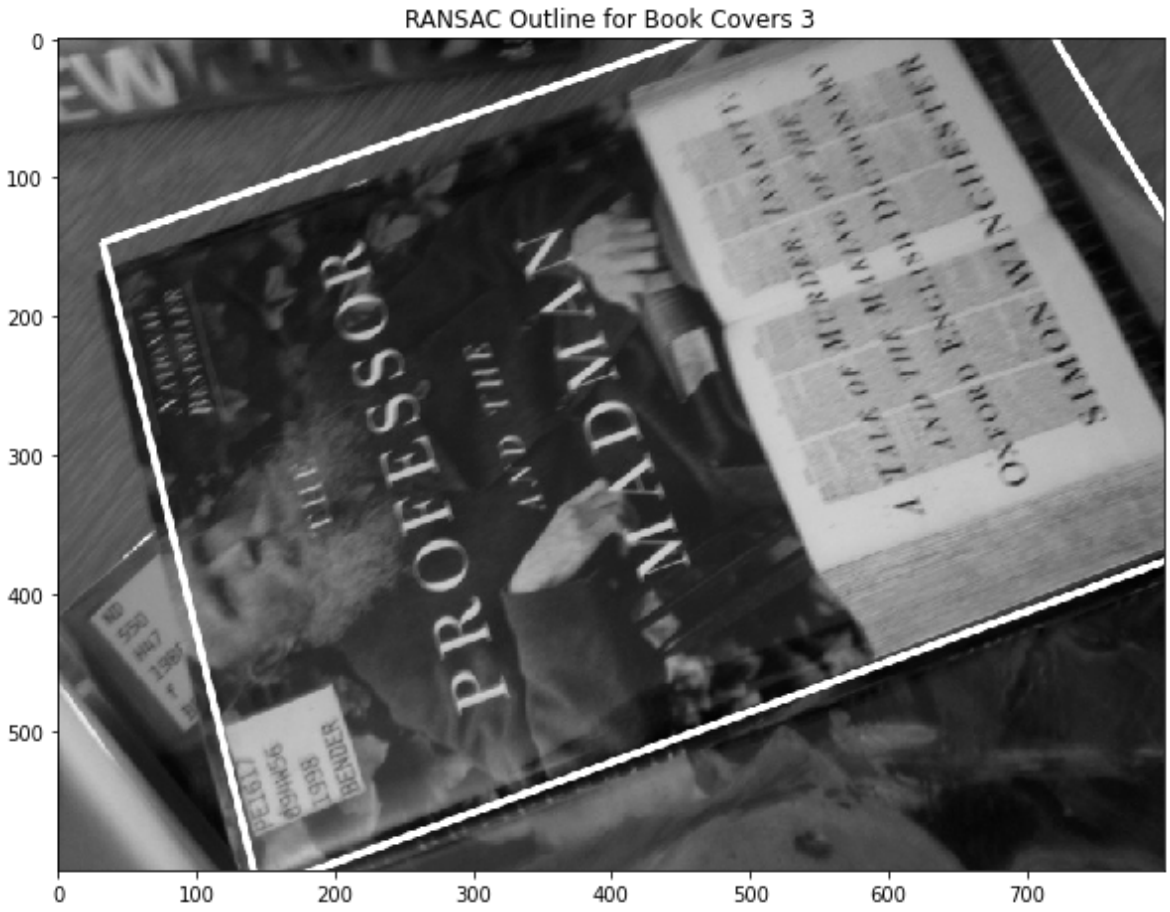
6. Finally, try matching several different image pairs from the data provided, including at least one success and one failure case. For the failure case, test and explain what step in the feature matching has failed, and try to improve it. Display and discuss your findings.
   A. Hint 1: In general, 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 [7]:

```
# Your results for other image pairs here
ref = cv2.imread('book_covers/Reference/003.jpg',0)
que = cv2.imread('book_covers/Query/003.jpg',0)

# compute detector and descriptor
orb = cv2.ORB_create()

kp_ref = orb.detect(ref,None)
kp_que = orb.detect(que,None)

kp_ref,des_ref = orb.compute(ref,kp_ref)
kp_que,des_que = orb.compute(que,kp_que)

# draw keypoints
ref_img2 = cv2.drawKeypoints(ref,kp_ref,None,color=(0,255,0),flags=0)
que_img2 = cv2.drawKeypoints(que,kp_que,None,color=(0,255,0),flags=0)

bf = cv2.BFMatcher(cv2.NORM_HAMMING)

# Match descriptors.
matches = bf.knnMatch(des_ref,des_que,k=2)

# Apply ratio test
good = []
good2 = []
for m,n in matches:
    if m.distance < 0.8*n.distance:
        good.append([m])
        good2.append(m)
img = cv2.drawMatchesKnn(ref_img2,kp_ref,que_img2,kp_que,good,None,flags=cv2.DrawMatchesFla

plt.title("Book Covers 3")
plt.imshow(img), plt.show()

# Your code to display book location after RANSAC here
que_pts = np.float32([kp_ref[m.queryIdx].pt for m in good2 ]).reshape(-1,1,2)
ref_pts = np.float32([kp_que[m.trainIdx].pt for m in good2 ]).reshape(-1,1,2)

matrix, mask = cv2.findHomography(que_pts, ref_pts, cv2.RANSAC, 5.0)

# draw outline and inliers
plt.title("RANSAC Inliers for Book Covers 3")
draw_inliers(ref, que, kp_ref, kp_que, good2, mask)
# draw outline
plt.title("RANSAC Outline for Book Covers 3")
draw_outline(ref, que, matrix)
```

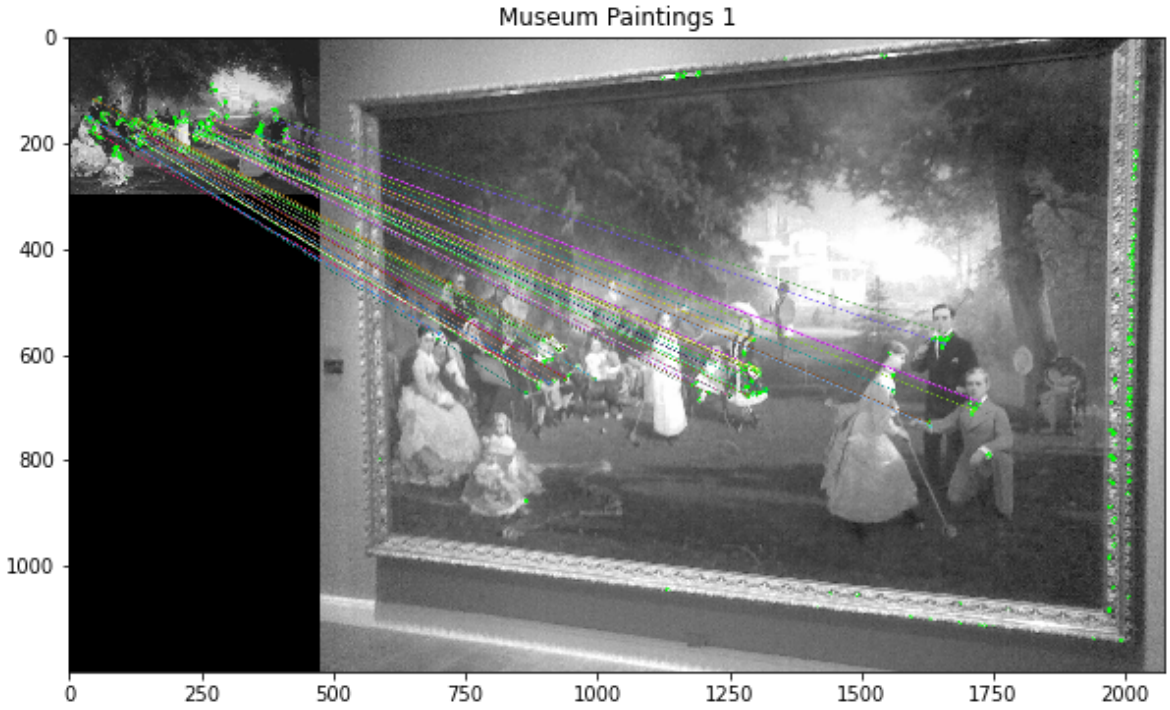

Book Covers 3
0
100
200
300
400
500
0
200
400
600
800
1000

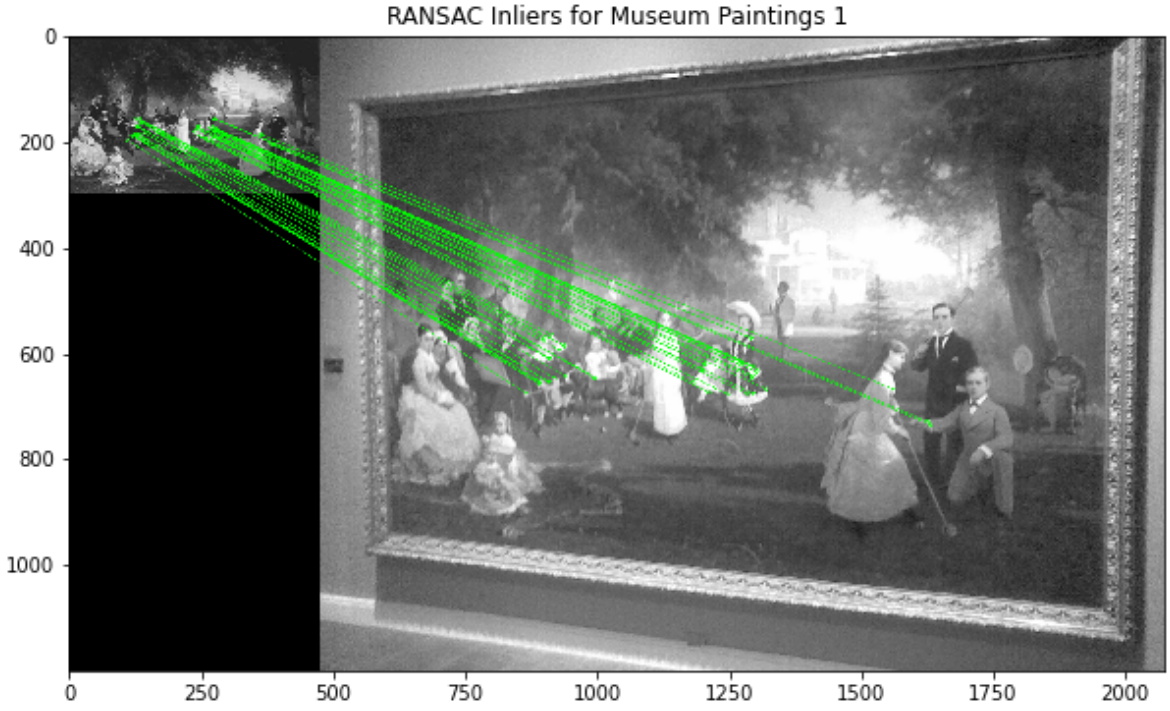

RANSAC Inliers for Book Covers 3
0
100
200
300
400
500
0
200
400
600
800
1000

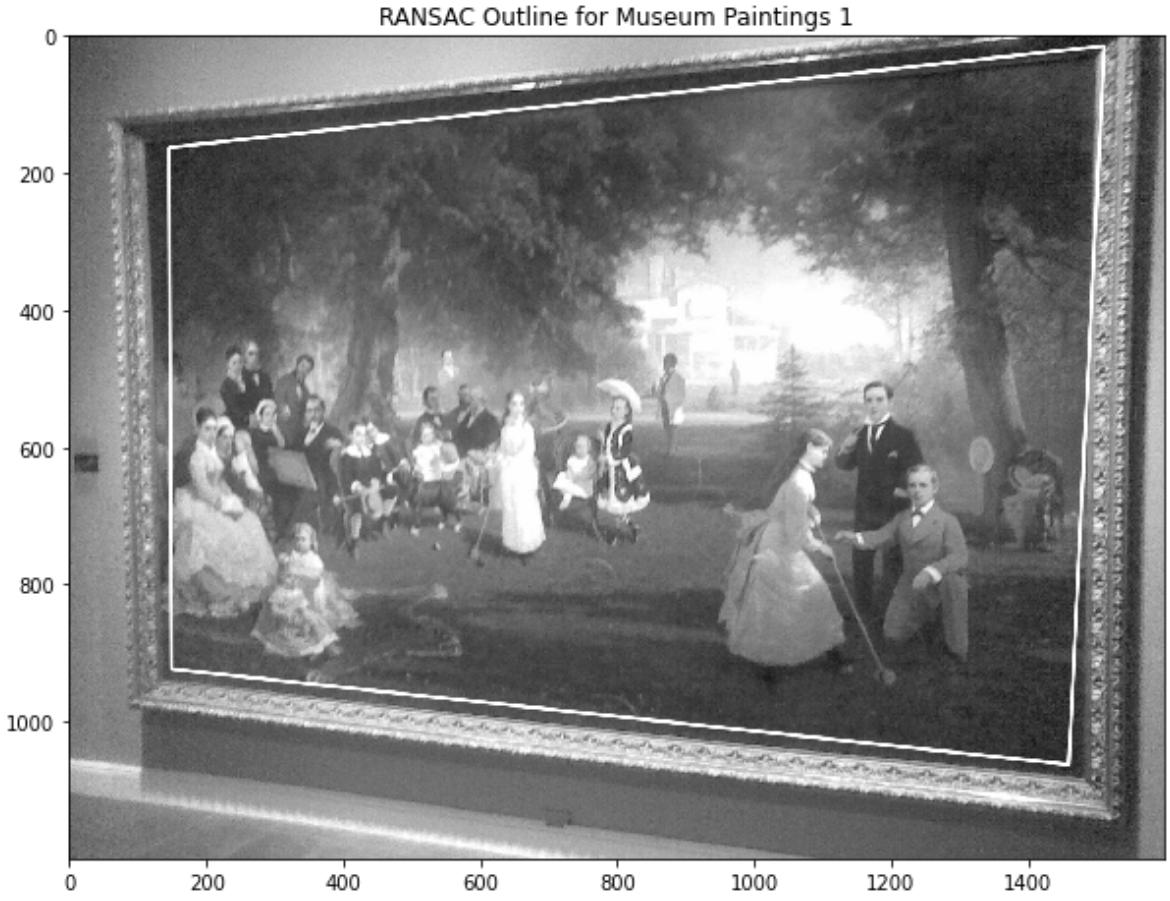

RANSAC Outline for Book Covers 3
0
100
200
300
400
500
0
100
200
300
400
500
600
700
PROFESSOR
MADMAN
SIMON WINCHESTER

In [8]:

```
# Your results for other image pairs here
ref = cv2.imread('museum_paintings/Reference/001.jpg',0)
que = cv2.imread('museum_paintings/Query/001.jpg',0)

# compute detector and descriptor
orb = cv2.ORB_create()

kp_ref = orb.detect(ref,None)
kp_que = orb.detect(que,None)

kp_ref,des_ref = orb.compute(ref,kp_ref)
kp_que,des_que = orb.compute(que,kp_que)

# draw keypoints
ref_img2 = cv2.drawKeypoints(ref,kp_ref,None,color=(0,255,0),flags=0)
que_img2 = cv2.drawKeypoints(que,kp_que,None,color=(0,255,0),flags=0)

bf = cv2.BFMatcher(cv2.NORM_HAMMING)

# Match descriptors.
matches = bf.knnMatch(des_ref,des_que,k=2)

# Apply ratio test
good = []
good2 = []
for m,n in matches:
    if m.distance < 0.8*n.distance:
        good.append([m])
        good2.append(m)
img = cv2.drawMatchesKnn(ref_img2,kp_ref,que_img2,kp_que,good,None,flags=cv2.DrawMatchesFla

plt.title("Museum Paintings 1")
plt.imshow(img), plt.show()

# Your code to display book location after RANSAC here
que_pts = np.float32([kp_ref[m.queryIdx].pt for m in good2 ]).reshape(-1,1,2)
ref_pts = np.float32([kp_que[m.trainIdx].pt for m in good2 ]).reshape(-1,1,2)

matrix, mask = cv2.findHomography(que_pts, ref_pts, cv2.RANSAC, 5.0)

# draw outline and inliers
plt.title("RANSAC Inliers for Museum Paintings 1")
draw_inliers(ref, que, kp_ref, kp_que, good2, mask)
# draw outline
plt.title("RANSAC Outline for Museum Paintings 1")
draw_outline(ref, que, matrix)
```

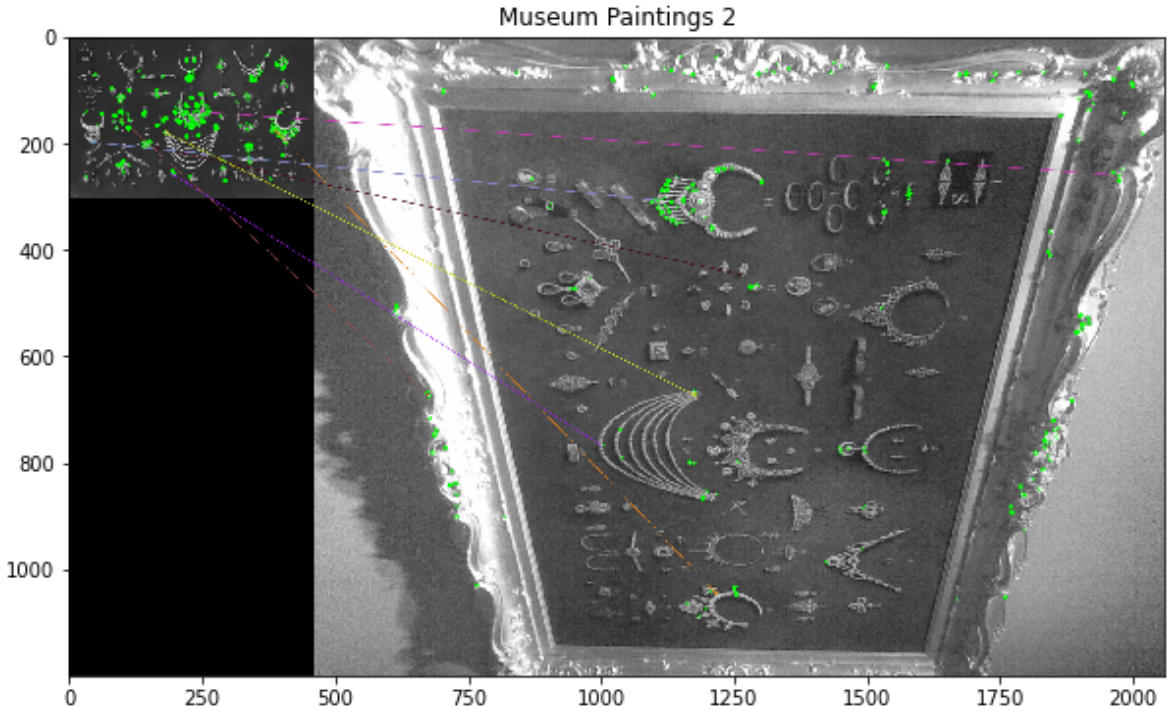

Museum Paintings 1
0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750
2000

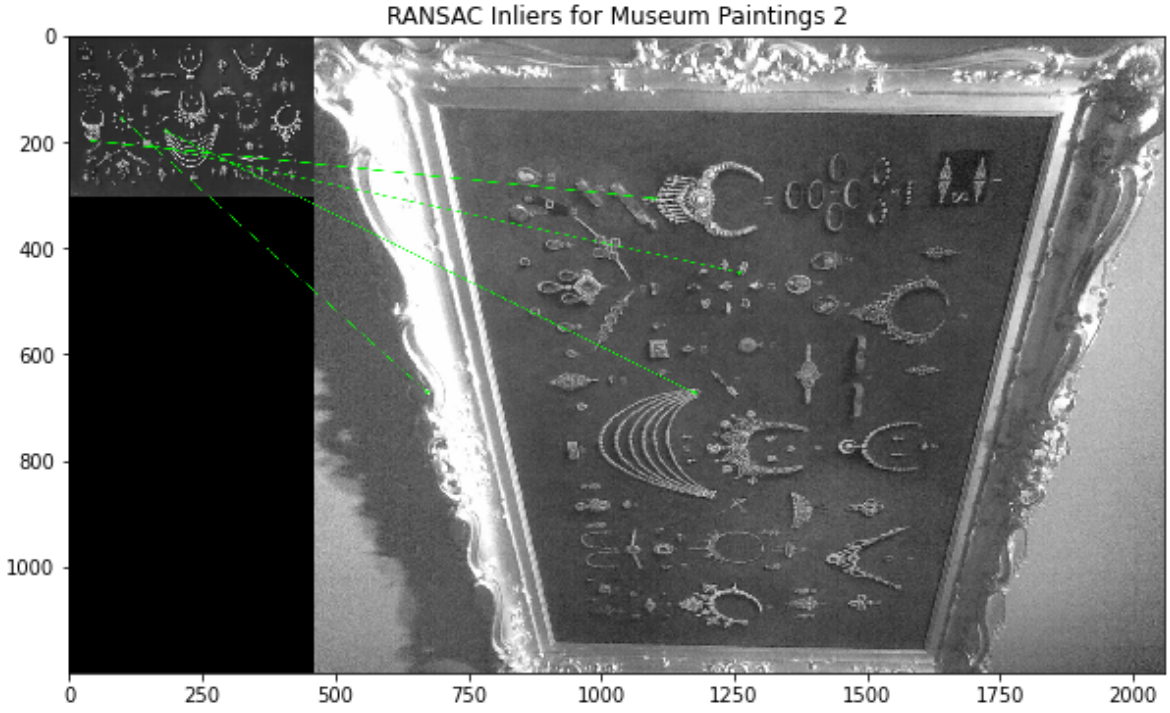

RANSAC Inliers for Museum Paintings 1
0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750
2000

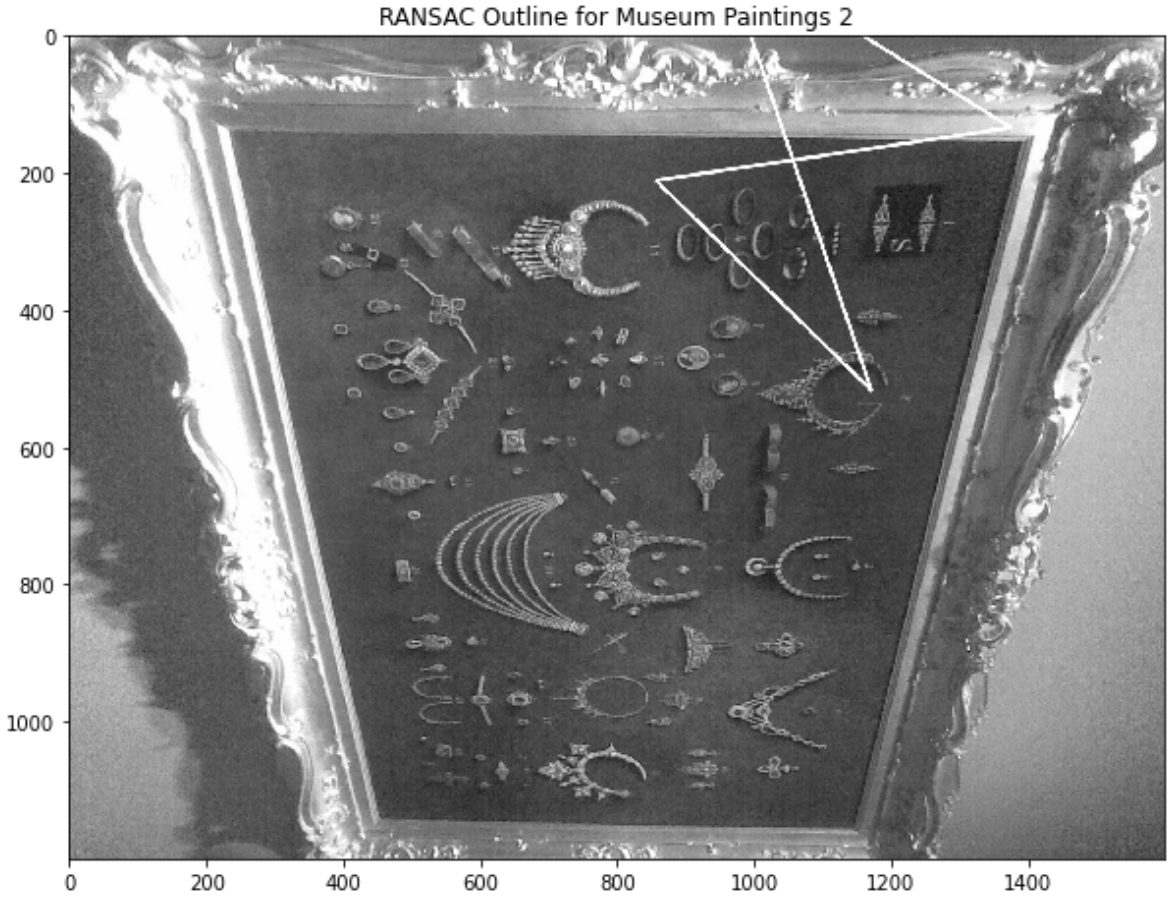

RANSAC Outline for Museum Paintings 1
0
200
400
600
800
1000
0
200
400
600
800
1000
1200
1400

In [9]:

```
# Your results for other image pairs here
ref = cv2.imread('museum_paintings/Reference/002.jpg',0)
que = cv2.imread('museum_paintings/Query/002.jpg',0)

# compute detector and descriptor
orb = cv2.ORB_create()

kp_ref = orb.detect(ref,None)
kp_que = orb.detect(que,None)

kp_ref,des_ref = orb.compute(ref,kp_ref)
kp_que,des_que = orb.compute(que,kp_que)

# draw keypoints
ref_img2 = cv2.drawKeypoints(ref,kp_ref,None,color=(0,255,0),flags=0)
que_img2 = cv2.drawKeypoints(que,kp_que,None,color=(0,255,0),flags=0)

bf = cv2.BFMatcher(cv2.NORM_HAMMING)

# Match descriptors.
matches = bf.knnMatch(des_ref,des_que,k=2)

# Apply ratio test
good = []
good2 = []
for m,n in matches:
    if m.distance < 0.8*n.distance:
        good.append([m])
        good2.append(m)
img = cv2.drawMatchesKnn(ref_img2,kp_ref,que_img2,kp_que,good,None,flags=cv2.DrawMatchesFla

plt.title("Museum Paintings 2")
plt.imshow(img), plt.show()

# Your code to display book location after RANSAC here
que_pts = np.float32([kp_ref[m.queryIdx].pt for m in good2 ]).reshape(-1,1,2)
ref_pts = np.float32([kp_que[m.trainIdx].pt for m in good2 ]).reshape(-1,1,2)

matrix, mask = cv2.findHomography(que_pts, ref_pts, cv2.RANSAC, 5.0)

# draw outline and inliers
plt.title("RANSAC Inliers for Museum Paintings 2")
draw_inliers(ref, que, kp_ref, kp_que, good2, mask)
# draw outline
plt.title("RANSAC Outline for Museum Paintings 2")
draw_outline(ref, que, matrix)
```

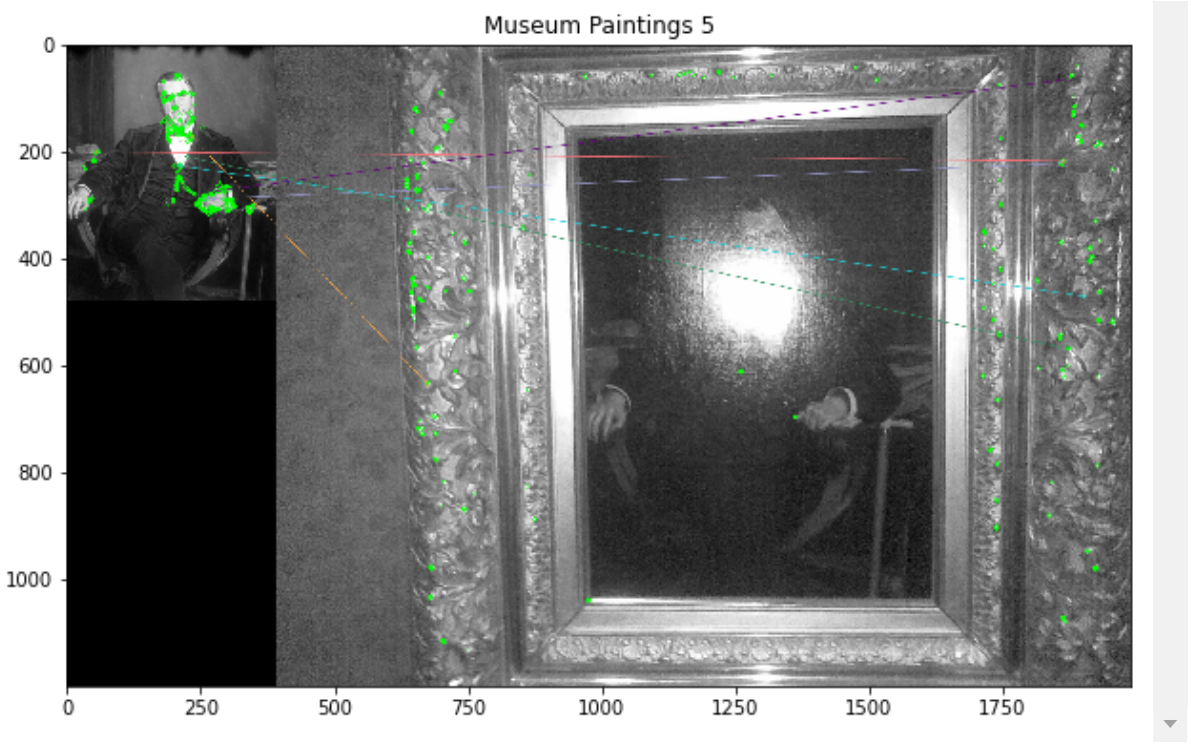

Museum Paintings 2
0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750
2000

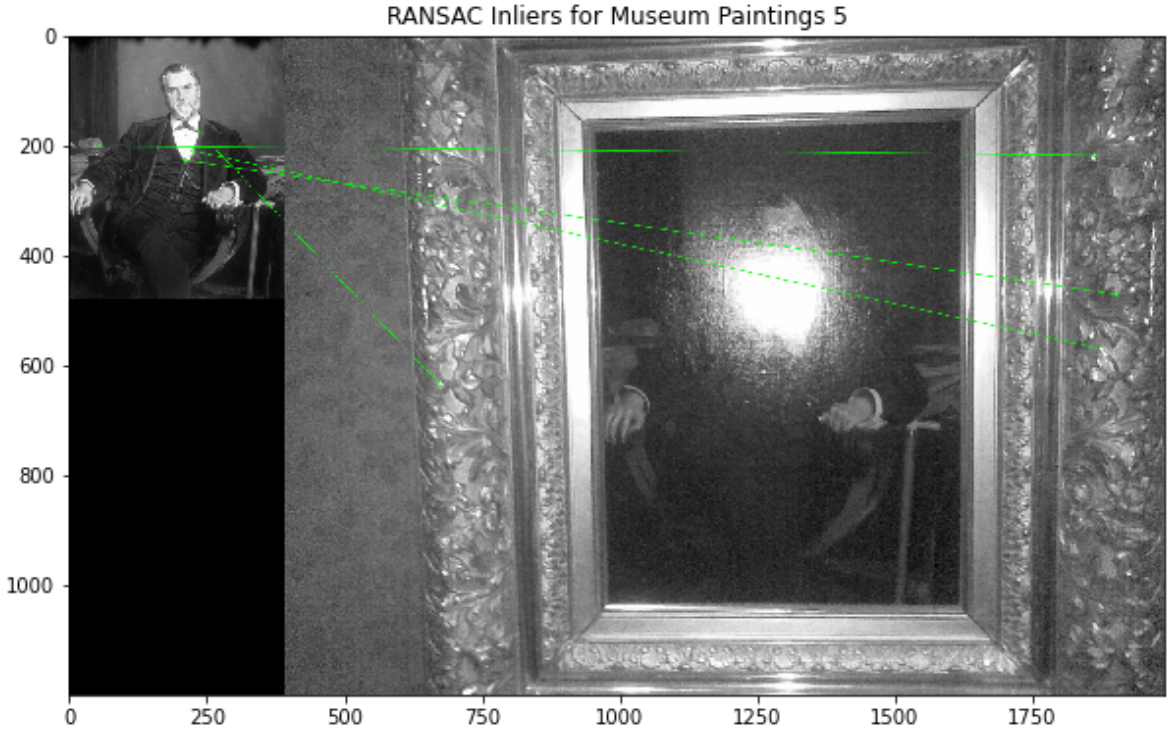

RANSAC Inliers for Museum Paintings 2
0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750
2000

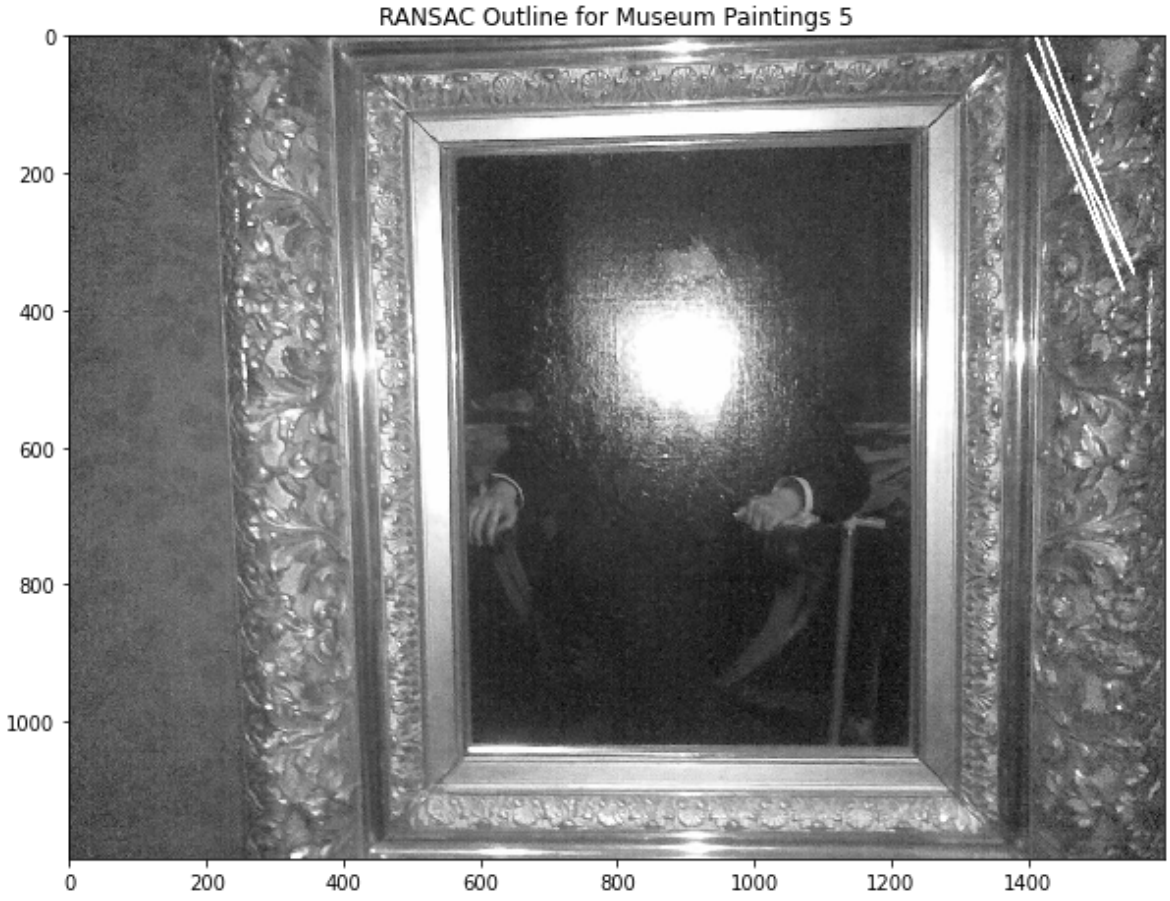

RANSAC Outline for Museum Paintings 2
0
200
400
600
800
1000
0
200
400
600
800
1000
1200
1400

In [10]:

```
# Your results for other image pairs here
ref = cv2.imread('museum_paintings/Reference/005.jpg',0)
que = cv2.imread('museum_paintings/Query/005.jpg',0)

# compute detector and descriptor
orb = cv2.ORB_create()

kp_ref = orb.detect(ref,None)
kp_que = orb.detect(que,None)

kp_ref,des_ref = orb.compute(ref,kp_ref)
kp_que,des_que = orb.compute(que,kp_que)

# draw keypoints
ref_img2 = cv2.drawKeypoints(ref,kp_ref,None,color=(0,255,0),flags=0)
que_img2 = cv2.drawKeypoints(que,kp_que,None,color=(0,255,0),flags=0)

bf = cv2.BFMatcher(cv2.NORM_HAMMING)

# Match descriptors.
matches = bf.knnMatch(des_ref,des_que,k=2)

# Apply ratio test
good = []
good2 = []
for m,n in matches:
    if m.distance < 0.8*n.distance:
        good.append([m])
        good2.append(m)
img = cv2.drawMatchesKnn(ref_img2,kp_ref,que_img2,kp_que,good,None,flags=cv2.DrawMatchesFla

plt.title("Museum Paintings 5")
plt.imshow(img), plt.show()

# Your code to display book location after RANSAC here
que_pts = np.float32([kp_ref[m.queryIdx].pt for m in good2 ]).reshape(-1,1,2)
ref_pts = np.float32([kp_que[m.trainIdx].pt for m in good2 ]).reshape(-1,1,2)

matrix, mask = cv2.findHomography(que_pts, ref_pts, cv2.RANSAC, 5.0)

# draw outline and inliers
plt.title("RANSAC Inliers for Museum Paintings 5")
draw_inliers(ref, que, kp_ref, kp_que, good2, mask)
# draw outline
plt.title("RANSAC Outline for Museum Paintings 5")
draw_outline(ref, que, matrix)
```


Museum Paintings 5
0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750


RANSAC Inliers for Museum Paintings 5
0
200
400
600
800
1000
0
250
500
750
1000
1250
1500
1750


RANSAC Outline for Museum Paintings 5
0
200
400
600
800
1000
0
200
400
600
800
1000
1200
1400

In [11]:

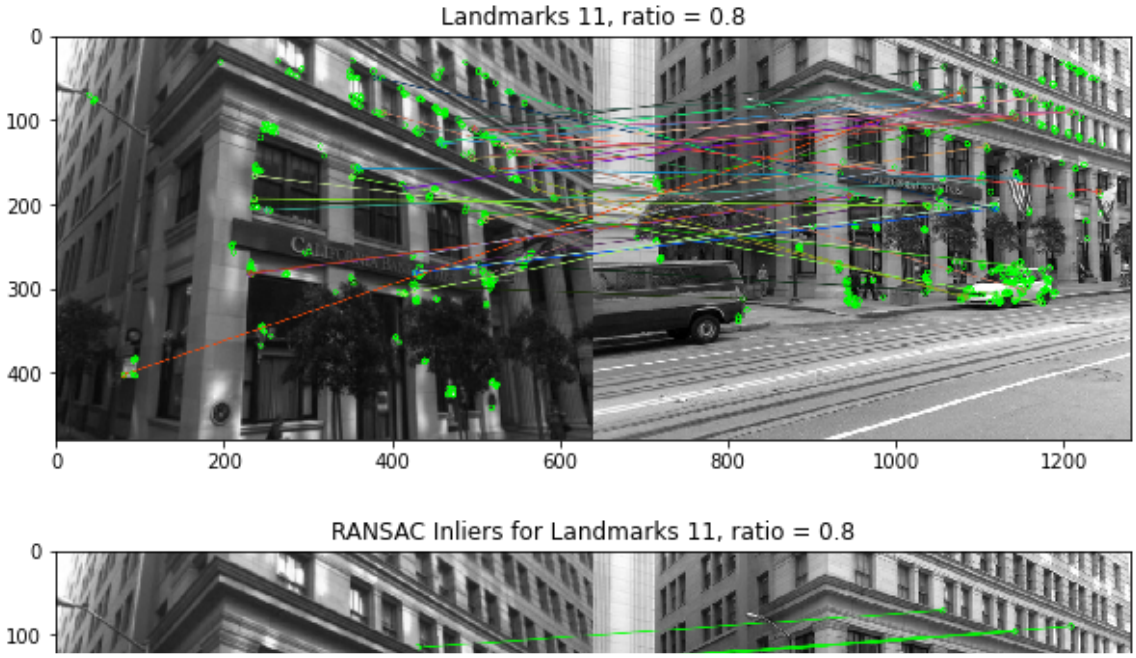
```
# Your results for other image pairs here
ref = cv2.imread('landmarks/Reference/001.jpg',0)
que = cv2.imread('landmarks/Query/001.jpg',0)

# compute detector and descriptor
orb = cv2.ORB_create()

kp_ref = orb.detect(ref,None)
kp_que = orb.detect(que,None)

kp_ref,des_ref = orb.compute(ref,kp_ref)
kp_que,des_que = orb.compute(que,kp_que)

# draw keypoints
ref_img2 = cv2.drawKeypoints(ref,kp_ref,None,color=(0,255,0),flags=0)
que_img2 = cv2.drawKeypoints(que,kp_que,None,color=(0,255,0),flags=0)

bf = cv2.BFMatcher(cv2.NORM_HAMMING)

# Match descriptors.
matches = bf.knnMatch(des_ref,des_que,k=2)

# Apply ratio test
good = []
good2 = []
for m,n in matches:
    if m.distance < 0.87*n.distance:
        good.append([m])
        good2.append(m)
img = cv2.drawMatchesKnn(ref_img2,kp_ref,que_img2,kp_que,good,None,flags=cv2.DrawMatchesFla

plt.title("Landmarks 1")
plt.imshow(img), plt.show()

# Your code to display book location after RANSAC here
que_pts = np.float32([kp_ref[m.queryIdx].pt for m in good2 ]).reshape(-1,1,2)
ref_pts = np.float32([kp_que[m.trainIdx].pt for m in good2 ]).reshape(-1,1,2)

matrix, mask = cv2.findHomography(que_pts, ref_pts, cv2.RANSAC, 5.0)

# draw outline and inliers
plt.title("RANSAC Inliers for Landmarks 1")
draw_inliers(ref, que, kp_ref, kp_que, good2, mask)
# draw outline
plt.title("RANSAC Outline for Landmarks 1")
draw_outline(ref, que, matrix)
```


Landmarks 1
RANSAC Inliers for Landmarks 1
0
100
0
200
100

In [12]:

```
# Your results for other image pairs here
ref = cv2.imread('landmarks/Reference/011.jpg',0)
que = cv2.imread('landmarks/Query/011.jpg',0)

# compute detector and descriptor
orb = cv2.ORB_create()

kp_ref = orb.detect(ref,None)
kp_que = orb.detect(que,None)

kp_ref,des_ref = orb.compute(ref,kp_ref)
kp_que,des_que = orb.compute(que,kp_que)

# draw keypoints
ref_img2 = cv2.drawKeypoints(ref,kp_ref,None,color=(0,255,0),flags=0)
que_img2 = cv2.drawKeypoints(que,kp_que,None,color=(0,255,0),flags=0)

bf = cv2.BFMatcher(cv2.NORM_HAMMING)

# Match descriptors.
matches = bf.knnMatch(des_ref,des_que,k=2)

# Apply ratio test
good = []
good2 = []
for m,n in matches:
    if m.distance < 0.8*n.distance:
        good.append([m])
        good2.append(m)
img = cv2.drawMatchesKnn(ref_img2,kp_ref,que_img2,kp_que,good,None,flags=cv2.DrawMatchesFla

plt.title("Landmarks 11, ratio = 0.8")
plt.imshow(img), plt.show()

# Your code to display book location after RANSAC here
que_pts = np.float32([kp_ref[m.queryIdx].pt for m in good2 ]).reshape(-1,1,2)
ref_pts = np.float32([kp_que[m.trainIdx].pt for m in good2 ]).reshape(-1,1,2)

matrix, mask = cv2.findHomography(que_pts, ref_pts, cv2.RANSAC, 5.0)

# draw outline and inliers
plt.title("RANSAC Inliers for Landmarks 11, ratio = 0.8")
draw_inliers(ref, que, kp_ref, kp_que, good2, mask)
# draw outline
plt.title("RANSAC Outline for Landmarks 11, ratio = 0.8")
draw_outline(ref, que, matrix)

good = []
good2 = []
for m,n in matches:
    if m.distance < 0.9*n.distance:
        good.append([m])
        good2.append(m)
img = cv2.drawMatchesKnn(ref_img2,kp_ref,que_img2,kp_que,good,None,flags=cv2.DrawMatchesFla

plt.title("Landmarks 11, ratio = 0.9")
plt.imshow(img), plt.show()

# Your code to display book location after RANSAC here
```

```
que_pts = np.float32([kp_ref[m.queryIdx].pt for m in good2 ]).reshape(-1,1,2)
ref_pts = np.float32([kp_que[m.trainIdx].pt for m in good2 ]).reshape(-1,1,2)

matrix, mask = cv2.findHomography(que_pts, ref_pts, cv2.RANSAC, 5.0)

# draw outline and inliers
plt.title("RANSAC Inliers for Landmarks 11, ratio = 0.9")
draw_inliers(ref, que, kp_ref, kp_que, good2, mask)
# draw outline
plt.title("RANSAC Outline for Landmarks 11, ratio = 0.9")
draw_outline(ref, que, matrix)
```



Your explanation of results here

1. Book Covers

For book covers, it is relatively easy for the algorithm to find a good match, as the lighting is usually consistent, and the features are relatively sharper (text on plain background as opposed to softer edges in paintings). Additionally, most book covers are rectangular, and the edges make it easier to find the inliers, to apply RANSAC and find the outline.

2. Museum Paintings

For museum paintings, it is similar to book covers that the paintings are generally rectangular and the lighting is also fairly consistent, but the gentler gradient in color results in softer edges, which are harder to detect. Additionally, the frame that the painting is in can also interfere with the results of the matches, most prominently in the second example of museum paintings, where the subject matter of the painting and the frame of the painting are very similar, thus interfering with each other. Furthermore, in the third example, there is a huge lens flare in the center of the query image, which significantly affects the matches found, resulting in the frame being matched with parts of the painting, drawing the outline on the frame.

3. Landmarks

Finally, for landmarks, it is very difficult to obtain a good match and outline, as there are many factors in the query images that are different from the reference. For instance, in the first set of landmark images, the weather conditions are different, resulting in different lighting conditions for the images, making them harder to match. Additionally, elements in the image also differ since they are taken at different times. The position of cars and

people are not the same, which makes it harder for the algorithm to find good matches. To improve the result, I increased the ratio used in the ratio test, to allow for matches with less similarity to be used. While this improved the outline found, it is still incomplete, as many sections which match are still excluded. In the second example (011.jpg), the algorithm did a decent job of finding the outline, possibly due to the similar lighting and sharp edges, but I still had to intervene to change the ratio to 0.9, to generate a more accurate outline.

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

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

The steps are as follows:

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

In [13]:

```
# Your code to identify query objects and measure search accuracy for data set here
import os
refs = []
for file in os.listdir('book_covers/Reference'):
        ref = cv2.imread(os.path.join('book_covers/Reference',file),0)
        if ref is not None:
            refs.append(ref)
ques = []
for file in os.listdir('book_covers/Query'):
        que = cv2.imread(os.path.join('book_covers/Query',file),0)
        if que is not None:
            ques.append(que)
```

In [14]:

```
expectedindex = -1
success = 0
notindataset = 0
for que in ques:
    expectedindex = expectedindex + 1
    scores = []
    index = 0
    maxscore = 0
    maxindex = 0
    for ref in refs:
        # find the keypoints and descriptors with ORB
        kp_ref = orb.detect(ref, None)
        kp_que = orb.detect(que, None)

        kp_ref, des_ref = orb.compute(ref, kp_ref)
        kp_que, des_que = orb.compute(que, kp_que)

        # draw keypoints
        ref_img2 = cv2.drawKeypoints(ref, kp_ref, None, color=(0,255,0), flags=0)
        que_img2 = cv2.drawKeypoints(que, kp_que, None, color=(0,255,0), flags=0)

        # create BFMatcher object
        bf = cv2.BFMatcher(cv2.NORM_HAMMING)

        # Match descriptors.
        matches = bf.knnMatch(des_ref,des_que,k=2)

        # Apply ratio test
        good = []
        good2 = []
        for m,n in matches:
            if m.distance < 0.8 * n.distance:
                good.append([m])
                good2.append(m)
        img = cv2.drawMatchesKnn(ref_img2,kp_ref,que_img2,kp_que,good,None,flags=cv2.DrawMa

        # score is the number of good matches
        score = len(good)
        scores.append(score)
        if score > maxscore:
            maxscore = score
            maxindex = index
        index = index + 1

    # minimum of 30 good matches required
    if maxscore > 30:
        print("Max score: ",maxscore, ", at index: ",maxindex,"expected index: ",expectedin
        plt.title("Query Image")
        plt.imshow(que),plt.show()
        plt.title("Best Match Reference")
        plt.imshow(refs[maxindex]),plt.show()
        if maxindex == expectedindex:
            success = success + 1
    else:
        notindataset = notindataset + 1
        print("not in dataset")
        plt.title("Query Image")
        plt.imshow(que),plt.show()
```

```
Max score:  131 , at index:  0 expected index:  0
```

In [15]:

```
print("Success rate: ",success*100/len(ques),"%")
print("Queries with less than 30 good matches: ",notindataset)
```

```
Success rate:  60.396039603960396 %
Queries with less than 30 good matches:  24
```

Your explanation of what you have done, and your results, here By going through each query image found in book_covers/Query, and matching it against each reference image in book_covers/Reference, the reference image the produces the greatest number of good matches is chosen as the best match, with a minimum of 30 good matches. In the given dataset, each Query matches with the Reference of the same name (ie. Query/001.jpg matches Reference/001.jpg), therefore, by comparing the indexes of the query with the index of the best matched reference, we can tell if the match is accurate. By dividing the number of accurate matches over the total number of queries, we can find the success rate of the matching algorithm, which is about 60%. There are also 24 queries with less than 30 good matches, even though the correct reference image exists. However, for Query/055.jpg, which is an error image, Reference/070.jpg was found to be a match, with a score of 42.

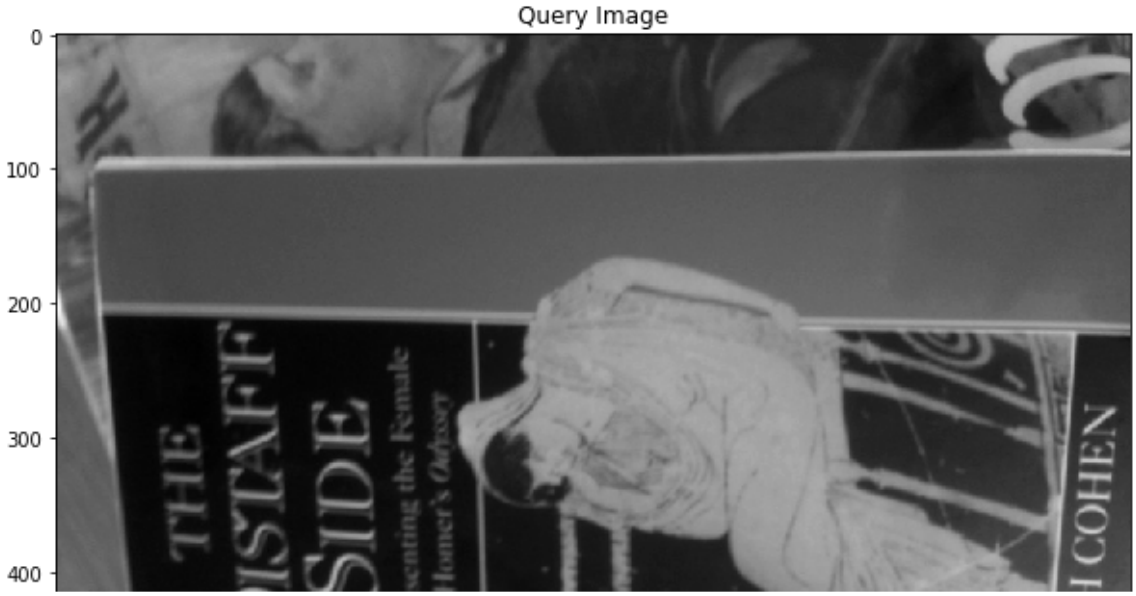
5. Choose some extra query images of objects that do not occur in the reference dataset. Repeat step 4 with these images added to your query set. Accuracy is now measured by the percentage of query images correctly identified in the dataset, or correctly identified as not occurring in the dataset. Report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

In [16]:

```
# Your code to run extra queries and display results here
# append the extra query images to the current list of query images
for file in os.listdir('book_covers/Query/ExtraQueries'):
        que = cv2.imread(os.path.join('book_covers/Query/ExtraQueries',file),0)
        if que is not None:
            ques.append(que)

expectedindex = -1
success = 0
notindataset = 0
for que in ques:
    expectedindex = expectedindex + 1
    scores = []
    index = 0
    maxscore = 0
    maxindex = 0
    for ref in refs:
        # find the keypoints and descriptors with ORB
        kp_ref = orb.detect(ref, None)
        kp_que = orb.detect(que, None)

        kp_ref, des_ref = orb.compute(ref, kp_ref)
        kp_que, des_que = orb.compute(que, kp_que)

        # draw keypoints
        ref_img2 = cv2.drawKeypoints(ref, kp_ref, None, color=(0,255,0), flags=0)
        que_img2 = cv2.drawKeypoints(que, kp_que, None, color=(0,255,0), flags=0)

        # create BFMatcher object
        bf = cv2.BFMatcher(cv2.NORM_HAMMING)

        # Match descriptors.
        matches = bf.knnMatch(des_ref,des_que,k=2)

        # Apply ratio test
        good = []
        good2 = []
        for m,n in matches:
            if m.distance < 0.8 * n.distance:
                good.append([m])
                good2.append(m)
        img = cv2.drawMatchesKnn(ref_img2,kp_ref,que_img2,kp_que,good,None,flags=cv2.DrawMa

        # score is the number of good matches
        score = len(good)
        scores.append(score)
        if score > maxscore:
            maxscore = score
            maxindex = index
        index = index + 1

    # minimum of 5 good matches required
    # original queries
    if maxscore > 30 and expectedindex < 101:
        print("Max score: ",maxscore, ", at index: ",maxindex,"expected index: ",expectedin
        plt.title("Query Image")
        plt.imshow(que),plt.show()
        plt.title("Best Match Reference")
        plt.imshow(refs[maxindex]),plt.show()
```

```
        if maxindex == expectedindex:
            success = success + 1
    # correctly identified as not in dataset
    elif expectedindex > 100:
        # if matches below threshold
        if maxscore < 31:
            success = success + 1
            notindataset = notindataset + 1
            print("not in dataset (correct)")
            plt.title("Query Image")
            plt.imshow(que),plt.show()
        # if a match is found
        else:
            print("Match found for query not in dataset, score: ",maxscore,", at index: ",m
            plt.title("Query Image")
            plt.imshow(que),plt.show()
            plt.title("Best Match Reference")
            plt.imshow(refs[maxindex]),plt.show()
    else:
        notindataset = notindataset + 1
        print("not in dataset")
        plt.title("Query Image")
        plt.imshow(que),plt.show()
```

```
Max score:  131 , at index:  0 expected index:  0
```



In [17]:

```
print("Success rate: ",success*100/len(ques),"%")
print("Queries with less than 30 good matches: ",notindataset)  # expected: 44
```

```
Success rate:  66.94214876033058 %
Queries with less than 30 good matches:  44
```

Your explanation of results and any changes made here

By adding 20 images from the museum_paintings/Query dataset, the success rate of the algorithm improved, as it is good at discerning images that do not have a match in the Reference dataset. Every single image added was correctly identified to not have a matching counterpart.

6. Repeat step 4 and 5 for at least one other set of reference images from museum paintings or landmarks,

and compare the accuracy obtained. Analyse both your overall result and individual image matches to diagnose where problems are occurring, and what you could do to improve performance. Test at least one of your proposed improvements and report its effect on accuracy.

In [18]:

```
# Your code to search images and display results here
refs = []
for file in os.listdir('museum_paintings/Reference'):
        ref = cv2.imread(os.path.join('museum_paintings/Reference',file),0)
        if ref is not None:
            refs.append(ref)
ques = []
for file in os.listdir('museum_paintings/Query'):
        que = cv2.imread(os.path.join('museum_paintings/Query',file),0)
        if que is not None:
            ques.append(que)
```

In [19]:

```
expectedindex = -1
success = 0
notindataset = 0
for que in ques:
    expectedindex = expectedindex + 1
    scores = []
    index = 0
    maxscore = 0
    maxindex = 0
    for ref in refs:
        # find the keypoints and descriptors with ORB
        kp_ref = orb.detect(ref, None)
        kp_que = orb.detect(que, None)

        kp_ref, des_ref = orb.compute(ref, kp_ref)
        kp_que, des_que = orb.compute(que, kp_que)

        # draw keypoints
        ref_img2 = cv2.drawKeypoints(ref, kp_ref, None, color=(0,255,0), flags=0)
        que_img2 = cv2.drawKeypoints(que, kp_que, None, color=(0,255,0), flags=0)

        # create BFMatcher object
        bf = cv2.BFMatcher(cv2.NORM_HAMMING)

        # Match descriptors.
        matches = bf.knnMatch(des_ref,des_que,k=2)

        # Apply ratio test
        good = []
        good2 = []
        for m,n in matches:
            if m.distance < 0.8 * n.distance:
                good.append([m])
                good2.append(m)
        img = cv2.drawMatchesKnn(ref_img2,kp_ref,que_img2,kp_que,good,None,flags=cv2.DrawMa

        # score is the number of good matches
        score = len(good)
        scores.append(score)
        if score > maxscore:
            maxscore = score
            maxindex = index
        index = index + 1

    # minimum of 30 good matches required
    if maxscore > 30:
        print("Max score: ",maxscore, ", at index: ",maxindex,"expected index: ",expectedin
        plt.title("Query Image")
        plt.imshow(que),plt.show()
        plt.title("Best Match Reference")
        plt.imshow(refs[maxindex]),plt.show()
        if maxindex == expectedindex:
            success = success + 1
    else:
        notindataset = notindataset + 1
        print("not in dataset")
        plt.title("Query Image")
        plt.imshow(que),plt.show()
```

```
Max score:  69 , at index:  0 expected index:  0
```

In [20]:

```
print("Success rate: ",success*100/len(ques),"%")
print("Queries with less than 30 good matches: ",notindataset)
```

```
Success rate:  38.46153846153846 %
Queries with less than 30 good matches:  44
```

In [21]:

```
expectedindex = -1
success = 0
notindataset = 0
orb2 = cv2.ORB_create(scaleFactor = 1.5)
for que in ques:
    expectedindex = expectedindex + 1
    scores = []
    index = 0
    maxscore = 0
    maxindex = 0
    for ref in refs:
        # find the keypoints and descriptors with ORB
        kp_ref = orb2.detect(ref, None)
        kp_que = orb2.detect(que, None)

        kp_ref, des_ref = orb2.compute(ref, kp_ref)
        kp_que, des_que = orb2.compute(que, kp_que)

        # draw keypoints
        ref_img2 = cv2.drawKeypoints(ref, kp_ref, None, color=(0,255,0), flags=0)
        que_img2 = cv2.drawKeypoints(que, kp_que, None, color=(0,255,0), flags=0)

        # create BFMatcher object
        bf = cv2.BFMatcher(cv2.NORM_HAMMING)

        # Match descriptors.
        matches = bf.knnMatch(des_ref,des_que,k=2)

        # Apply ratio test
        good = []
        good2 = []
        for m,n in matches:
            if m.distance < 0.8 * n.distance:
                good.append([m])
                good2.append(m)
        img = cv2.drawMatchesKnn(ref_img2,kp_ref,que_img2,kp_que,good,None,flags=cv2.DrawMa

        # score is the number of good matches
        score = len(good)
        scores.append(score)
        if score > maxscore:
            maxscore = score
            maxindex = index
        index = index + 1

    # minimum of 30 good matches required
    if maxscore > 30:
        print("Max score: ",maxscore, ", at index: ",maxindex,"expected index: ",expectedin
        plt.title("Query Image")
        plt.imshow(que),plt.show()
        plt.title("Best Match Reference")
        plt.imshow(refs[maxindex]),plt.show()
        if maxindex == expectedindex:
            success = success + 1
    else:
        notindataset = notindataset + 1
        print("not in dataset")
        plt.title("Query Image")
        plt.imshow(que),plt.show()
```

```
Max score:  53 , at index:  0 expected index:  0
```

In [22]:

```
print("Success rate: ",success*100/len(ques),"%")
print("Queries with less than 30 good matches: ",notindataset)
```

```
Success rate:  48.35164835164835 %
Queries with less than 30 good matches:  32
```

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

By increasing the scale factor of the ORB used, the accuracy of the matches can be raised by 10%. Using a larger scale factor, the decimation factor of the image pyramid will be increased, thus each level of the pyramid will have less pixels.

Question 3 (10%)

In Question 1, We hope that `ratio_test` can provide reasonable results for RANSAC. However, if it fails, the RANSAC may not get good results. In this case, we would like to try an improved matching method to replace the `ratio_test` . Here, the `gms_matcher` is recommened. You need to implement it and save results of 3 image pairs (you can select any image pairs from the dataset), where you new method is better than 'ratio_test'.

1. Hint 1: `cv2.xfeatures2d.matchGMS()` can be used, but you need to install the opencv-contrib by `pip install opencv-contrib-python`
2. Hint 2: You do not need use KNN matching, because GMS does not require second nearest neighbor.
3. Hint 3: You need to change the parameters in `cv2.ORB_create()` for best results. See the setting in Github.
4. Hint 4: If your are interested in more details. Read the paper "GMS: Grid-based Motion Statistics for Fast, Ultra-robust Feature Correspondence", and the Github "https://github.com/JiawangBian/GMS-Feature-Matcher" (https://github.com/JiawangBian/GMS-Feature-Matcher").

In [23]:

```
# from the github because I had difficulty importing
```

In [24]:

```
import math
from enum import Enum

import cv2
cv2.ocl.setUseOpenCL(False)

import numpy as np

THRESHOLD_FACTOR = 6

ROTATION_PATTERNS = [
    [1, 2, 3,
     4, 5, 6,
     7, 8, 9],

    [4, 1, 2,
     7, 5, 3,
     8, 9, 6],

    [7, 4, 1,
     8, 5, 2,
     9, 6, 3],

    [8, 7, 4,
     9, 5, 1,
     6, 3, 2],

    [9, 8, 7,
     6, 5, 4,
     3, 2, 1],

    [6, 9, 8,
     3, 5, 7,
     2, 1, 4],

    [3, 6, 9,
     2, 5, 8,
     1, 4, 7],

    [2, 3, 6,
     1, 5, 9,
     4, 7, 8]]

class Size:
    def __init__(self, width, height):
        self.width = width
        self.height = height

class DrawingType(Enum):
    ONLY_LINES = 1
    LINES_AND_POINTS = 2
    COLOR_CODED_POINTS_X = 3
    COLOR_CODED_POINTS_Y = 4
    COLOR_CODED_POINTS_XpY = 5


class GmsMatcher:
    def __init__(self, descriptor, matcher):
        self.scale_ratios = [1.0, 1.0 / 2, 1.0 / math.sqrt(2.0), math.sqrt(2.0), 2.0]
```

```
        # Normalized vectors of 2D points
        self.normalized_points1 = []
        self.normalized_points2 = []
        # Matches - list of pairs representing numbers
        self.matches = []
        self.matches_number = 0
        # Grid Size
        self.grid_size_right = Size(0, 0)
        self.grid_number_right = 0
        # x      : left grid idx
        # y      :  right grid idx
        # value  : how many matches from idx_left to idx_right
        self.motion_statistics = []

        self.number_of_points_per_cell_left = []
        # Inldex  : grid_idx_left
        # Value   : grid_idx_right
        self.cell_pairs = []

        # Every Matches has a cell-pair
        # first  : grid_idx_left
        # second : grid_idx_right
        self.match_pairs = []

        # Inlier Mask for output
        self.inlier_mask = []
        self.grid_neighbor_right = []

        # Grid initialize
        self.grid_size_left = Size(20, 20)
        self.grid_number_left = self.grid_size_left.width * self.grid_size_left.height

        # Initialize the neihbor of left grid
        self.grid_neighbor_left = np.zeros((self.grid_number_left, 9))

        self.descriptor = descriptor
        self.matcher = matcher
        self.gms_matches = []
        self.keypoints_image1 = []
        self.keypoints_image2 = []

    def empty_matches(self):
        self.normalized_points1 = []
        self.normalized_points2 = []
        self.matches = []
        self.gms_matches = []

    def compute_matches(self, img1, img2):
        self.keypoints_image1, descriptors_image1 = self.descriptor.detectAndCompute(img1,
        self.keypoints_image2, descriptors_image2 = self.descriptor.detectAndCompute(img2,
        size1 = Size(img1.shape[1], img1.shape[0])
        size2 = Size(img2.shape[1], img2.shape[0])

        if self.gms_matches:
            self.empty_matches()

        all_matches = self.matcher.match(descriptors_image1, descriptors_image2)
        self.normalize_points(self.keypoints_image1, size1, self.normalized_points1)
        self.normalize_points(self.keypoints_image2, size2, self.normalized_points2)
        self.matches_number = len(all_matches)
        self.convert_matches(all_matches, self.matches)
```

```
        self.initialize_neighbours(self.grid_neighbor_left, self.grid_size_left)

        mask, num_inliers = self.get_inlier_mask(False, False)
        print('Found', num_inliers, 'matches')

        for i in range(len(mask)):
            if mask[i]:
                self.gms_matches.append(all_matches[i])
        return self.gms_matches

    # Normalize Key points to range (0-1)
    def normalize_points(self, kp, size, npts):
        for keypoint in kp:
            npts.append((keypoint.pt[0] / size.width, keypoint.pt[1] / size.height))

    # Convert OpenCV match to list of tuples
    def convert_matches(self, vd_matches, v_matches):
        for match in vd_matches:
            v_matches.append((match.queryIdx, match.trainIdx))

    def initialize_neighbours(self, neighbor, grid_size):
        for i in range(neighbor.shape[0]):
            neighbor[i] = self.get_nb9(i, grid_size)

    def get_nb9(self, idx, grid_size):
        nb9 = [-1 for _ in range(9)]
        idx_x = idx % grid_size.width
        idx_y = idx // grid_size.width

        for yi in range(-1, 2):
            for xi in range(-1, 2):
                idx_xx = idx_x + xi
                idx_yy = idx_y + yi

                if idx_xx < 0 or idx_xx >= grid_size.width or idx_yy < 0 or idx_yy >= grid_
                    continue
                nb9[xi + 4 + yi * 3] = idx_xx + idx_yy * grid_size.width

        return nb9

    def get_inlier_mask(self, with_scale, with_rotation):
        max_inlier = 0
        self.set_scale(0)

        if not with_scale and not with_rotation:
            max_inlier = self.run(1)
            return self.inlier_mask, max_inlier
        elif with_scale and with_rotation:
            vb_inliers = []
            for scale in range(5):
                self.set_scale(scale)
                for rotation_type in range(1, 9):
                    num_inlier = self.run(rotation_type)
                    if num_inlier > max_inlier:
                        vb_inliers = self.inlier_mask
                        max_inlier = num_inlier

            if vb_inliers != []:
                return vb_inliers, max_inlier
            else:
                return self.inlier_mask, max_inlier
```

```
        elif with_rotation and not with_scale:
            vb_inliers = []
            for rotation_type in range(1, 9):
                num_inlier = self.run(rotation_type)
                if num_inlier > max_inlier:
                    vb_inliers = self.inlier_mask
                    max_inlier = num_inlier

            if vb_inliers != []:
                return vb_inliers, max_inlier
            else:
                return self.inlier_mask, max_inlier
        else:
            vb_inliers = []
            for scale in range(5):
                self.set_scale(scale)
                num_inlier = self.run(1)
                if num_inlier > max_inlier:
                    vb_inliers = self.inlier_mask
                    max_inlier = num_inlier

            if vb_inliers != []:
                return vb_inliers, max_inlier
            else:
                return self.inlier_mask, max_inlier

    def set_scale(self, scale):
        self.grid_size_right.width = self.grid_size_left.width * self.scale_ratios[scale]
        self.grid_size_right.height = self.grid_size_left.height * self.scale_ratios[scale]
        self.grid_number_right = self.grid_size_right.width * self.grid_size_right.height

        # Initialize the neighbour of right grid
        self.grid_neighbor_right = np.zeros((int(self.grid_number_right), 9))
        self.initialize_neighbours(self.grid_neighbor_right, self.grid_size_right)

    def run(self, rotation_type):
        self.inlier_mask = [False for _ in range(self.matches_number)]

        # Initialize motion statistics
        self.motion_statistics = np.zeros((int(self.grid_number_left), int(self.grid_number
        self.match_pairs = [[0, 0] for _ in range(self.matches_number)]

        for GridType in range(1, 5):
            self.motion_statistics = np.zeros((int(self.grid_number_left), int(self.grid_nu
            self.cell_pairs = [-1 for _ in range(self.grid_number_left)]
            self.number_of_points_per_cell_left = [0 for _ in range(self.grid_number_left)]

            self.assign_match_pairs(GridType)
            self.verify_cell_pairs(rotation_type)

            # Mark inliers
            for i in range(self.matches_number):
                if self.cell_pairs[int(self.match_pairs[i][0])] == self.match_pairs[i][1]:
                    self.inlier_mask[i] = True

        return sum(self.inlier_mask)

    def assign_match_pairs(self, grid_type):
        for i in range(self.matches_number):
            lp = self.normalized_points1[self.matches[i][0]]
            rp = self.normalized_points2[self.matches[i][1]]
```

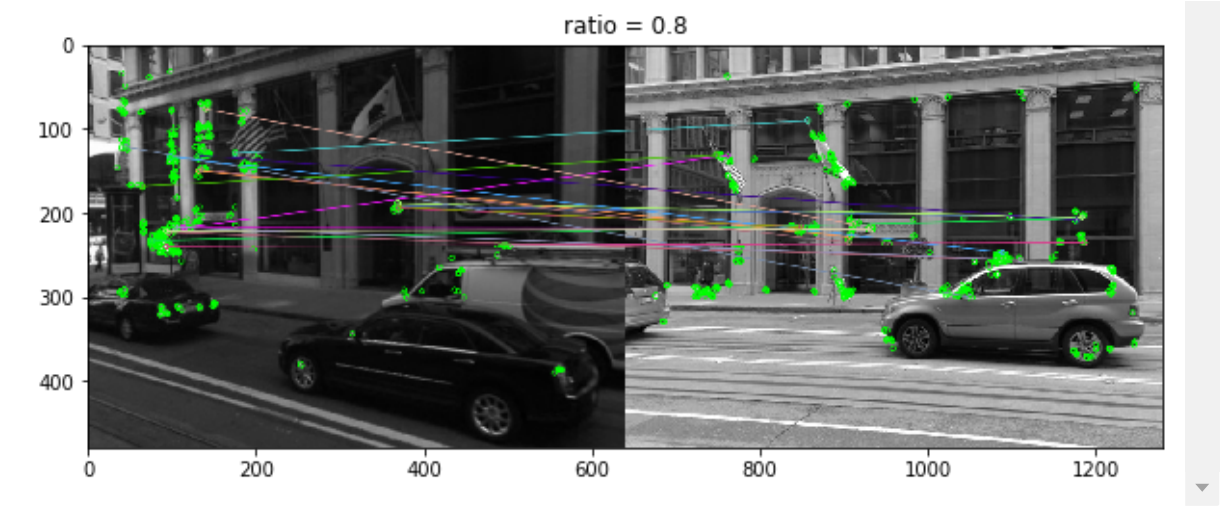
```
            lgidx = self.match_pairs[i][0] = self.get_grid_index_left(lp, grid_type)

            if grid_type == 1:
                rgidx = self.match_pairs[i][1] = self.get_grid_index_right(rp)
            else:
                rgidx = self.match_pairs[i][1]

            if lgidx < 0 or rgidx < 0:
                continue
            self.motion_statistics[int(lgidx)][int(rgidx)] += 1
            self.number_of_points_per_cell_left[int(lgidx)] += 1

    def get_grid_index_left(self, pt, type_of_grid):
        x = pt[0] * self.grid_size_left.width
        y = pt[1] * self.grid_size_left.height

        if type_of_grid == 2:
            x += 0.5
        elif type_of_grid == 3:
            y += 0.5
        elif type_of_grid == 4:
            x += 0.5
            y += 0.5

        x = math.floor(x)
        y = math.floor(y)

        if x >= self.grid_size_left.width or y >= self.grid_size_left.height:
            return -1
        return x + y * self.grid_size_left.width

    def get_grid_index_right(self, pt):
        x = int(math.floor(pt[0] * self.grid_size_right.width))
        y = int(math.floor(pt[1] * self.grid_size_right.height))
        return x + y * self.grid_size_right.width

    def verify_cell_pairs(self, rotation_type):
        current_rotation_pattern = ROTATION_PATTERNS[rotation_type - 1]

        for i in range(self.grid_number_left):
            if sum(self.motion_statistics[i]) == 0:
                self.cell_pairs[i] = -1
                continue
            max_number = 0
            for j in range(int(self.grid_number_right)):
                value = self.motion_statistics[i]
                if value[j] > max_number:
                    self.cell_pairs[i] = j
                    max_number = value[j]

            idx_grid_rt = self.cell_pairs[i]
            nb9_lt = self.grid_neighbor_left[i]
            nb9_rt = self.grid_neighbor_right[idx_grid_rt]
            score = 0
            thresh = 0
            numpair = 0

            for j in range(9):
                ll = nb9_lt[j]
                rr = nb9_rt[current_rotation_pattern[j] - 1]
                if ll == -1 or rr == -1:
```

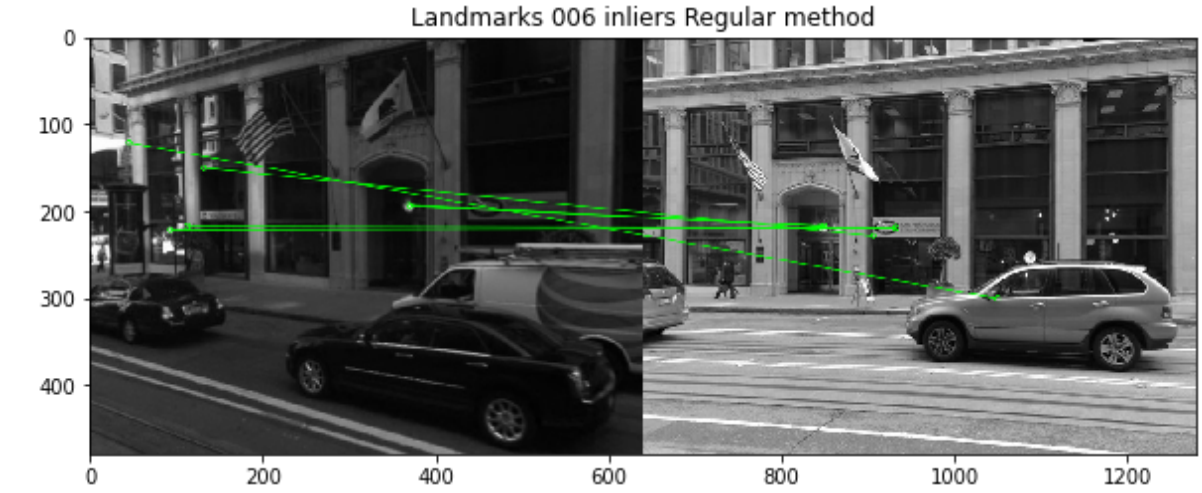
```
                    continue

                score += self.motion_statistics[int(ll), int(rr)]
                thresh += self.number_of_points_per_cell_left[int(ll)]
                numpair += 1

            thresh = THRESHOLD_FACTOR * math.sqrt(thresh/numpair)
            if score < thresh:
                self.cell_pairs[i] = -2

    def draw_matches(self, src1, src2, drawing_type):
        height = max(src1.shape[0], src2.shape[0])
        width = src1.shape[1] + src2.shape[1]
        output = np.zeros((height, width, 3), dtype=np.uint8)
        output[0:src1.shape[0], 0:src1.shape[1]] = src1
        output[0:src2.shape[0], src1.shape[1]:] = src2[:]

        if drawing_type == DrawingType.ONLY_LINES:
            for i in range(len(self.gms_matches)):
                left = self.keypoints_image1[self.gms_matches[i].queryIdx].pt
                right = tuple(sum(x) for x in zip(self.keypoints_image2[self.gms_matches[i]
                cv2.line(output, tuple(map(int, left)), tuple(map(int, right)), (0, 255, 25

        elif drawing_type == DrawingType.LINES_AND_POINTS:
            for i in range(len(self.gms_matches)):
                left = self.keypoints_image1[self.gms_matches[i].queryIdx].pt
                right = tuple(sum(x) for x in zip(self.keypoints_image2[self.gms_matches[i]
                cv2.line(output, tuple(map(int, left)), tuple(map(int, right)), (255, 0, 0)

            for i in range(len(self.gms_matches)):
                left = self.keypoints_image1[self.gms_matches[i].queryIdx].pt
                right = tuple(sum(x) for x in zip(self.keypoints_image2[self.gms_matches[i]
                cv2.circle(output, tuple(map(int, left)), 1, (0, 255, 255), 2)
                cv2.circle(output, tuple(map(int, right)), 1, (0, 255, 0), 2)

        elif drawing_type == DrawingType.COLOR_CODED_POINTS_X or drawing_type == DrawingTyp
            _1_255 = np.expand_dims( np.array( range( 0, 256 ), dtype='uint8' ), 1 )
            _colormap = cv2.applyColorMap(_1_255, cv2.COLORMAP_HSV)

            for i in range(len(self.gms_matches)):
                left = self.keypoints_image1[self.gms_matches[i].queryIdx].pt
                right = tuple(sum(x) for x in zip(self.keypoints_image2[self.gms_matches[i]

                if drawing_type == DrawingType.COLOR_CODED_POINTS_X:
                    colormap_idx = int(left[0] * 256. / src1.shape[1] ) # x-gradient
                if drawing_type == DrawingType.COLOR_CODED_POINTS_Y:
                    colormap_idx = int(left[1] * 256. / src1.shape[0] ) # y-gradient
                if drawing_type == DrawingType.COLOR_CODED_POINTS_XpY:
                    colormap_idx = int( (left[0] - src1.shape[1]*.5 + left[1] - src1.shape[

                color = tuple( map(int, _colormap[ colormap_idx,0,: ]) )
                cv2.circle(output, tuple(map(int, left)), 1, color, 2)
                cv2.circle(output, tuple(map(int, right)), 1, color, 2)


#         cv2.imshow('show', output)
#         cv2.waitKey()

#         image = cv2.imread("input_path")
        #Show the image with matplotlib
        plt.imshow(cv2.cvtColor(output, cv2.COLOR_BGR2RGB))
```

```
        plt.show()
```

In [25]:

```
# regular method
# load image at gray scale
ref = cv2.imread('landmarks/Reference/006.jpg',0)
que = cv2.imread('landmarks/Query/006.jpg',0)

# compute detector and descriptor
orb = cv2.ORB_create()
# find the keypoints and descriptors with ORB
kp_ref = orb.detect(ref,None)
kp_que = orb.detect(que,None)

kp_ref,des_ref = orb.compute(ref,kp_ref)
kp_que,des_que = orb.compute(que,kp_que)

# draw keypoints
ref_img2 = cv2.drawKeypoints(ref,kp_ref,None,color=(0,255,0),flags=0)
que_img2 = cv2.drawKeypoints(que,kp_que,None,color=(0,255,0),flags=0)

# create BFMatcher object
bf = cv2.BFMatcher(cv2.NORM_HAMMING)

# Match descriptors.
matches = bf.knnMatch(des_ref,des_que,k=2)

# Apply ratio test
good = []
good2 = []
for m,n in matches:
    if m.distance < 0.8*n.distance:
        good.append([m])
        good2.append(m)
img = cv2.drawMatchesKnn(ref_img2,kp_ref,que_img2,kp_que,good,None,flags=cv2.DrawMatchesFla

# draw matches
plt.title("ratio = 0.8")
plt.imshow(img),plt.show()

que_pts = np.float32([kp_ref[m.queryIdx].pt for m in good2 ]).reshape(-1,1,2)
ref_pts = np.float32([kp_que[m.trainIdx].pt for m in good2 ]).reshape(-1,1,2)

matrix, mask = cv2.findHomography(que_pts, ref_pts, cv2.RANSAC, 5.0)

# draw outline and inliers
plt.title("Landmarks 006 inliers Regular method")
draw_inliers(ref, que, kp_ref, kp_que, good2, mask)
# draw outline
plt.title("Landmarks 006 outline Regular method")
draw_outline(ref, que, matrix)
```


ratio = 0.8
0
100
200
300
400
0
200
400
600
800
1000
1200

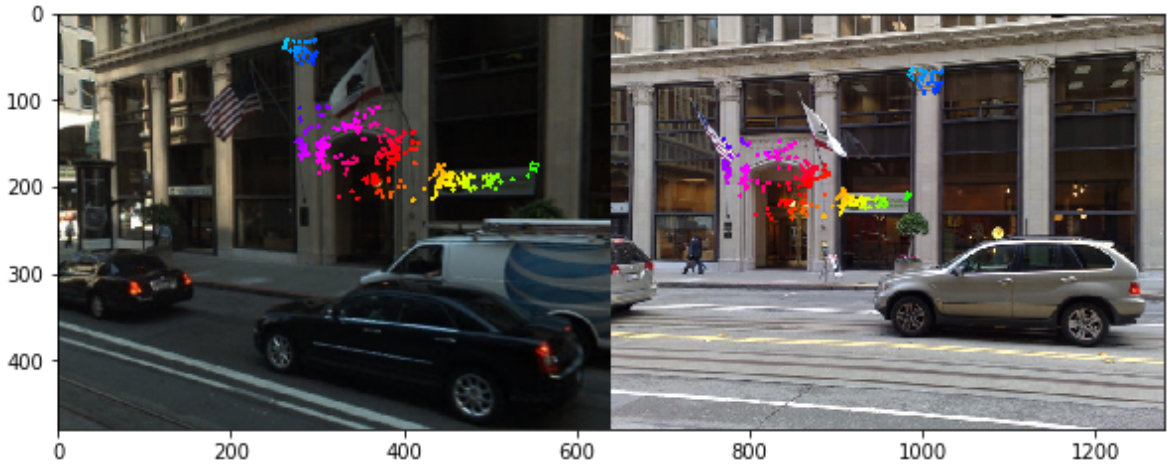

Landmarks 006 inliers Regular method
0
100
200
300
400
0
200
400
600
800
1000
1200


Landmarks 006 outline Regular method
0
100
200
300
400
0
100
200
300
400
500
600

In [26]:

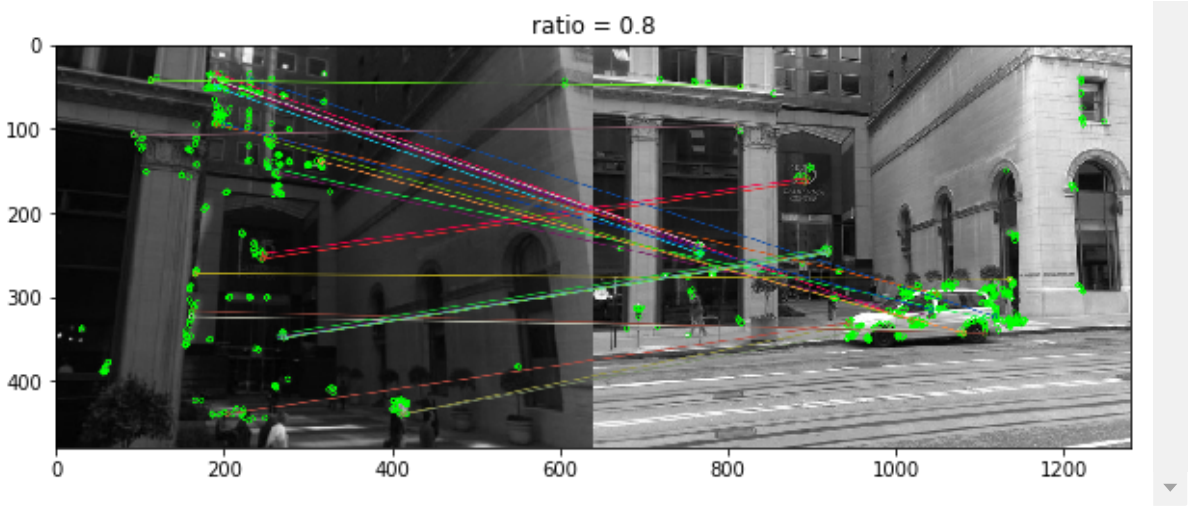
```
#using GMS matcher
ref = cv2.imread("landmarks/Reference/006.jpg")
que = cv2.imread("landmarks/Query/006.jpg")

orb = cv2.ORB_create(10000)
orb.setFastThreshold(0)

kp_ref,des_ref = orb.compute(ref,kp_ref)
kp_que,des_que = orb.compute(que,kp_que)

matcher = cv2.BFMatcher(cv2.NORM_HAMMING)

gms = GmsMatcher(orb,matcher)

matches = gms.compute_matches(ref, que)
print("Landmarks 006 GMS")
gms.draw_matches(ref, que, DrawingType.COLOR_CODED_POINTS_XpY)
```

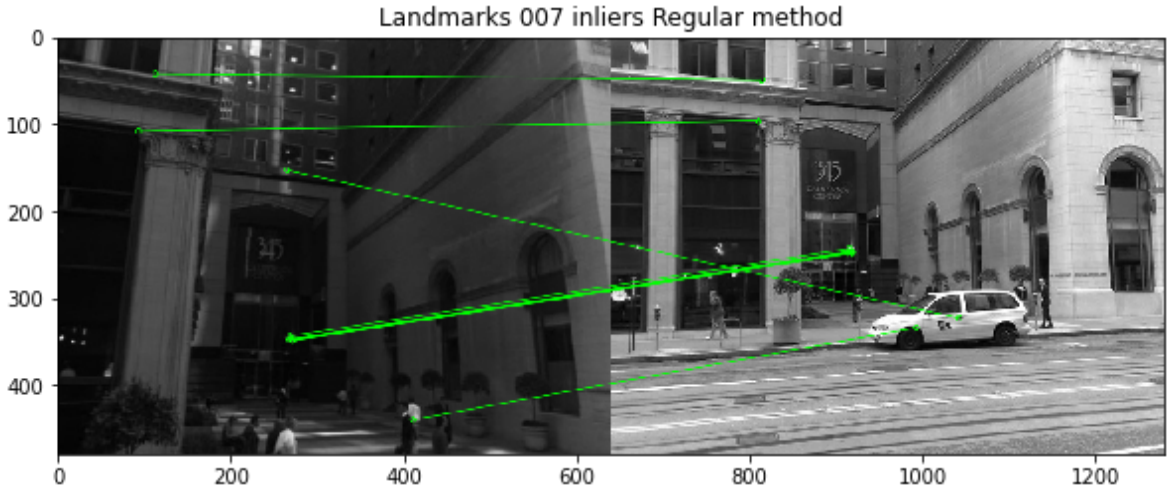
```
Found 423 matches
Landmarks 006 GMS
```

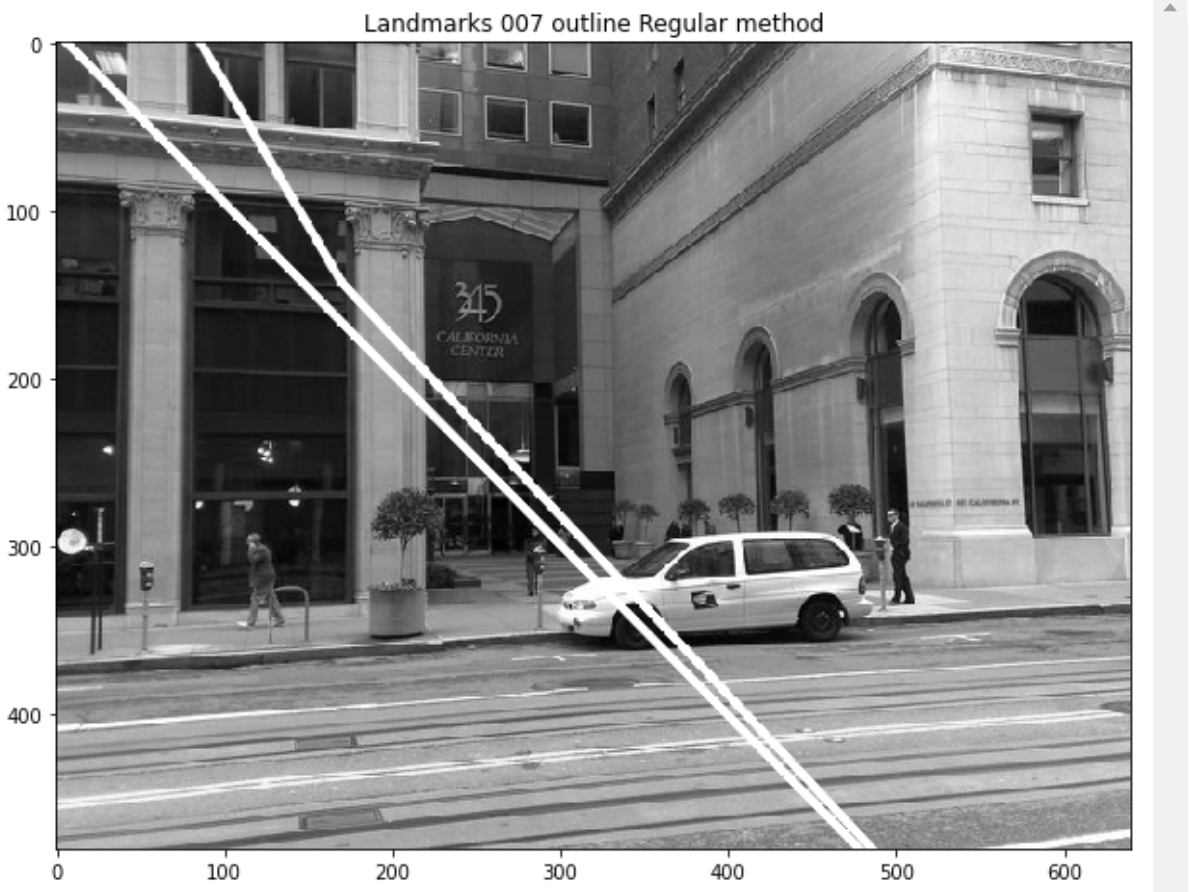


In [27]:

```
# regular method
# load image at gray scale
ref = cv2.imread('landmarks/Reference/007.jpg',0)
que = cv2.imread('landmarks/Query/007.jpg',0)

# compute detector and descriptor
orb = cv2.ORB_create()
# find the keypoints and descriptors with ORB
kp_ref = orb.detect(ref,None)
kp_que = orb.detect(que,None)

kp_ref,des_ref = orb.compute(ref,kp_ref)
kp_que,des_que = orb.compute(que,kp_que)

# draw keypoints
ref_img2 = cv2.drawKeypoints(ref,kp_ref,None,color=(0,255,0),flags=0)
que_img2 = cv2.drawKeypoints(que,kp_que,None,color=(0,255,0),flags=0)

# create BFMatcher object
bf = cv2.BFMatcher(cv2.NORM_HAMMING)

# Match descriptors.
matches = bf.knnMatch(des_ref,des_que,k=2)

# Apply ratio test
good = []
good2 = []
for m,n in matches:
    if m.distance < 0.8*n.distance:
        good.append([m])
        good2.append(m)
img = cv2.drawMatchesKnn(ref_img2,kp_ref,que_img2,kp_que,good,None,flags=cv2.DrawMatchesFla

# draw matches
plt.title("ratio = 0.8")
plt.imshow(img),plt.show()

que_pts = np.float32([kp_ref[m.queryIdx].pt for m in good2 ]).reshape(-1,1,2)
ref_pts = np.float32([kp_que[m.trainIdx].pt for m in good2 ]).reshape(-1,1,2)

matrix, mask = cv2.findHomography(que_pts, ref_pts, cv2.RANSAC, 5.0)

# draw outline and inliers
plt.title("Landmarks 007 inliers Regular method")
draw_inliers(ref, que, kp_ref, kp_que, good2, mask)
# draw outline
plt.title("Landmarks 007 outline Regular method")
draw_outline(ref, que, matrix)
```


ratio = 0.8
0
100
200
300
400
0
200
400
600
800
1000
1200

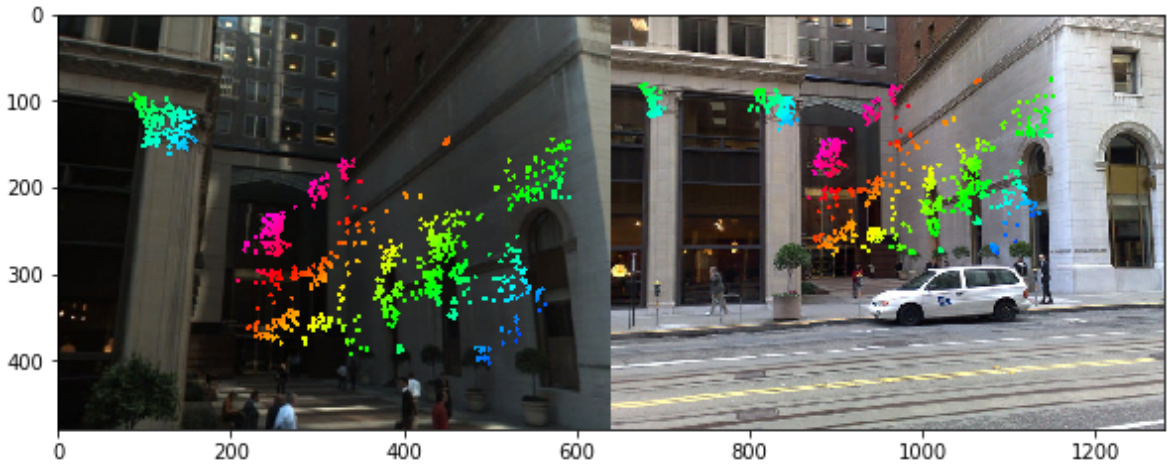

Landmarks 007 inliers Regular method
0
100
200
300
400
0
200
400
600
800
1000
1200


Landmarks 007 outline Regular method
0
100
200
300
400
0
100
200
300
400
500
600

In [28]:

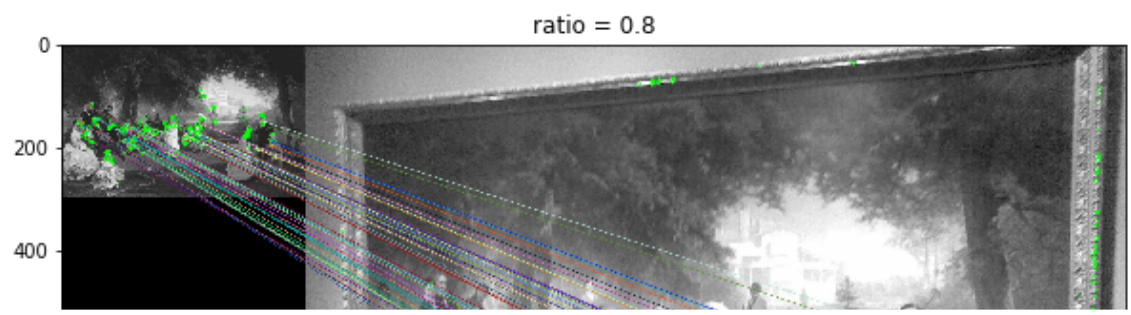
```
#using GMS matcher
ref = cv2.imread("landmarks/Reference/007.jpg")
que = cv2.imread("landmarks/Query/007.jpg")

orb = cv2.ORB_create(10000)
orb.setFastThreshold(0)

kp_ref,des_ref = orb.compute(ref,kp_ref)
kp_que,des_que = orb.compute(que,kp_que)

matcher = cv2.BFMatcher(cv2.NORM_HAMMING)

gms = GmsMatcher(orb,matcher)

matches = gms.compute_matches(ref, que)
print("Landmarks 007 GMS")
gms.draw_matches(ref, que, DrawingType.COLOR_CODED_POINTS_XpY)
```

```
Found 1141 matches
Landmarks 007 GMS
```



In [29]:

```
# regular method
# load image at gray scale
ref = cv2.imread('museum_paintings/Reference/001.jpg',0)
que = cv2.imread('museum_paintings/Query/001.jpg',0)

# compute detector and descriptor
orb = cv2.ORB_create()
# find the keypoints and descriptors with ORB
kp_ref = orb.detect(ref,None)
kp_que = orb.detect(que,None)

kp_ref,des_ref = orb.compute(ref,kp_ref)
kp_que,des_que = orb.compute(que,kp_que)

# draw keypoints
ref_img2 = cv2.drawKeypoints(ref,kp_ref,None,color=(0,255,0),flags=0)
que_img2 = cv2.drawKeypoints(que,kp_que,None,color=(0,255,0),flags=0)

# create BFMatcher object
bf = cv2.BFMatcher(cv2.NORM_HAMMING)

# Match descriptors.
matches = bf.knnMatch(des_ref,des_que,k=2)

# Apply ratio test
good = []
good2 = []
for m,n in matches:
    if m.distance < 0.8*n.distance:
        good.append([m])
        good2.append(m)
img = cv2.drawMatchesKnn(ref_img2,kp_ref,que_img2,kp_que,good,None,flags=cv2.DrawMatchesFla

# draw matches
plt.title("ratio = 0.8")
plt.imshow(img),plt.show()

que_pts = np.float32([kp_ref[m.queryIdx].pt for m in good2 ]).reshape(-1,1,2)
ref_pts = np.float32([kp_que[m.trainIdx].pt for m in good2 ]).reshape(-1,1,2)

matrix, mask = cv2.findHomography(que_pts, ref_pts, cv2.RANSAC, 5.0)

# draw outline and inliers
plt.title("Museum Paintings 001 inliers Regular method")
draw_inliers(ref, que, kp_ref, kp_que, good2, mask)
# draw outline
plt.title("Museum Paintings 001 outline Regular method")
draw_outline(ref, que, matrix)
```



In [30]:

```
#using GMS matcher
ref = cv2.imread("museum_paintings/Reference/001.jpg")
que = cv2.imread("museum_paintings/Query/001.jpg")

orb = cv2.ORB_create(10000)
orb.setFastThreshold(0)

kp_ref,des_ref = orb.compute(ref,kp_ref)
kp_que,des_que = orb.compute(que,kp_que)

matcher = cv2.BFMatcher(cv2.NORM_HAMMING)

gms = GmsMatcher(orb,matcher)

matches = gms.compute_matches(ref, que)
print("Museum Paintings 001 GMS")
gms.draw_matches(ref, que, DrawingType.COLOR_CODED_POINTS_XpY)
```

```
Found 1582 matches
Museum Paintings 001 GMS
```

Your results here

Using the GMS Matcher, some query and reference image pairs return better results, such as landmarks/Query/006.jpg, landmarks/Query/007.jpg, and museum_paintings/Query/001.jpg, where GMS was able to find many more matches than the regular ratio test method. Furthermore, as seen from these examples,

GMS is better able to distinguish softer edges, especially those found in the paintings. This may be due to the fact that GMS was computed in color, where there is more variation in the edges.

Question 4: Reflection Questions (5%)

1. Describe the hardest situation you faced during the first two assignments. And how you overcome it? (3%)

The hardest situation I faced during the first two assignments is actually right now, 5 hours before the submission deadline for Assignment 2, which has a 40% hurdle. As I am typing this reflection, I am running the code for question 2, which has to do tens of thousands of comparisons between images. Due to a combination of my procrastination and obligations for other courses, I have put off this assignment to the last minute, but expectedly, since this assignment requires a lot of runtime, I have less margin for error, as each mistake in my code can cost precious time. To overcome this, I must start doing my assignments in advance, especially for assignments like these, where the datasets require a lot of time. For the competition, especially, where the datasets are much larger, I will work with my teammate closely to do our work early, so that we do not have to rush come the deadline.

2. How do you plan to finish the assignment to meet tight deadline? (2%)

As stated above, especially due to the large datasets involved, it is important to do my assignments well before the deadline, to avoid rushing on the day of submission. To meet the tight deadline, I will maintain a good schedule for doing my assignments, such that they do not pile up. Since many courses have assignments have deadlines that are very close to each other, I will start doing assignments well in advance, to avoid burning out once the deadlines start coming up

In [ ]: