Rept_raymo_BA_20667930

April 5, 2021

Task 7: Feature Matching & RANSAC

Name: Raymond Ren
Degree: Bachelors
ID: 20667930

[1]:
```
# import libraries
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import null_space
from scipy import io
import random
import cv2
import os
```

Problem 1: LoG and DoG (10 points)

Please answer these questions in detail.

(a) What is the Laplacian of Gaussian (LoG)? When do we use LoG?

$$L = \sigma^2(G_{xx}(x, y, k\sigma) + G_{yy}(x, y, \sigma))$$

The Laplacian of Gaussian (LOG) is a second derivative image filter obtained by applying the Laplacian operator to an image convolved by a Gaussian kernel. Since the derivative operation is very sensitive to noise, the Gaussian filter is included to smooth the image before applying the Laplacian. LOG filters highlights areas with rapid intensity change and is therefore commonly used as an edge detection filter.

(b) What is the Difference of Gaussian (DoG)? When do we use and why DoG is advantageous compared to LoG?

$$DoG = G(x, y, k\sigma) - G(x, y, \sigma)$$

The Difference of Gaussian (DoG) filter is obtained by applying two Gaussian kernels with different standard deviations to an image and subtracting the less blurred resulting image from the more blurred image. Taking the difference between two filtered images allows the DoG to remove high-frequency noise components as well as low-frequency (homogenous) areas in the image. Like the LoG, the DoG acts as an edge detection and/or feature enhancement filter. Compared to the LoG, the DoG is more computationally efficient since it is separable, and therefore requires two

1D convolutions as opposed to the more computationally intensive 2D convolution required for the LoG.

Problem 2: Least Squares (20 points)

(a) Explain the approach 1 and approach 2 for least-square line fitting in your words. Please refer to the course slide and tutorials.

In Approach 1, the error is defined by:

$$E = \sum_{i=1}^{n} (y_i - mx_i - b)^2$$

The error is the sum of squares of the error for each of the n points. The error is minimized by taking the partial derivatives of E with respect to m and b and setting them to zero. Solving the resulting system of equations results in the following solutions to m and b:

$$m = \frac{\sum_{i=1}^{n} x_i y_i - \frac{1}{n}(\sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i)}{\sum_{i=1}^{n} x_i^2 - \frac{1}{n}(\sum_{i=1}^{n} x_i)^2}$$

$$b = \frac{\frac{1}{n}\sum_{i=1}^{n} y_i \sum_{i=1}^{n} x_i^2 - \frac{1}{n}\sum_{i=1}^{n} x_i \sum_{i=1}^{n} x_i y_i}{\sum_{i=1}^{n} x_i^2 - \frac{1}{n}(\sum_{i=1}^{n} x_i)^2}$$

In Approach 2, the error is defined by:

$$E = \|Y - XB\|^2$$

where

$$Y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}, X = \begin{bmatrix} x_1 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{bmatrix}, B = \begin{bmatrix} m \\ b \end{bmatrix}$$

This vectorized approach is essentially the same as Approach 1. The derivative of the sum of squares error is set to zero and the resulting equation is solved. This results in the following solution:

$$B = (X'X)^{-1}X'Y$$

A response yi is measured from a linear system when an input is xi. The measurement data is provided (prob2_data1.mat). A model of your system can be approximated as yi = mxi + b. Please find m and b using the following methods.

(b) Least squares (approach 1)

(c) Least squares (approach 2)

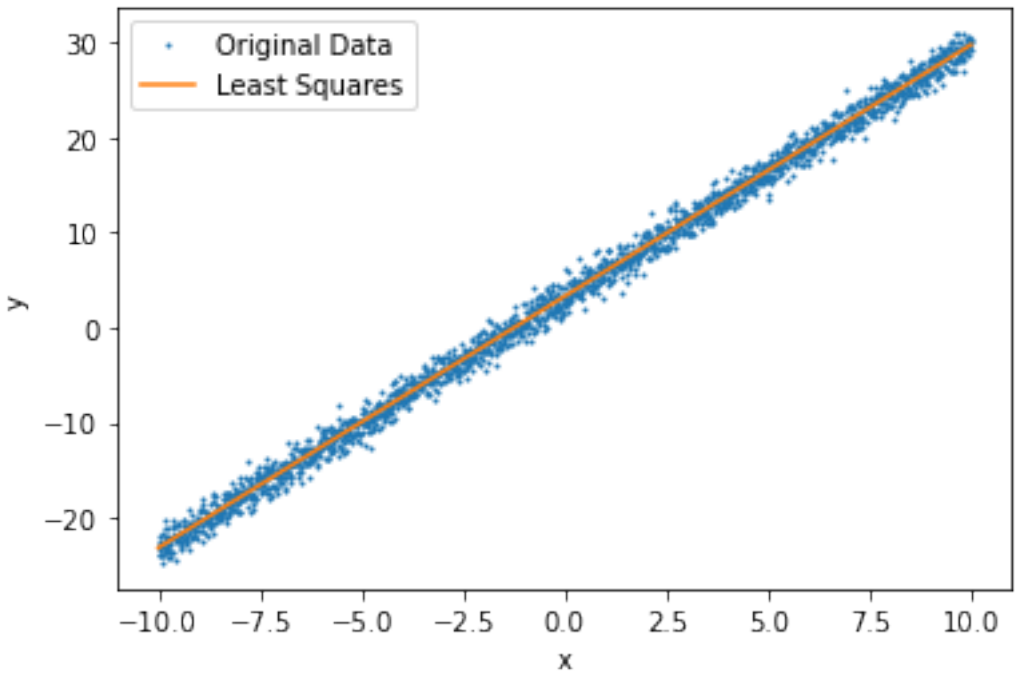
[254]:
```
# (b) using approach 1
data = io.loadmat('prob2_data1.mat')
X = data['x'][0]
Y = data['y'][0]
n = len(X)

m = (sum(X*Y) - 1/n*sum(X)*sum(Y)) / (sum(X**2) - 1/n*sum(X)**2)
```

```
b = (1/n*sum(Y)*sum(X**2) - 1/n*sum(X)*sum(X*Y)) / (sum(X**2) - 1/n*sum(X)**2)
print("m = " + str(m))
print("b = " + str(b))
```

```
m = 2.6450023173267905
b = 3.2778983142176705
```

[263]:
```
# (c) using approach 2
data = io.loadmat('prob2_data1.mat')
X = data['x'][0]
Y = data['y'][0]
n = len(X)

X = X.reshape(n,1)
X = np.concatenate((X, np.ones((n,1))), axis=1)
Y = Y.reshape(n,1)

B = np.matmul(np.linalg.inv(np.matmul(np.transpose(X),X)), np.matmul(np.
 ↪transpose(X),Y))
print("m = " + str(B[0]))
print("b = " + str(B[1]))

x = [-10,10]
y = B[0]*x + B[1]
plt.plot(X[:,0], Y[:,0], 'o', ms=1)
plt.plot(x,y)
plt.legend(['Original Data', 'Least Squares'])
plt.xlabel('x'), plt.ylabel('y')
plt.show()
```

```
m = [2.64500232]
b = [3.27789831]
```



Here is another measurement data (prob2_data2.mat). Please find m and b using the following methods.

(d) Least squares (either approach 1 or approach 2)

[255]:
```
# using approach 1
data = io.loadmat('prob2_data2.mat')
X = data['x'][0]
Y = data['y'][0]
n = len(X)

m = (sum(X*Y) - 1/n*sum(X)*sum(Y)) / (sum(X**2) - 1/n*sum(X)**2)
b = (1/n*sum(Y)*sum(X**2) - 1/n*sum(X)*sum(X*Y)) / (sum(X**2) - 1/n*sum(X)**2)
print("m = " + str(m))
print("b = " + str(b))
```

```
m = 3.3349792943663177
b = 3.7060180763951185
```

(e) Use of RANSAC. You need to have your own RANSAC implementation without using existing functions in MATLAB or Python (must not use ransac or any other relevant functions)

[256]:
```
# load data
data = io.loadmat('prob2_data2.mat')
X = np.array(data['x'][0])
```

```
Y = np.array(data['y'][0])
n = len(X)

# RANSAC parameters
N = 10000 #number of trials
threshold = 1

maxInliers = 0
m = 0
b = 0
for i in range(N):
    # randomly select subset of points
    idx = random.sample(range(n), 2)
    idx.sort()
    Xi = X[idx]
    Yi = Y[idx]

    # generate hypothesis model
    m_test = (Yi[1]-Yi[0])/(Xi[1]-Xi[0])
    b_test = Yi[0]-m_test*Xi[0]
    Y_test = m_test*X+b_test

    # compute distances between data and estimate
    dist = abs(Y_test-Y)

    # select inliers within distance threshold
    numInliers = len(dist[dist<threshold])
    if(numInliers > maxInliers):
        m = m_test
        b = b_test
        maxInliers = numInliers

print("m = " + str(m))
print("b = " + str(b))
```

```
m = 3.350535357164516
b = 3.7536053255551014
```

(f) When do we use either least squares or RANSAC? What are the pros and cons of each technique?

Least squares is a simple and easy to understand data fitting method. This simplicity makes it applicable for a wide variety of uses. However, since least squares considers the entire dataset, this method is very sensitive to the presence of outliers. On the other RANSAC is a more in-depth data fitting method algorithim to fit data which contains outliers. By using random samples to estimate model parameters, RANSAC can avoid undue influence from outliers. However, there is no upper bound on RANSAC's computation time. It also not a general solution, as there the threshold values are problem-specific.

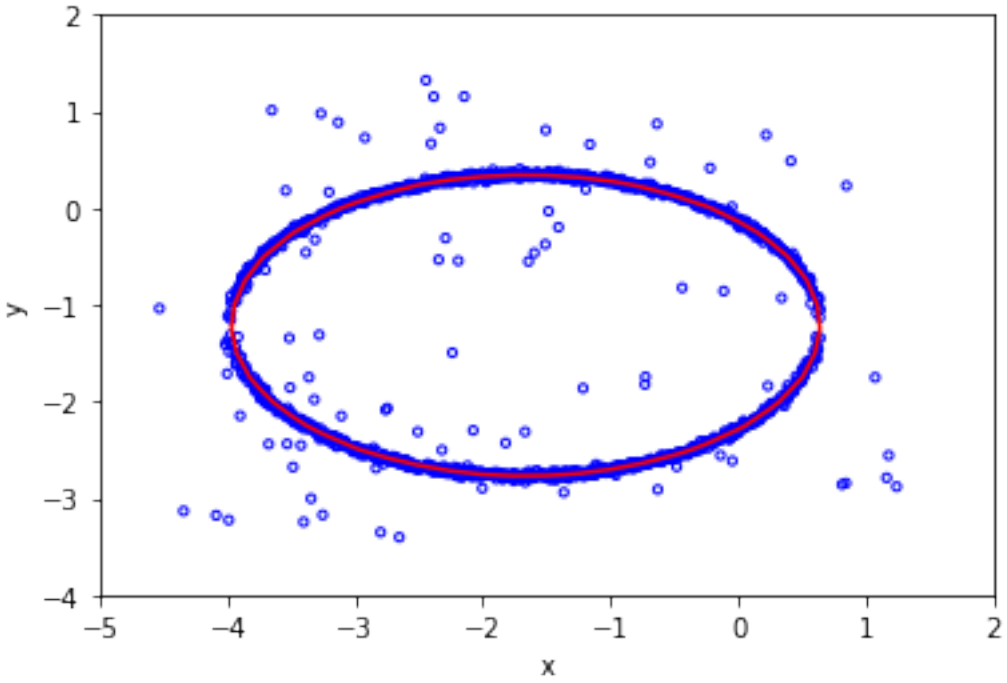
Problem 3: Fitting using RANSAC (20 points)

(a) Fit an ellipse to the given data (prob3_ellipse.mat) using RANSAC

$$ax^2 + cy^2 + dx + ey + f = 0$$

[264]:
```
# load data
data = io.loadmat('prob3_ellipse.mat')
x = np.array(data['x'][0])
y = np.array(data['y'][0])
n = len(x)

# RANSAC parameters
N = 5000 #number of trials
threshold = 0.05

lineCoeff = 0
maxInliers = 0
for i in range(N):
    # randomly select subset of points
    idx = random.sample(range(n), 4)
    xi = x[idx].reshape(4,1)
    yi = y[idx].reshape(4,1)
    coeff_mat = np.column_stack((xi**2, yi**2, xi, yi, np.ones((4,1))))
    coeff = null_space(coeff_mat)

    # generate hypothesis model
    coeff_mat = np.column_stack((xi**2, yi**2, xi, yi, np.ones((4,1))))
    coeff = null_space(coeff_mat)

    # compute distances between data and estimate
    pts = np.column_stack((x**2, y**2, x, y, np.ones((n,1))))
    dist = abs(np.matmul(pts, coeff))

    # select inliers within distance threshold
    numInliers = len(dist[dist<threshold])
    if(numInliers > maxInliers):
        lineCoeff = coeff
        maxInliers = numInliers


a = lineCoeff[0]
c = lineCoeff[1]
d = lineCoeff[2]
e = lineCoeff[3]
f = lineCoeff[4]

#plot ellipse
```

```
x_range = np.arange(-5, 5, 0.25)
y_range = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(x_range,y_range)
F = a*(X**2) + c*(Y**2)
G = d*X + e*Y + f
contour = plt.contour(X, Y, (F+G), [0], colors='red')

# plot data
plt.scatter(x,y, s=10, facecolors='none', edgecolors='b')
plt.xlim([-5,2]), plt.ylim([-4,2])
plt.xlabel('x'), plt.ylabel('y')
plt.show()
```

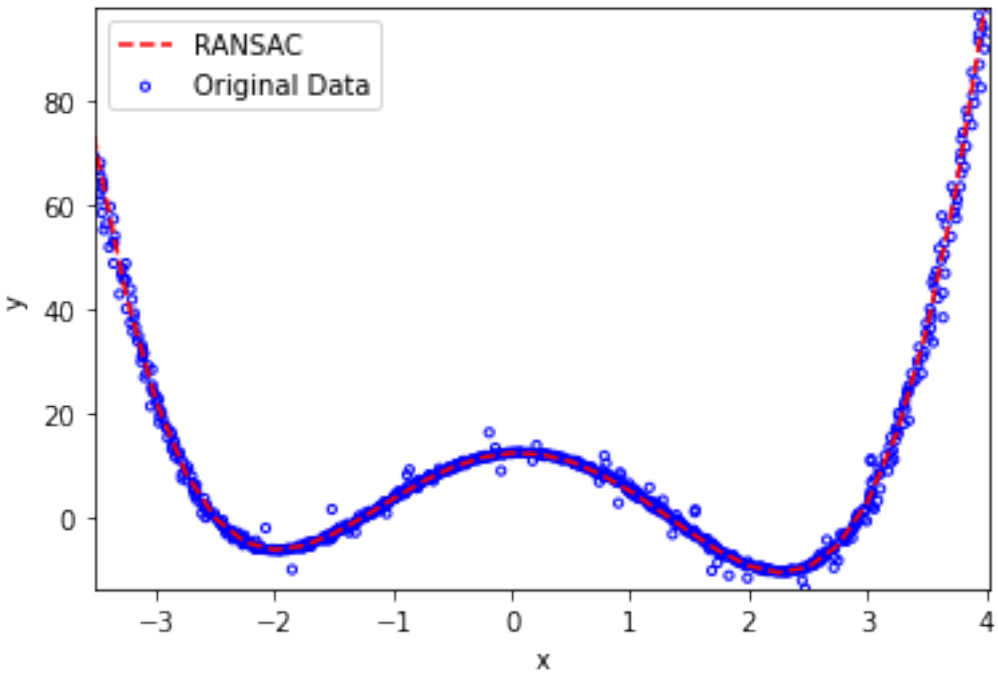


(b) Fit a fourth-degree polynomial to the given data (prob3_polynomial.mat) using RANSAC

$$y = ax^4 + bx^3 + cx^2 + dx + e$$

[265]:
```
# load data
data = io.loadmat('prob3_polynomial.mat')
x = np.array(data['x'][0])
y = np.array(data['y'][0])
n = len(x)

# RANSAC parameters
```

```
N = 1000 #number of trials
threshold = 1

lineCoeff = 0
maxInliers = 0
for i in range(N):
    # randomly select subset of points
    idx = random.sample(range(n), 5)
    xi = x[idx].reshape(5,1)
    yi = y[idx].reshape(5,1)

    # generate hypothesis model
    coeff_mat = np.column_stack((xi**4, xi**3, xi**2, xi, np.ones((5,1))))
    coeff = np.matmul(np.linalg.inv(coeff_mat), yi)
    y_h = coeff[0]*x**4 + coeff[1]*x**3 + coeff[2]*x**2 + coeff[3]*x + coeff[4]

    # compute distances between data and estimate
    dist = abs(y-y_h)

    # select inliers within distance threshold
    numInliers = len(dist[dist<threshold])
    if(numInliers > maxInliers):
        lineCoeff = coeff
        maxInliers = numInliers


a = lineCoeff[0]
b = lineCoeff[1]
c = lineCoeff[2]
d = lineCoeff[3]
e = lineCoeff[4]

# plot data
X = np.arange(-5, 5, 0.25)
Y = a*X**4 + b*X**3 + c*X**2 + d*X + e
plt.scatter(x,y, s=10, facecolors='none', edgecolors='b')
plt.plot(X, Y, 'r--')
plt.xlim([min(x),max(x)])
plt.ylim([min(y), max(y)])
plt.xlabel('x'), plt.ylabel('y')
plt.legend(['RANSAC', 'Original Data'])
plt.show()
```



Problem 4: Improved 3D Planar Measurement Tool (20 points)

You are going to build an improved 3D planar measurement tool. This application is designed to measure a 3D distance on a plane without picking the corners of the booklet in advance. Users will take a photo on a planar surface where a calibrate paper (here, booklet) is placed and simply click two points on the image for measurement. This reduces the step for picking the corners of your calibration paper. Here is a demo video:

(a) Build your own measurement tool and evaluate your measurement using the images provided (see the folder of prob4_img). You may need to estimate homography based on SIFT feature matching. The exact size of the booklet is 24 cm x 31.5 cm and use cover.jpg to solve this problem.

[2]:
```
def ComputeH(old_pts, new_pts):
    x = old_pts[:,0]
    y = old_pts[:,1]
    xp = new_pts[:,0]
    yp = new_pts[:,1]

    A = np.zeros((new_pts.shape[0]*2, 9))
    A[::2,0:2] = old_pts
    A[1::2,3:5] = old_pts
    A[::2,2] = 1
    A[1::2,5] = 1
    A[0::2,6] = -x*xp
```

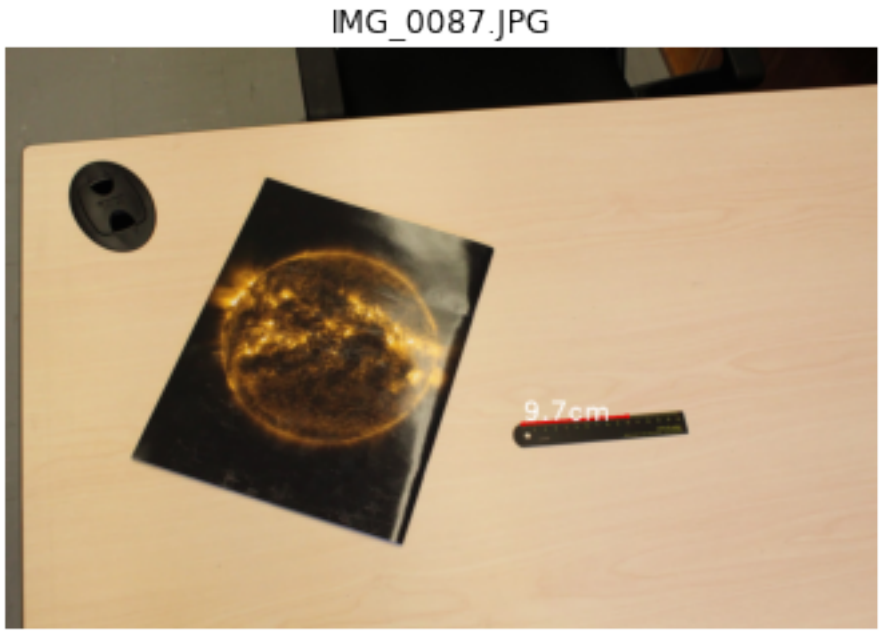
```
    A[0::2,7] = -xp*y
    A[0::2,8] = -xp
    A[1::2,6] = -x*yp
    A[1::2,7] = -yp*y
    A[1::2,8] = -yp

    H = null_space(A)
    H = H[:,0]
    return H.reshape(3,3)
```

[266]:
```
def click_event(event, x, y, flags, params):
    if event == cv2.EVENT_LBUTTONDOWN:
        if len(pts) < 2:
            cv2.circle(img_copy, (x, y), 4, (0, 0, 255), -1)
            pts.append([x, y])
            isClosed = True if len(pts)>=4 else False
            poly_pts = np.array(pts).reshape((-1,1,2))
            cv2.polylines(img_copy, [poly_pts], isClosed, (0, 0, 225), 4)
            cv2.imshow('Select Measurement', img_copy)

# image list
dirImg = 'prob4_img'
imgList = []
for f in os.listdir(dirImg):
    ext = os.path.splitext(f)[1]
    if ext.lower() == '.jpg':
        imgList.append(os.path.join(dirImg,f))

# load calibration image
bookletSize = (24, 31.5)
cover = cv2.imread('cover.jpg')
dim = (int(cover.shape[1]/(cover.shape[0]/1000)), 1000)
cover = cv2.resize(cover, dim, interpolation = cv2.INTER_AREA)
cover_gray = cv2.cvtColor(cover, cv2.COLOR_BGR2GRAY)

for imgPath in imgList:

    # read image
    img = cv2.imread(imgPath)
    dim = (int(img.shape[1]/(img.shape[0]/1000)), 1000)
    img = cv2.resize(img, dim, interpolation = cv2.INTER_AREA)

    # keypoint detection
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    sift = cv2.SIFT_create()
    kp_cover, des_cover = sift.detectAndCompute(cover_gray,None)
    kp_img, des_img = sift.detectAndCompute(img_gray,None)
```

```
    # feature matching
    bf = cv2.BFMatcher(cv2.NORM_L1, crossCheck=True)
    matches = bf.match(des_cover, des_img)
    numMatches = len(matches)

    # plot matches
    # matches = sorted(matches, key = lambda x:x.distance)
    # img_match = cv2.drawMatches(cover_gray, kp_cover, img_gray, kp_img,␣
↪matches[:50], img_gray, flags=2)
    # plt.imshow(img_match)
    # plt.show()

    pts_cover = []
    pts_img = []

    # determined pixel coordinates of matches
    for match in matches:
        # matching keypoints for each of the images
        cover_idx = match.queryIdx
        img_idx = match.trainIdx

        # coordinates
        (x1, y1) = kp_cover[cover_idx].pt
        (x2, y2) = kp_img[img_idx].pt
        pts_cover.append([x1, y1])
        pts_img.append([x2, y2])

    # convert to homogenous coords
    pts_cover = [[pt[0], pt[1], 1] for pt in pts_cover]
    pts_img = [[pt[0], pt[1], 1] for pt in pts_img]
    pts_cover = np.array(pts_cover)
    pts_img = np.array(pts_img)

    # compute homography matrix using RANSAC
    N = 1000 # number of trials
    threshold = 2
    maxInliers = 0

    for i in range(N):
        # randomly select subset of points
        idx = random.sample(range(numMatches), 4)
        pts1 = pts_cover[idx]
        pts2 = pts_img[idx]

        # generate hypothesis model
        H_test = ComputeH(pts1[:,0:2], pts2[:,0:2])
```

```
        # compute distances between data and estimate
        pts_img_h = np.dot(H_test, np.transpose(pts_cover))
        pts_img_h = np.transpose(pts_img_h) # nx3
        pts_img_h = np.column_stack((pts_img_h[:,0]/pts_img_h[:,2], pts_img_h[:
↪,1]/pts_img_h[:,2])) # in cartesian

        dist = [np.linalg.norm(pts_img[j,0:2]-pts_img_h[j,:]) for j in␣
↪range(numMatches)]
        dist = np.array(dist)

        # select inliers within distance threshold
        numInliers = len(dist[dist<threshold])
        if(numInliers > maxInliers):
            # print(numInliers)
            H = H_test
            maxInliers = numInliers

    # select measurement
    pts = []
    img_copy = img.copy()
    cv2.namedWindow('Select Measurement', cv2.WINDOW_NORMAL)
    cv2.imshow('Select Measurement', img_copy)
    cv2.setMouseCallback('Select Measurement', click_event)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

    # convert measurement points to cover plane
    pts = [[pt[0], pt[1], 1] for pt in pts] # in HC
    pts = np.array(pts)
    pts_H = np.matmul(np.linalg.inv(H), np.transpose(pts)) # in cover plane
    pts_H = np.transpose(pts_H)
    pts_H = np.column_stack((pts_H[:,0]/pts_H[:,2], pts_H[:,1]/pts_H[:,2])) #␣
↪in cartesian

    # determine length of measurement
    scale = bookletSize[1]/cover.shape[0]
    length = np.linalg.norm(pts_H[0,:]-pts_H[1,:])
    length = length*scale

    # plot results
    fileName = imgPath.split("\\")[-1]
    x_text = np.min(pts[:,0])
    y_text = np.mean(pts[:,1])
    loc = (int(x_text), int(y_text))
    cv2.putText(img_copy, str(round(length, 2))+'cm', loc, cv2.
↪FONT_HERSHEY_DUPLEX, 1.5, (255, 255, 255), 2)
```

```
    plt.figure(figsize = (6,4))
    plt.title(fileName)
    plt.imshow(img_copy[:,:,::-1])
    plt.axis('off')
    plt.show()
```

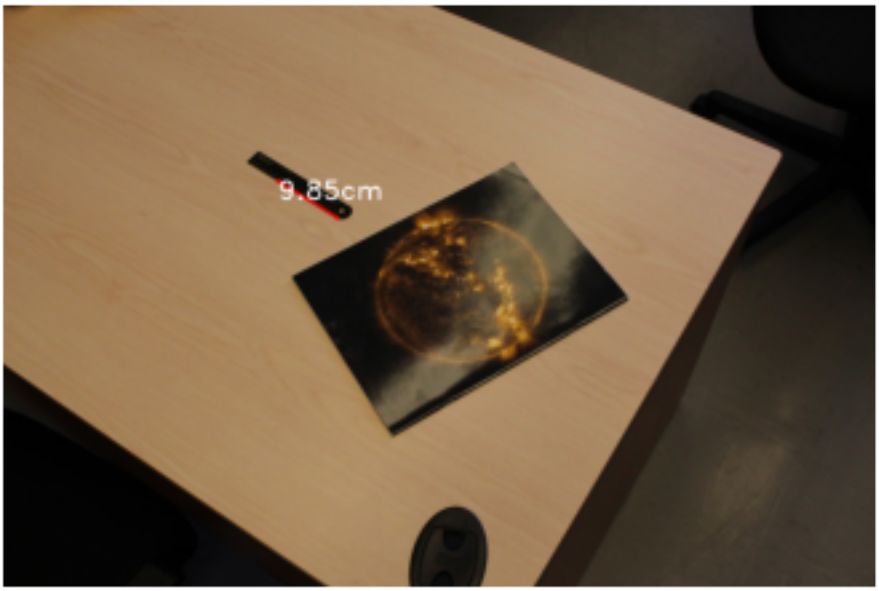


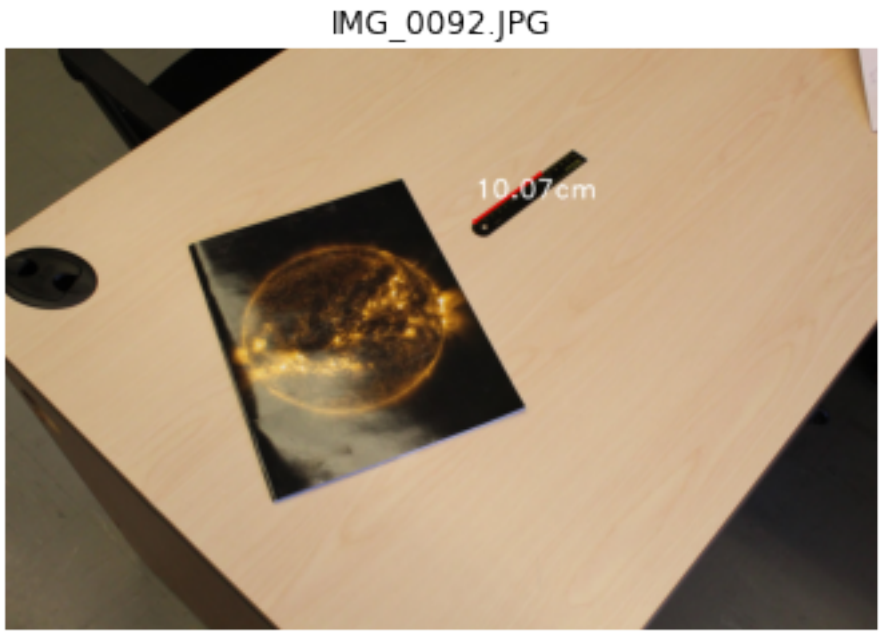
IMG_0088.JPG

IMG_0089.JPG

IMG_0090.JPG

IMG_0091.JPG





(b) Prepare your own calibration paper and take photos similar to the ones in (a). Then, evaluate your tool.

Using a Christmas card (11x16cm) as the calibration paper:

[251]:
```
def click_event(event, x, y, flags, params):
    if event == cv2.EVENT_LBUTTONDOWN:
        if len(pts) < 2:
            cv2.circle(img_copy, (x, y), 4, (0, 0, 255), -1)
            pts.append([x, y])
            isClosed = True if len(pts)>=4 else False
            poly_pts = np.array(pts).reshape((-1,1,2))
            cv2.polylines(img_copy, [poly_pts], isClosed, (0, 0, 225), 4)
            cv2.imshow('Select Measurement', img_copy)

# image list
dirImg = 'prob4_img2'
imgList = []
for f in os.listdir(dirImg):
    ext = os.path.splitext(f)[1]
    if ext.lower() == '.jpg':
        imgList.append(os.path.join(dirImg,f))

# load calibration image
bookletSize = (11, 16)
cover = cv2.imread('cover2.jpg')
dim = (int(cover.shape[1]/(cover.shape[0]/1000)), 1000)
cover = cv2.resize(cover, dim, interpolation = cv2.INTER_AREA)
cover_gray = cv2.cvtColor(cover, cv2.COLOR_BGR2GRAY)

for imgPath in imgList:

    # read image
    img = cv2.imread(imgPath)
    dim = (int(img.shape[1]/(img.shape[0]/1000)), 1000)
    img = cv2.resize(img, dim, interpolation = cv2.INTER_AREA)

    # keypoint detection
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    sift = cv2.SIFT_create()
    kp_cover, des_cover = sift.detectAndCompute(cover_gray,None)
    kp_img, des_img = sift.detectAndCompute(img_gray,None)

    # feature matching
    bf = cv2.BFMatcher(cv2.NORM_L1, crossCheck=True)
    matches = bf.match(des_cover, des_img)
    numMatches = len(matches)

    # plot matches
    # matches = sorted(matches, key = lambda x:x.distance)
    # img_match = cv2.drawMatches(cover_gray, kp_cover, img_gray, kp_img,␣
↪matches[:50], img_gray, flags=2)
```

```
    # plt.imshow(img_match)
    # plt.show()

    pts_cover = []
    pts_img = []

    # determined pixel coordinates of matches
    for match in matches:
        # matching keypoints for each of the images
        cover_idx = match.queryIdx
        img_idx = match.trainIdx

        # coordinates
        (x1, y1) = kp_cover[cover_idx].pt
        (x2, y2) = kp_img[img_idx].pt
        pts_cover.append([x1, y1])
        pts_img.append([x2, y2])

    # convert to homogenous coords
    pts_cover = [[pt[0], pt[1], 1] for pt in pts_cover]
    pts_img = [[pt[0], pt[1], 1] for pt in pts_img]
    pts_cover = np.array(pts_cover)
    pts_img = np.array(pts_img)

    # compute homography matrix using RANSAC
    N = 1000 # number of trials
    threshold = 2
    maxInliers = 0

    for i in range(N):
        # randomly select subset of points
        idx = random.sample(range(numMatches), 4)
        pts1 = pts_cover[idx]
        pts2 = pts_img[idx]

        # generate hypothesis model
        H_test = ComputeH(pts1[:,0:2], pts2[:,0:2])

        # compute distances between data and estimate
        pts_img_h = np.dot(H_test, np.transpose(pts_cover))
        pts_img_h = np.transpose(pts_img_h) # nx3
        pts_img_h = np.column_stack((pts_img_h[:,0]/pts_img_h[:,2], pts_img_h[:
↪,1]/pts_img_h[:,2])) # in cartesian

        dist = [np.linalg.norm(pts_img[j,0:2]-pts_img_h[j,:]) for j in␣
↪range(numMatches)]
        dist = np.array(dist)
```

```
        # select inliers within distance threshold
        numInliers = len(dist[dist<threshold])
        if(numInliers > maxInliers):
            # print(numInliers)
            H = H_test
            maxInliers = numInliers

    # select measurement
    pts = []
    img_copy = img.copy()
    cv2.namedWindow('Select Measurement', cv2.WINDOW_NORMAL)
    cv2.imshow('Select Measurement', img_copy)
    cv2.setMouseCallback('Select Measurement', click_event)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

    # convert measurement points to cover plane
    pts = [[pt[0], pt[1], 1] for pt in pts] # in HC
    pts = np.array(pts)
    pts_H = np.matmul(np.linalg.inv(H), np.transpose(pts)) # in cover plane
    pts_H = np.transpose(pts_H)
    pts_H = np.column_stack((pts_H[:,0]/pts_H[:,2], pts_H[:,1]/pts_H[:,2])) #␣
↪in cartesian

    # determine length of measurement
    scale = bookletSize[1]/cover.shape[0]
    length = np.linalg.norm(pts_H[0,:]-pts_H[1,:])
    length = length*scale

    # plot results
    fileName = imgPath.split("\\")[-1]
    # fileName = fileName[-1]
    x_text = np.min(pts[:,0])
    y_text = np.mean(pts[:,1])
    loc = (int(x_text), int(y_text))
    cv2.putText(img_copy, str(round(length, 2))+'cm', loc, cv2.
↪FONT_HERSHEY_DUPLEX, 1.5, (255, 255, 255), 2)

    plt.figure(figsize = (6,4))
    plt.imshow(img_copy[:,:,::-1])
    plt.axis('off')
    plt.show()
```


9.61cm


10.12cm


10.36cm


10.17cm

(c) Compare measurements using this new tool and the one developed in Task 3. The measurements from the five images shown above were compared with those obtained from the homography measurement tool developed for Task 5.

Image	Task 7 Measurement (cm)	Task 5 Measurement (cm)
1	9.61	10.00
2	10.12	10.18
3	10.36	10.08
4	10.17	10.06
5	10.54	10.00

The measurements from Task 5 are clearly more accurate. This is because this tool involved the user manually selecting the four corners of the calibration paper to determine the homography matrix, which is likely more accurate compared to using feature matching and RANSAC to estimate the homography.

Problem 5: Book Classification using SIFT (20 points)

You are going to categorize books on images. Here are the input images and expected outcomes.

Your code needs to automatically compute the outlines (boundary) of each book and its identity (book name). There are 17 images (see the folder of prob5_img) and in each image, four books are placed on a desk. Your code should not fail to identify a book or estimate its boundary no more than 5 books among all books (85 = 17 images

x 5 books). Note that you should not do hard cording. Note that this problem is not asking you to use edge detection!

Hint: You can reuse the code that you are developing in Problem 4. Like cover.jpg, you need to prepare cover images and name them. Note this problem cannot be solved using Hough transform.
Hint: You can download the book covers from the web.

[42]:
```
# image list
dirImg = 'prob5_img'
imgList = []
for f in os.listdir(dirImg):
    ext = os.path.splitext(f)[1]
    if ext.lower() == '.jpg':
        imgList.append(os.path.join(dirImg,f))

# read cover images
dirCover = 'prob5_img/covers'
covers = []
coverNames = []
for f in os.listdir(dirCover):
    ext = os.path.splitext(f)[1]
    if ext.lower() == '.jpg':
        cover = cv2.imread(os.path.join(dirCover,f))
        dim = (int(cover.shape[1]/(cover.shape[0]/1000)), 1000)
        cover = cv2.resize(cover, dim, interpolation = cv2.INTER_AREA)
        covers.append(cover)
        coverNames.append(f)

# cover keypoint detection
kp_covers = []
des_covers = []
sift = cv2.SIFT_create()
for cover in covers:
    cover_gray = cv2.cvtColor(cover, cv2.COLOR_BGR2GRAY)
    kp_cover, des_cover = sift.detectAndCompute(cover_gray,None)
    kp_covers.append(kp_cover)
    des_covers.append(des_cover)

for imgPath in imgList:
    # read image
    img = cv2.imread(imgPath)
    dim = (int(img.shape[1]/(img.shape[0]/1000)), 1000)
    img = cv2.resize(img, dim, interpolation = cv2.INTER_AREA)
    img_out = img.copy()

    # keypoint detection
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

```
    kp_img, des_img = sift.detectAndCompute(img_gray,None)

    # feature matching for each cover
    bf = cv2.BFMatcher(cv2.NORM_L2)
    for i in range(len(covers)):
        matches = bf.knnMatch(des_covers[i], des_img,k=2)

        # Lowe's ratio test to filter good matches
        good_matches = []
        for m, n in matches:
            if m.distance < 0.75 * n.distance:
                good_matches.append(m)

        numMatches = len(good_matches)

        # plot matches
        # img_match = cv2.drawMatches(covers[i], kp_covers[i], img_gray,␣
↪kp_img, matches[:50], img_gray, flags=2)
        # plt.imshow(img_match)
        # plt.show()

        pts_cover = []
        pts_img = []
        # determined pixel coordinates of matches
        for match in good_matches:
            # matching keypoints for each of the images
            cover_idx = match.queryIdx
            img_idx = match.trainIdx

            # coordinates
            (x1, y1) = kp_covers[i][cover_idx].pt
            (x2, y2) = kp_img[img_idx].pt
            pts_cover.append([x1, y1])
            pts_img.append([x2, y2])

        # convert to homogenous coords
        pts_cover = [[pt[0], pt[1], 1] for pt in pts_cover]
        pts_img = [[pt[0], pt[1], 1] for pt in pts_img]
        pts_cover = np.array(pts_cover)
        pts_img = np.array(pts_img)

        # compute homography matrix using RANSAC
        N = 1000 # number of trials
        threshold = 2
        maxInliers = 0
        for n in range(N):
            # randomly select subset of points
```

```
            idx = random.sample(range(numMatches), 4)
            pts1 = pts_cover[idx]
            pts2 = pts_img[idx]

            # generate hypothesis model
            H_test = ComputeH(pts1[:,0:2], pts2[:,0:2])

            # compute distances between data and estimate
            pts_img_h = np.dot(H_test, np.transpose(pts_cover))
            pts_img_h = np.transpose(pts_img_h) # nx3
            pts_img_h = np.column_stack((pts_img_h[:,0]/pts_img_h[:,2],␣
↪pts_img_h[:,1]/pts_img_h[:,2])) # in cartesian

            dist = [np.linalg.norm(pts_img[j,0:2]-pts_img_h[j,:]) for j in␣
↪range(numMatches)]
            dist = np.array(dist)

            # select inliers within distance threshold
            numInliers = len(dist[dist<threshold])
            if(numInliers > maxInliers):
                # print(numInliers)
                H = H_test
                maxInliers = numInliers

        # determine cover corner points
        size = covers[i].shape
        corner_cover = np.
↪array([[0,0,1],[size[1],0,1],[size[1],size[0],1],[0,size[0],1]]) #in HC

        # convert to points in image plane
        corner_img = np.matmul(H, np.transpose(corner_cover))
        corner_img = np.column_stack((corner_img[0,:]/corner_img[2,:],␣
↪corner_img[1,:]/corner_img[2,:])) # cartesian

        # draw on image
        pts = corner_img.astype(int).reshape((-1, 1, 2))
        cv2.polylines(img_out, [pts], True, (0, 0, 225), 10)
        x_text = np.min(corner_img[:,0])
        y_text = np.mean(corner_img[:,1])
        loc = (int(x_text), int(y_text))
        cv2.putText(img_out, coverNames[i], loc, cv2.FONT_HERSHEY_SIMPLEX, 1,␣
↪(255, 255, 255), 2)

    # plot result
    fig, axs = plt.subplots(1,2, figsize=(9,6))
    axs[0].imshow(img[:,:,::-1])
    axs[0].axis('off'), axs[0].set_title('Original Image')
```

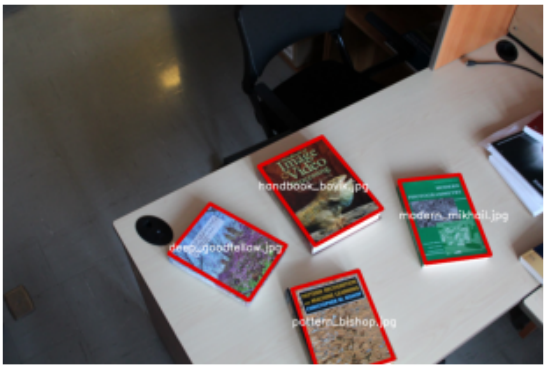
```
    axs[1].imshow(img_out[:,:,::-1])
    axs[1].axis('off'), axs[1].set_title('Result')
    fig.tight_layout()
    plt.show()
```

Original Image

Result

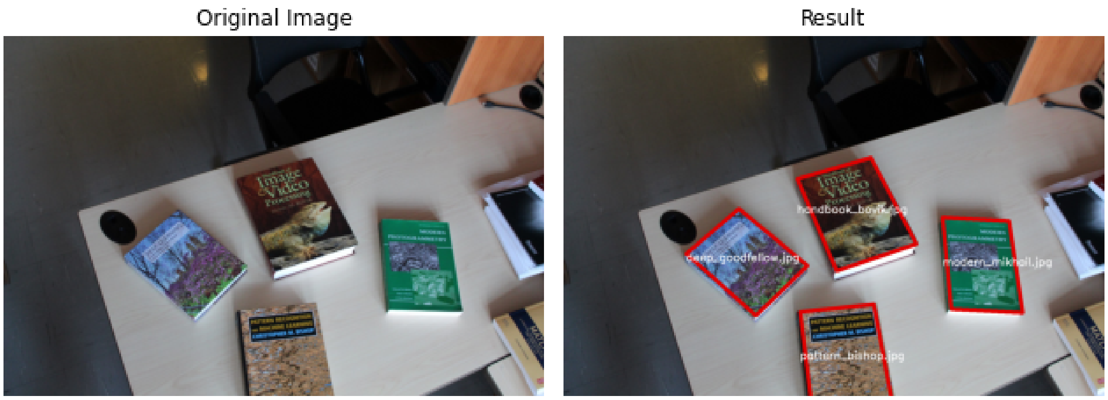
Original Image

Result

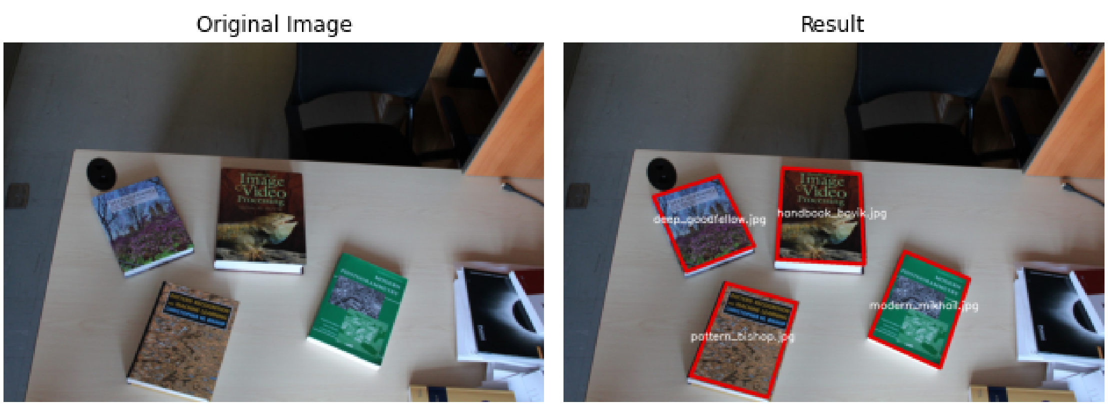


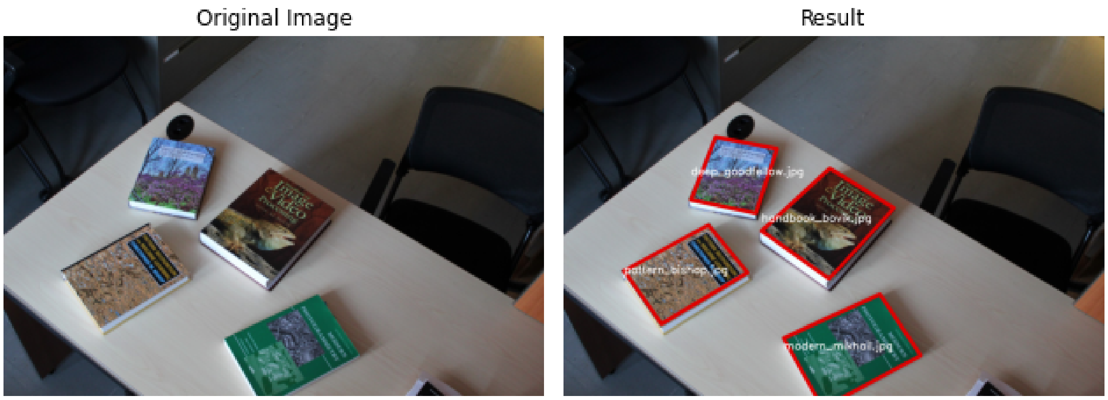
Original Image

Result

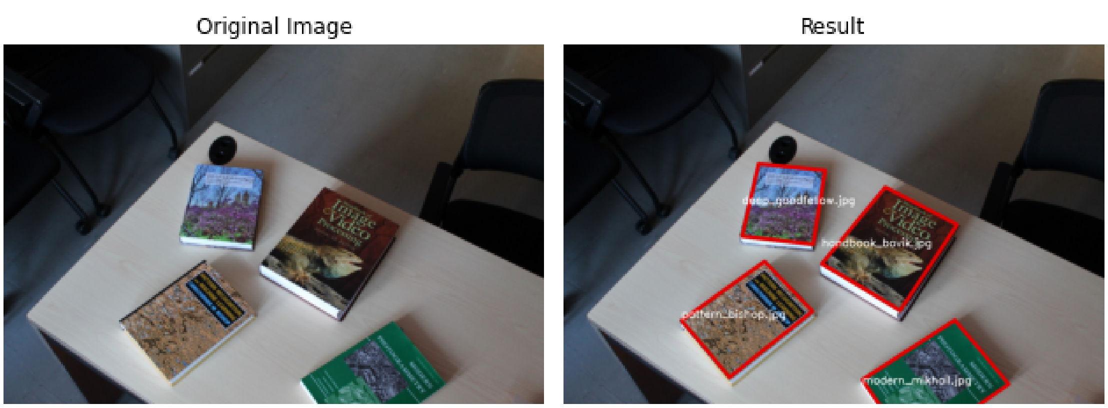

Original Image
Result
handbook_bovik.jpg
deep_goodfellow.jpg
modern_mikhail.jpg
pattern_bishop.jpg


Original Image
Result
handbook_bovik.jpg
deep_goodfellow.jpg
modern_mikhail.jpg
pattern_bishop.jpg


Original Image
Result
deep_goodfellow.jpg
handbook_bovik.jpg
modern_mikhail.jpg
pattern_bishop.jpg


Original Image
Result
deep_goodfellow.jpg
handbook_bovik.jpg
pattern_bishop.jpg
modern_mikhail.jpg


Original Image
Result
deep_goodfellow.jpg
handbook_bovik.jpg
pattern_bishop.jpg
modern_mikhail.jpg


Original Image
Result
deep_goodfellow.jpg
handbook_bovik.jpg
pattern_bishop.jpg
modern_mikhail.jpg

Original Image

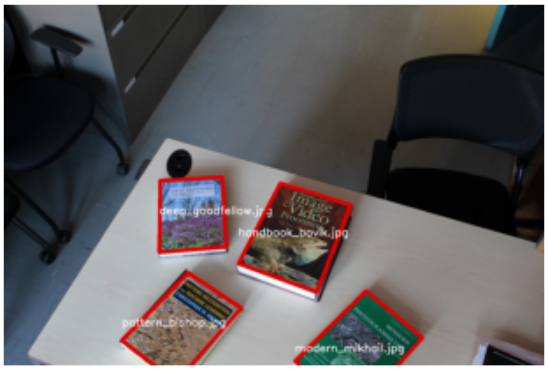
Result

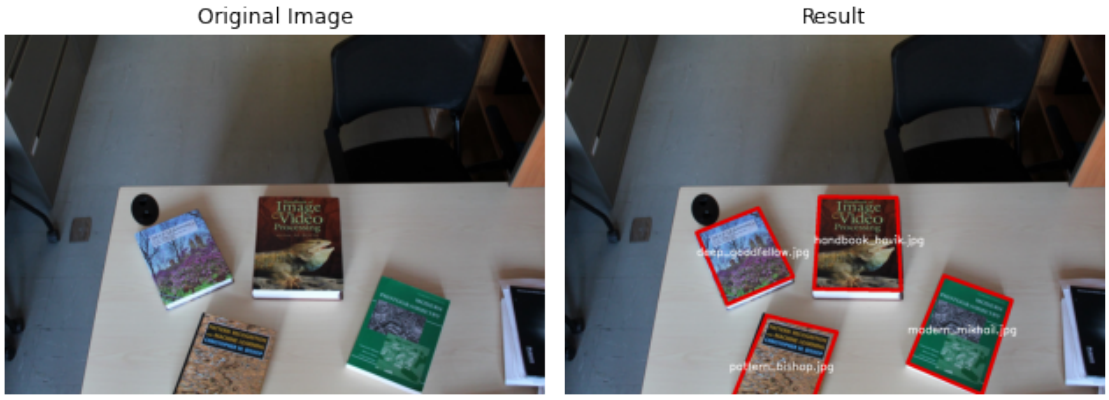
Original Image

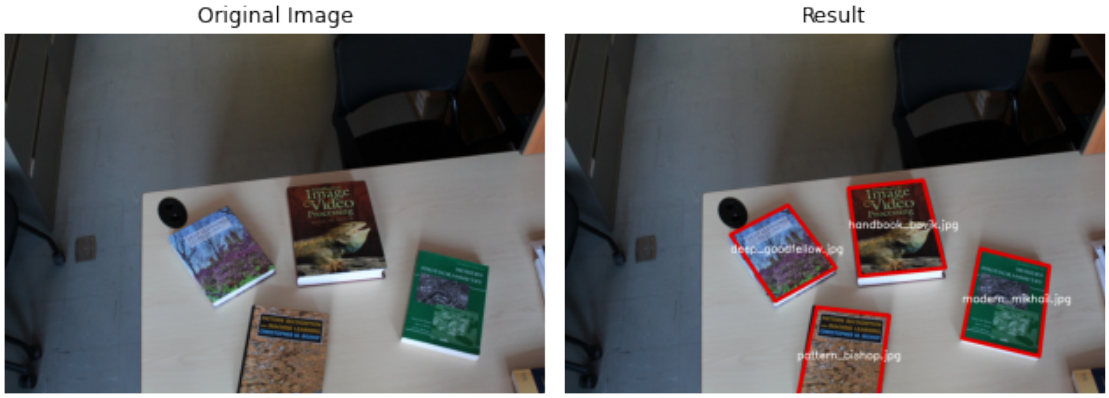
Result

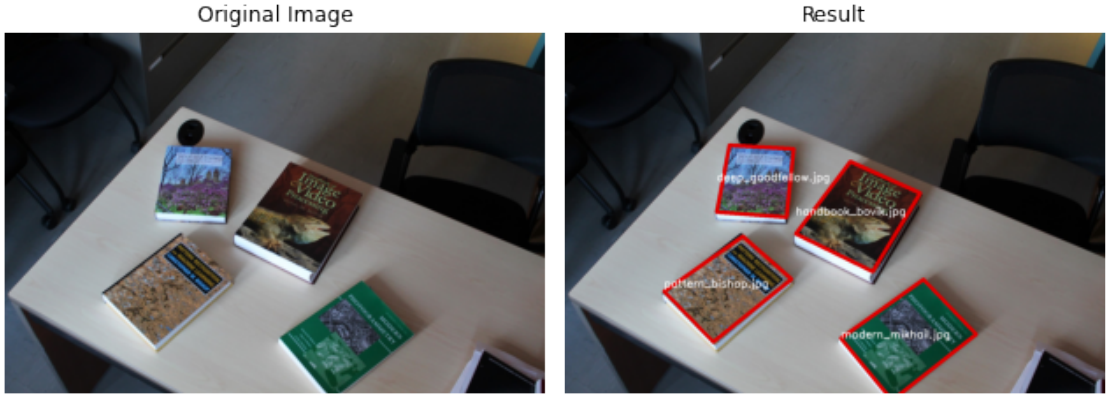
Original Image

Result









Problem 6: Image Stitching (30 points)

First, go to an image stitching tutorial and thoroughly review the code. You have to fully understand its process.

Second, you are going to update this code. Your code must satisfy the following conditions:
- Do not use imageDatastore and related functions that call this object in MATLAB. - Use SIFT feature extractor and descriptor rather than SURF. You can use vl_sift for MATLAB and opencv for Python. - Do not use estimateGeometricTransform and replace it using your own homography estimator that you may developed in problem 4. You need to use RANSAC for robust matching.

Third, test your code using the building dataset (the image set used in the tutorial) in MATLAB. For Python users, you can find those images prob6_building_img.zip. Fourth, take your own image set (more than 3 images) from a scenic place and stitch them to create a nice panorama image. You should not download images on the web or use my images and should not share your images with your colleagues. Please take your images using your cameras. If your resulting panorama is not good quality, you need to explain what factors make dropping its quality.

[4]:
```
def combineImages(warp_img, base_img, corners):
    mask = np.zeros(base_img.shape, dtype=np.uint8)
    roi_corners = np.int32(corners)
    cv2.fillConvexPoly(mask, roi_corners, (255, 255, 255))
    mask = cv2.bitwise_not(mask)
    masked_img = cv2.bitwise_and(base_img, mask)
    output = cv2.bitwise_or(warp, masked_img)
    return output
```

[40]:
```
# load images
dirImg = 'prob6_building_img'
imgList = []
for f in os.listdir(dirImg):
    ext = os.path.splitext(f)[1]
    if ext.lower() == '.jpg':
        img = cv2.imread(os.path.join(dirImg,f))
        imgList.append(img)
numImages = len(imgList)

# initialize features for I[0]
img_gray = cv2.cvtColor(imgList[0], cv2.COLOR_BGR2GRAY)
sift = cv2.SIFT_create()
kp_img, des_img = sift.detectAndCompute(img_gray,None)

# initialize variable to hold image sizes
imgSize = np.zeros((numImages, 2))
imgSize[0,:] = img_gray.shape

# initialize tranfsormation matrices array
H_arr = np.zeros((3,3,numImages))
H_arr[:,:,0] = np.identity(3)

# keypoint matching parameters
```

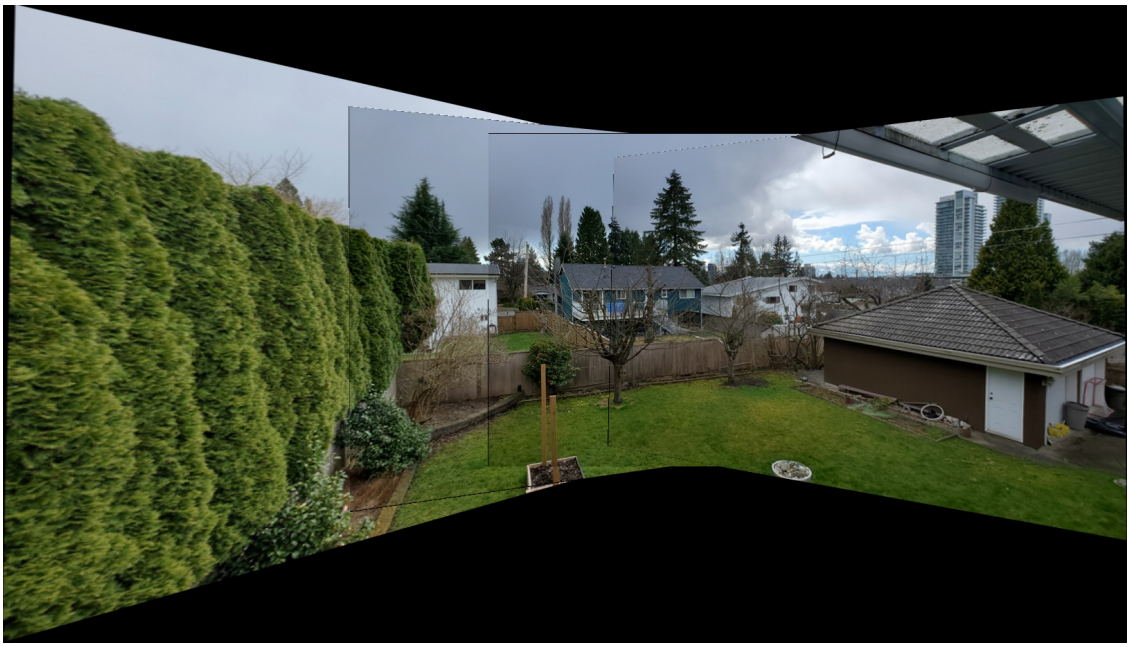
```
bf = cv2.BFMatcher(cv2.NORM_L2)

for n in range(1, numImages):
    # Store points and features for I(n-1)
    kp_prev = kp_img
    des_prev = des_img

    # feature deteciton for I(n)
    img_gray = cv2.cvtColor(imgList[n], cv2.COLOR_BGR2GRAY)
    kp_img, des_img = sift.detectAndCompute(img_gray,None)
    imgSize[n,:] = img_gray.shape

    # keypoint matching between I(n) and I(n-1)
    matches = bf.knnMatch(des_prev, des_img, k=2)

    # using Lowe's ratio test to filter good matches
    good_matches = []
    for a, b in matches:
        if a.distance < 0.6 * b.distance:
            good_matches.append(a)

    numMatches = len(good_matches)

    # determined pixel coordinates of matches
    pts_prev = []
    pts_img = []
    for match in good_matches:
        prev_idx = match.queryIdx
        img_idx = match.trainIdx

        (x1, y1) = kp_prev[prev_idx].pt
        (x2, y2) = kp_img[img_idx].pt
        pts_prev.append([x1, y1])
        pts_img.append([x2, y2])

    # plot matches
    # good_matches = sorted(good_matches, key = lambda x:x.distance)
    # img_match = cv2.drawMatches(imgList[n-1], kp_prev, img_gray, kp_img,␣
↪good_matches[:100], img_gray, flags=2)
    # plt.figure(figsize=(9,6))
    # plt.imshow(img_match)
    # plt.show()

    # convert to homogenous coords
    pts_prev = [[pt[0], pt[1], 1] for pt in pts_prev]
    pts_img = [[pt[0], pt[1], 1] for pt in pts_img]
    pts_prev = np.array(pts_prev)
```

```
    pts_img = np.array(pts_img)

    # compute homography matrix using RANSAC
    N = 2500 # number of trials
    threshold = 1
    maxInliers = 0
    for i in range(N):
        # randomly select subset of points
        idx = random.sample(range(numMatches), 4)
        pts2 = pts_prev[idx]
        pts1 = pts_img[idx]

        # generate hypothesis model
        H_test = ComputeH(pts1[:,0:2], pts2[:,0:2])

        # compute distances between data and estimate
        pts_prev_h = np.dot(H_test, np.transpose(pts_img))
        pts_prev_h = np.column_stack((pts_prev_h[0,:]/pts_prev_h[2,:],␣
↪pts_prev_h[1,:]/pts_prev_h[2,:])) # in cartesian

        dist = [np.linalg.norm(pts_prev[j,0:2]-pts_prev_h[j,:]) for j in␣
↪range(numMatches)]
        dist = np.array(dist)

        # select inliers within distance threshold
        numInliers = len(dist[dist<threshold])
        if(numInliers > maxInliers):
            H = H_test
            maxInliers = numInliers

    H_arr[:,:,n] = np.matmul(H, H_arr[:,:,n-1])

# transform H-arr such that centre image is the least distorted
# output limits for each transform
xlim = []
ylim = []
for i in range(numImages):
    size = imgSize[i]
    corner = np.
↪array([[0,0,1],[size[1],0,1],[size[1],size[0],1],[0,size[0],1]]) #in HC
    pts = np.matmul(H_arr[:,:,i], np.transpose(corner)) # in panorama plane
    pts = np.column_stack((pts[0,:]/pts[2,:], pts[1,:]/pts[2,:])) # in cartesian
    xlim.append([min(pts[:,0]), max(pts[:,0])])
    ylim.append([min(pts[:,1]), max(pts[:,1])])

# find horizontal (x) center of each image (only x means only horz panorama can␣
↪be used)
```

```
# to determine center image
avg_xlim = [np.mean(lim) for lim in xlim]
idx = np.argsort(avg_xlim)
center_idx = idx[numImages//2]

# apply center image's inverse transform to other images
H_inv = np.linalg.inv(H_arr[:,:,center_idx])
for i in range(numImages):
    H_arr[:,:,i] = np.matmul(H_arr[:,:,i], H_inv)

# compute size of panorama
xlim = []
ylim = []
corners = []
for i in range(numImages):
    size = imgSize[i]
    corner = np.
 ↪array([[0,0,1],[size[1],0,1],[size[1],size[0],1],[0,size[0],1]]) #in HC
    pts = np.matmul(H_arr[:,:,i], np.transpose(corner)) # in panorama plane
    pts = np.column_stack((pts[0,:]/pts[2,:], pts[1,:]/pts[2,:])) # in cartesian
    xlim.append([min(pts[:,0]), max(pts[:,0])])
    ylim.append([min(pts[:,1]), max(pts[:,1])])
    corners.append(pts)

# find the minimum and maximum output limits
maxImgSize = np.amax(imgSize, axis=0)
x_min = min([1, np.amin(xlim)])
x_max = max([maxImgSize[1], np.amax(xlim)])
y_min = min([1, np.amin(ylim)])
y_max = max([maxImgSize[0], np.amax(ylim)])

# Width and height of panorama
width  = int(round(x_max - x_min))
height = int(round(y_max - y_min))

# reapply homography to translate images so resulting image isn't cut off
for i in range(numImages):
    size = imgSize[i]
    pts1 = corners[i]
    pts2 = np.zeros((4,2))
    pts2[:,0] = pts1[:,0] - x_min
    pts2[:,1] = pts1[:,1] - y_min
    H = ComputeH(pts1, np.int32(pts2))
    H_arr[:,:,i] = np.matmul(H, H_arr[:,:,i])

    corner = np.
 ↪array([[0,0,1],[size[1],0,1],[size[1],size[0],1],[0,size[0],1]]) #in HC
```

```
    pts = np.matmul(H_arr[:,:,i], np.transpose(corner)) # in panorama plane
    pts = np.column_stack((pts[0,:]/pts[2,:], pts[1,:]/pts[2,:])) # in cartesian
    corners[i] = pts

# create the panorama
panorama = np.zeros((height,width,3), np.uint8)
for i in range(numImages):
    warp = cv2.warpPerspective(imgList[i], H_arr[:,:,i], (width, height))
    panorama = combineImages(warp, panorama, corners[i])

# plot result
plt.figure(figsize=(width/200, height/200))
plt.imshow(panorama[:,:,::-1])
plt.axis('off')
plt.show()
```

Using my own images, I obtained the following panorama.



The quality of the panorama is quite good, but there are black areas around the edges of the individual images because I created my own subpar image stiching script as opposed to using a package such as vision.AlphaBlender in MATLAB since I couldn't find an equivalent Python alternative.

Problem 7: Card Number Identification and Replacement (30 points)

You are going to recognize the card numbers and replace A with A . Here are the input images and expected outcomes.
Your code needs to automatically compute a number of each card using the given card image in (prob7_img/card). The sequence of the card numbers on the given images (prob7_img/img) are marked on the top of the corresponding images (see the resulting image above). All cards are the same size. Next, I like better than . Please replace (or overlay) the A card with the A card. The quality of the overlay is similar to the ones in the resulting images above.

You should not do hard cording to find out the sequence and card overlay. Note that this problem is NOT edge detection or object recognition. You can solve it using the code that you made for the previous problems or tasks.

[22]:
```
# load images
dirImg = 'prob7_img\img'
imgList = []
for f in os.listdir(dirImg):
    ext = os.path.splitext(f)[1]
    if ext.lower() == '.jpg':
        img = cv2.imread(os.path.join(dirImg,f))
        imgList.append(img)
numImages = len(imgList)
```

```
# load cards
dirCard = 'prob7_img\card'
cards = ['card_a.jpg', 'card_j.jpg', 'card_k.jpg', 'card_q.jpg']
card_ah = cv2.imread(os.path.join(dirCard,'card_ah.jpg'))
cardList = []
kp_cards = []
des_cards = []
for card in cards:
    img = cv2.imread(os.path.join(dirCard,card))
    cardList.append(img)

    # feature detection for cards
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    kp_img, des_img = sift.detectAndCompute(img_gray,None)
    kp_cards.append(kp_img)
    des_cards.append(des_img)

bf = cv2.BFMatcher(cv2.NORM_L2)
for img in imgList:

    # feature detection
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    kp_img, des_img = sift.detectAndCompute(img_gray,None)

    H_arr = np.zeros((3,3,len(cardList)))
    # match cards to image and compute homography
    for n in range(len(cardList)):

        # keypoint matching
        matches = bf.knnMatch(des_cards[n], des_img, k=2)

        # using Lowe's ratio test to filter good matches
        good_matches = []
        for a, b in matches:
            if a.distance < 0.6 * b.distance:
                good_matches.append(a)

        numMatches = len(good_matches)

        # determine pixel coordinates of matches
        pts_card = []
        pts_img = []
        for match in good_matches:
            query_idx = match.queryIdx
            train_idx = match.trainIdx

            (x1, y1) = kp_cards[n][query_idx].pt
```

```
            (x2, y2) = kp_img[train_idx].pt
            pts_card.append([x1, y1])
            pts_img.append([x2, y2])

        # plot matches
        # matches = sorted(matches, key = lambda x:x.distance)
        # img_match = cv2.drawMatches(cardList[n][:,:,::-1], kp_cards[n],␣
↪img_gray, kp_img, good_matches[:100], img_gray, flags=2)
        # plt.figure(figsize=(9,6))
        # plt.imshow(img_match)
        # plt.show()

        # convert to homogenous coords
        pts_card = [[pt[0], pt[1], 1] for pt in pts_card]
        pts_img = [[pt[0], pt[1], 1] for pt in pts_img]
        pts_card = np.array(pts_card)
        pts_img = np.array(pts_img)

        # compute homography matrix using RANSAC
        N = 1000 # number of trials
        threshold = 2
        maxInliers = 0
        for i in range(N):
            # randomly select subset of points
            idx = random.sample(range(numMatches), 4)
            pts1 = pts_card[idx]
            pts2 = pts_img[idx]

            # generate hypothesis model
            H_test = ComputeH(pts1[:,0:2], pts2[:,0:2])

            # compute distances between data and estimate
            pts_img_h = np.dot(H_test, np.transpose(pts_card))
            pts_img_h = np.column_stack((pts_img_h[0,:]/pts_img_h[2,:],␣
↪pts_img_h[1,:]/pts_img_h[2,:])) # in cartesian

            dist = [np.linalg.norm(pts_img[j,0:2]-pts_img_h[j,:]) for j in␣
↪range(numMatches)]
            dist = np.array(dist)

            # select inliers within distance threshold
            numInliers = len(dist[dist<threshold])
            if(numInliers > maxInliers):
                # print(numInliers)
                H = H_test
                maxInliers = numInliers
```

```
        # print(maxInliers)
        H_arr[:,:,n] = H

    # compute locations of cards in image
    xlim = []
    ylim = []
    for i in range(len(cardList)):
        size = (cardList[i].shape[0], cardList[i].shape[1])
        corner = np.
↪array([[0,0,1],[size[1],0,1],[size[1],size[0],1],[0,size[0],1]]) #in HC
        pts = np.matmul(H_arr[:,:,i], np.transpose(corner)) # in panorama plane
        pts = np.column_stack((pts[0,:]/pts[2,:], pts[1,:]/pts[2,:])) # in␣
↪cartesian
        xlim.append([min(pts[:,0]), max(pts[:,0])])
        ylim.append([min(pts[:,1]), max(pts[:,1])])

    # determine card number using x-position
    x_pos = [np.mean(lim) for lim in xlim]
    idx = np.argsort(x_pos)
    cardNum = ['A', 'J', 'K', 'Q'] #order in which card img are read
    cardOrder = [cardNum[i] for i in idx]

    # replace Ace of Spades
    size = (card_ah.shape[0], card_ah.shape[1])
    corner = np.
↪array([[0,0,1],[size[1],0,1],[size[1],size[0],1],[0,size[0],1]]) #in HC
    pts = np.matmul(H_arr[:,:,0], np.transpose(corner)) # in panorama plane
    pts = np.column_stack((pts[0,:]/pts[2,:], pts[1,:]/pts[2,:])) # in cartesian

    warp = cv2.warpPerspective(card_ah, H_arr[:,:,0], (img.shape[1], img.
↪shape[0]))
    img_out = combineImages(warp, img, pts)

    # print card order on output image
    text = ' '.join(cardOrder)
    cv2.putText(img_out, text, (25,75), cv2.FONT_HERSHEY_DUPLEX, 2.5, (255,␣
↪255, 255), 2)

    # plot result
    fig, axs = plt.subplots(1,2, figsize=(9,6))
    axs[0].imshow(img[:,:,::-1])
    axs[0].axis('off'), axs[0].set_title('Original Image')
    axs[1].imshow(img_out[:,:,::-1])
    axs[1].axis('off'), axs[1].set_title('Result')
    fig.tight_layout()
    plt.show()
```

Original Image

Result

Original Image

Result

Original Image

Result

Original Image

Result