CS 6320 Project 4: Projection Matrix and Fundamental Matrix Estimation with RANSAC

You are expected to complete this notebook with lines of code, plots and texts. You will need to create new cells with original code or text for your analyses and explanations. Explain what you do and analyze your results. This assignment has a total of 100 points distributed for three sub-parts listed below.

(1) Projection Matrix

(2) Fundamental Matrix Estimation

(3) Fundamental Matrix with RANSAC

Brief

```
Due Data: Listed in Canvas
Hand-in : Through Canvas
Required files: <your_uid>.zip.(Please begin with 'u' for your uid)
```

Setup

0. Unzip proj4_6320.zip and go to proj4_6320 directory.
   - You can run `unzip proj4_6320.zip && cd proj4_6320` in your terminal.
1. Install Miniconda. It doesn't matter whether you use Python 2 or 3 because we will create our own environment that uses 3 anyways.

2. Create a conda environment using the appropriate command. On Windows, open the installed "Conda prompt" to run the command. On MacOS and Linux, you can just use a terminal window to run the command, Modify the command based on your OS (linux, mac, or win): `conda env create -f proj4_env_<OS>.yml`. - NOTE that proj4_env_.yml is inside the project folder.
3. This should create an environment named 'proj4'. Activate it using the Windows command, activate proj4 or the MacOS / Linux command, conda activate proj4
4. Run the notebook using `jupyter notebook` under *proj4_6320* directory.
5. Generate the zip folder for the code portion of your submission once you've finished the project using

   ```
   python zip_submission.py --uid <your_uid>
   ```

Writeup

For this project, this notebook itself is the report. You must answer each of the questions (associated with each part) in a separate cell.

You must run all your cells before you hand in it. You code, results, visualization, and discussion will be used for the grading. You will be deducted points if the results are not shown in this notebook. Do not change the order of the cells. You can add cells in need. You can copy a cell and run it seperately if you need to run a cell multiple times and thus every result is displayed in the cell.

Rubric

```
+33 pts: Part_1
+34 pts: Part_2
+33 pts: Part_3

-5 pts: Lose 5 points for every time you do not follow the instructions for the hand-in forma
```

Submission Format

This is very important as you will lose 5 points for every time you do not follow the instructions. You will attach only one items in your submission on Canvas:

1. .zip containing:
   - proj4_code/ - directory containing all your code for this assignment
   - data/ - directory containing all the input images
   - results/ - directory containing all the output images

Do not install any additional packages inside the conda environment. The TAs will use the same environment as defined in the config files we provide you, so anything that's not in there by default will probably cause your code to break during grading. Do not use absolute paths in your code or your code will break. Use relative paths. Failure to follow any of these instructions will lead to point deductions. Create the zip file using python zip_submission.py –uid (it will zip up the appropriate directories/files for you!)

Setup

```
# %matplotlib notebook
# %matplotlib inline
%load_ext autoreload
%autoreload 2
import cv2
import numpy as np
import matplotlib.pyplot as plt
from utils import *
```

Part 1 Projection Matrix Estimation

Part 1.1 Implement Camera Projection (4pts)

In this initial part, in `projection_matrix.py` you will implement camera projection in the `projection(P, points_3d)` from homogenous world coordinates $X_i = [X_i, Y_i, Z_i, 1]$ to non-homogenous pixel coordinates x_i, y_i.

It will be helpful to recall the equations to convert to pixel coordinates

$$x_i = \frac{p_{11}X_i + p_{12}Y_i + p_{13}Z_i + p_{14}}{p_{31}X_i + p_{32}Y_i + p_{33}Z_i + p_{34}} \quad y_i = \frac{p_{21}X_i + p_{22}Y_i + p_{23}Z_i + p_{24}}{p_{31}X_i + p_{32}Y_i + p_{33}Z_i + p_{34}} \tag{1}$$

```
import projection_matrix
# np.set_printoptions(suppress=True) # Suppresses printing in scientific notation

from unit_tests.part1_unit_test import (
    verify,
    test_projection,
    test_objective_func,
    test_decompose_camera_matrix,
    test_calculate_camera_center,
    test_estimate_camera_matrix)
```

```
print('Test for camera projection:', verify(test_projection))
```

```
Test for camera projection: "Correct"
```

Part 1.2: Objective Function (4pts)

In this part, in `projection_matrix.py` you will implement the objective function `objective_function(x, **kwargs)` that will be passed to `scipy.optimize.least_squares` for minimization with the Levenberg-Marquardt algorithm. The input to this function is a vectorized camera matrix, the output is the term that gets squared in the objective function and should also be vectorized. Scipy takes care of the squaring + summation part.

$$\sum_{i=1}^{N}(\hat{\mathbf{P}}\mathbf{X}_w^i - \mathbf{x}^i)^2 \tag{2}$$

```
print('Test for objective_function:', verify(test_objective_func))
```

```
Test for objective_function: "Correct"
```

Part 1.3: Estimating the Projection Matrix Given Point Correspondences (4pts)

Initially you will run the optimization to estimate $\mathbf{P}$ using an initial guess that we provide.

Good initial estimate for $\mathbf{P}$.

Optimizing the reprojection loss using Levenberg-Marquardt requires a good initial estimate for $\mathbf{P}$. This can be done by having good initial estimates for $\mathbf{K}$ and $\mathbf{R}^T$ and $\mathbf{t}$ which you can multiply to then generate your estimated $\mathbf{K}$.

```
initial_guess_K = np.array([[ 500,   0, 535],
                            [   0, 500, 390],
                            [   0,   0,  -1]])
```

```
initial_guess_R_T = np.array([[ 0.5,   -1,  0],
                            [   0,    0, -1],
                            [   1,  0.5,  0]])

initial_guess_I_t = np.array([[   1,    0, 0, 300],
                              [   0,    1, 0, 300],
                              [   0,    0, 1,  30]])

initial_guess_P = np.matmul(initial_guess_K, np.matmul(initial_guess_R_T, initial_guess_I_
```

```
# set the paths and load the data
pts2d_path = '../data/pts2d-pic_b.txt'
pts3d_path = '../data/pts3d.txt'
img_path   = '../data/pic_b.jpg'

points_2d = np.loadtxt(pts2d_path)
points_3d = np.loadtxt(pts3d_path)
img = load_image(img_path)
```

Estimate the projection matrix given corresponding 2D & 3D points

```
print('Test for estimate_camera_matrix:', verify(test_estimate_camera_matrix))
```

```
`ftol` termination condition is satisfied.
Function evaluations 1055, initial cost 2.9929e+07, final cost 6.8831e+00, first-order optima
Time since optimization start 0.033353328704833984
Test for estimate_camera_matrix: "Correct"
```

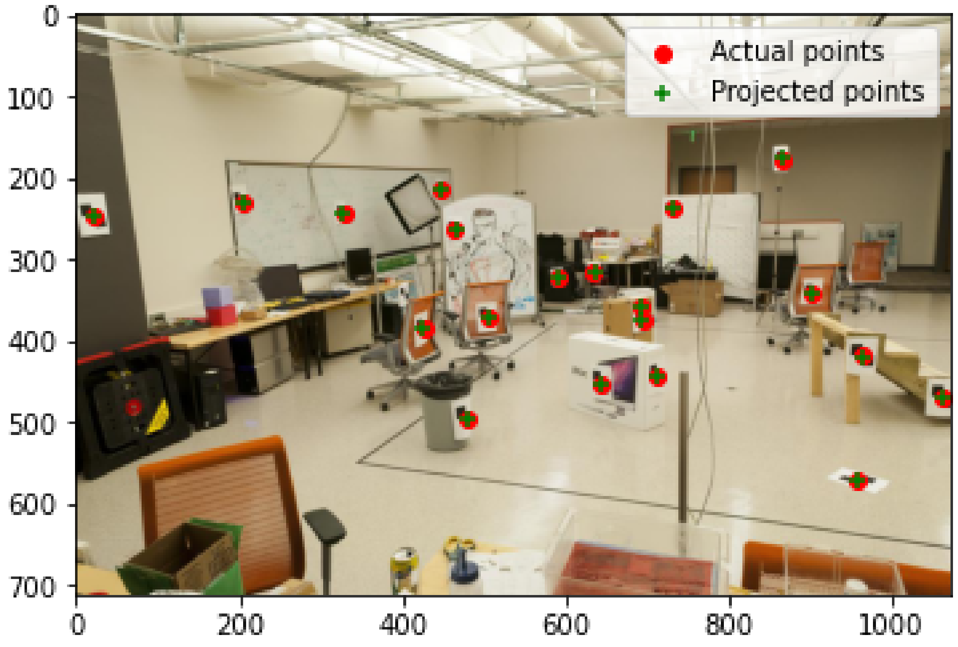
```
P = projection_matrix.estimate_camera_matrix(points_2d, points_3d, initial_guess_P)

print('The projection matrix is\n', P)

[projected_2d_pts, residual] = evaluate_points(P, points_2d, points_3d);

# residual is the sum of Euclidean distances between actual and projected points
print('The total residual is {:f}'.format(residual))
visualize_points_image(points_2d, projected_2d_pts, '../data/pic_b.jpg')
```

```
`ftol` termination condition is satisfied.
Function evaluations 1055, initial cost 2.9929e+07, final cost 6.8831e+00, first-order optima
Time since optimization start 0.03491711616516113
The projection matrix is
 [[-2.04554494e+00  1.18126347e+00  4.05588700e-01  2.44822772e+02]
 [-4.55828572e-01 -3.04148029e-01  2.14988423e+00  1.66188175e+02]
 [-2.24222858e-03 -1.09957049e-03  5.71552982e-04  1.00000000e+00]]
The total residual is 14.711462
```



Part 1.4: Decomposing the projection matrix (4pts)

In this part in `projection_matrix.py` you will implement `decompose_camera_matrix(P)` that takes as input the camera matrix $\mathbf{P}$ and outputs the intrinsic $\mathbf{K}$ and rotation matrix ${}_c\mathbf{R}_w$,

```
print('Test for decomposing projection matrix:', verify(test_decompose_camera_matrix))
K, R = projection_matrix.decompose_camera_matrix(P)
```

```
Test for decomposing projection matrix: "Correct"
```

Part 1.5: Calculating Camera Center (4pts)

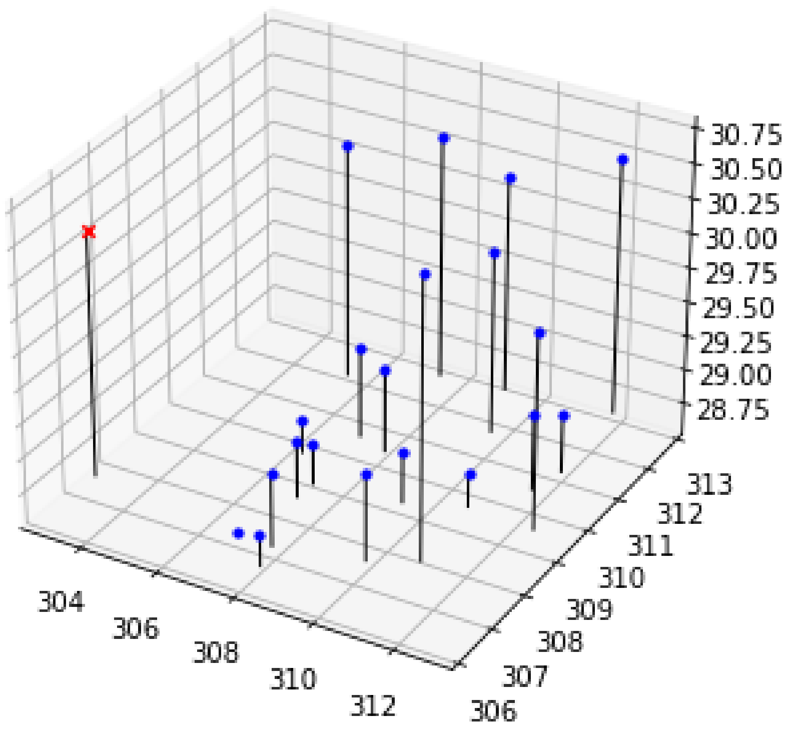
In this part in `projection_matrix.py` you will implement `calculate_camera_center(P, K, R)` that takes as input the projection $\mathbf{P}$, intrinsic $\mathbf{K}$ and extrinsic ${}_c\mathbf{R}_w$ matrix and outputs the camera position in world coordinates.

```
print('Test for calculating camera center:', verify(test_calculate_camera_center))
center = projection_matrix.calculate_camera_center(P, K, R)
```

```
Test for calculating camera center: "Correct"
```

We can now visualize the camera center and the camera coordinate system as well as the the world coordinate system.

```
plot3dview(points_3d, center)
```



Part 1.6: Taking Your Own Images and Estimating the Projection Matrix + Camera Pose (3pts)

In this part you will take two images of your fiducial object. If you want to also reuse these images for Part II, keep in mind how to take good images for estimating the Fundamental Matrix.

```
image1_path = '../data/new_book_img1.jpg'
image2_path = '../data/new_book_img2.jpg'

img1 = load_image(image1_path)
img2 = load_image(image2_path)
```

Measure your fiducial object and define a coordinate system. Fill out the `points_3d` variable with the 3D point locations of the points you'll use for correspondences.

```
points_3d = np.array([[0, 0, 0], [0, 0, 40], [0, 180, 0], [0, 180, 40], [255, 0, 0], [255,
```

Now for each image, find the 2D pixel locations of your 3D points. Hovering over the image gives you the `x,y` coordinates of your cursor on the image. You can use the lower left side controls to zoom into the image for more precise measurements. Fill out `points2d_img1` with these coordinate values.

```
# plotting image 1
fig = plt.figure(); ax = fig.add_subplot(111); ax.imshow(img1)
```

```
points2d_img1 = np.array([[3292, 2644], [3335, 2275], [3748, 2086], [3765, 1820], [1350, 2
```

```
# plotting image 2
fig = plt.figure(); ax = fig.add_subplot(111); ax.imshow(img2)
```

```
points2d_img2 = np.array([[1608, 2730], [1638, 2347], [2652, 2168], [2648, 1897], [268, 21
```

Our objective function will need to read the measurements you just saved from disk. We need to save this data now.

```
np.savetxt('../results/pts3d_fiducial.npy', points_3d)
np.savetxt('../results/pts2d_image1.npy', points2d_img1)
np.savetxt('../results/pts2d_image2.npy', points2d_img2)
```

Part 1.7: Making your own K, R^T and $[\mathrm{I}|\mathrm{t}]$ estimates. (3pts)

```
initial_guess_K = np.array([[   1,    0,   0],
                            [   0,    1,   0],
                            [   0,    0,   1]])

initial_guess_R_T = np.array([[ 1,    0, 0],
                              [ 0,    1, 0],
                              [ 0,    0, 1]])

initial_guess_I_t = np.array([[   1,     0, 0, 80],
                              [   0,     1, 0, 290],
                              [   0,     0, 1, -150]])

initial_guess_P = np.matmul(initial_guess_K, np.matmul(initial_guess_R_T, initial_guess_I_
```

```
# set the paths and load the data
pts2d_path = '../results/pts2d_image1.npy'
pts3d_path = '../results/pts3d_fiducial.npy'

points_2d = np.loadtxt(pts2d_path)
points_3d = np.loadtxt(pts3d_path)
img = load_image(image1_path)
```

Visualize your estimate for the camera pose relative to the world coordinate system. RGB colors correspond with XYZ (first, second and third coordinate). Be mindful of whether you should be passing `R` or `R.T` in for your rotation matrix.

```
plot3dview_with_coordinates(points_3d, -initial_guess_I_t[:,3], initial_guess_R_T.T)
```

View the optimization results given your initial guess. If your initial guess is poor the optimizition **will not** work. You will need to make initial estimates for both the images you took.

```
P1 = projection_matrix.estimate_camera_matrix(points_2d, points_3d, initial_guess_P)
print('The projection matrix is\n', P1)

[projected_2d_pts, residual] = evaluate_points(P1, points_2d, points_3d);
print('The total residual is {:f}'.format(residual))
visualize_points_image(points_2d, projected_2d_pts, image1_path)
```

Visualize your estimate for the camera pose relative to the world coordinate system.

```
# set the pats and load the data
pts2d_path = '../results/pts2d_image2.npy'
pts3d_path = '../results/pts3d_fiducial.npy'

points_2d = np.loadtxt(pts2d_path)
points_3d = np.loadtxt(pts3d_path)
img = load_image(image1_path)
np.array(img.shape[:2])/2
```

```
initial_guess_K = np.array([[   1,   0,  0],
                            [   0,   1,  0],
                            [   0,   0,  1]])

initial_guess_R_T = np.array([[ 1,   0, 0],
                              [ 0,   1, 0],
                              [ 0,   0, 1]])

initial_guess_I_t = np.array([[1, 0, 0, 80],
                              [0, 1, 0, 290],
                              [0, 0, 1, -160]])

initial_guess_P = np.matmul(initial_guess_K, np.matmul(initial_guess_R_T, initial_guess_I_
```

```
plot3dview_with_coordinates(points_3d, -initial_guess_I_t[:,3], initial_guess_R_T.T) #chan
```

```
P2 = projection_matrix.estimate_camera_matrix(points_2d, points_3d, initial_guess_P)
#M = sc.calculate_projection_matrix(points_2d, points_3d)
print('The projection matrix is\n', P2)

[projected_2d_pts, residual] = evaluate_points(P2, points_2d, points_3d);
print('The total residual is {:f}'.format(residual))
visualize_points_image(points_2d, projected_2d_pts, image2_path)
#visualize_points(points_2d, projected_2d_pts)
```

Part 1.8 Visualizing both camera poses in the world coordinate system (3pts)

```
K1, R1 = projection_matrix.decompose_camera_matrix(P1)
center_1 = projection_matrix.calculate_camera_center(P1, K1, R1);
print(center_1)

K2, R2 = projection_matrix.decompose_camera_matrix(P2)
center_2 = projection_matrix.calculate_camera_center(P2, K2, R2);
print(center_2)

plot3dview_2_cameras(points_3d, center_1, center_2, R1.T, R2.T)
```

Part 1.9 Questions (4pts):

1. What would happen to the projected points if you increased/decreased the x coordinate, or the other coordinates of the camera center t? Write down a description of your expectations in the appropriate part of your writeup submission.
2. Perform this shift for each of the camera coordinates and then recompose the projection matrix and visualize the result in your Jupyter notebook. You could use the image given as the part of the assignment for this analyses.

Answer

```
initial_guess_K = np.array([[   1,    0,   0],
                            [   0,    1,   0],
                            [   0,    0,   1]])

initial_guess_R_T = np.array([[ 1,    0, 0],
                              [ 0,    1, 0],
                              [ 0,    0, 1]])

initial_guess_I_t = np.array([[1, 0, 0, 70],
                              [0, 1, 0, 290],
                              [0, 0, 1, -160]])

initial_guess_P = np.matmul(initial_guess_K, np.matmul(initial_guess_R_T, initial_guess_I_
# set the pats and load the data
pts2d_path = '../results/pts2d_image1.npy'
pts3d_path = '../results/pts3d_fiducial.npy'

points_2d = np.loadtxt(pts2d_path)
points_3d = np.loadtxt(pts3d_path)
img = load_image(image1_path)
np.array(img.shape[:2])/2
```

```
P = projection_matrix.estimate_camera_matrix(points_2d, points_3d, initial_guess_P)

print('The projection matrix is\n', P)

[projected_2d_pts, residual] = evaluate_points(P, points_2d, points_3d);

# residual is the sum of Euclidean distances between actual and projected points
print('The total residual is {:f}'.format(residual))
visualize_points_image(points_2d, projected_2d_pts, image1_path)
```

```
initial_guess_K = np.array([[   1,    0,   0],
                            [   0,    1,   0],
                            [   0,    0,   1]])

initial_guess_R_T = np.array([[ 1,    0, 0],
                              [ 0,    1, 0],
                              [ 0,    0, 1]])

initial_guess_I_t = np.array([[1, 0, 0, -50],
                              [0, 1, 0, 290],
                              [0, 0, 1, -160]])

initial_guess_P = np.matmul(initial_guess_K, np.matmul(initial_guess_R_T, initial_guess_I_
```

```
# set the pats and load the data
pts2d_path = '../results/pts2d_image1.npy'
pts3d_path = '../results/pts3d_fiducial.npy'

points_2d = np.loadtxt(pts2d_path)
points_3d = np.loadtxt(pts3d_path)
img = load_image(image1_path)
```

```
np.array(img.shape[:2])/2
```

```
P = projection_matrix.estimate_camera_matrix(points_2d, points_3d, initial_guess_P)

print('The projection matrix is\n', P)

[projected_2d_pts, residual] = evaluate_points(P, points_2d, points_3d);

# residual is the sum of Euclidean distances between actual and projected points
print('The total residual is {:f}'.format(residual))
visualize_points_image(points_2d, projected_2d_pts, image1_path)
```

Part 2 Fundamental Matrix Estimation

In this part you'll be estimating the fundamental matrix. You'll be using a least squares optimizer from SciPy. (Documentation here: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.le

The least squares optimizer takes an objective function, your variables to optimize, and the points that you want to fit a line to. In this case, the objective function is to minimize the point to line distance, where the line is the projection of a point onto another image by the fundamental matrix, and the point is an actual point of a feature in that image. The variable that you want to optimize would be the 9 values in the 3x3 Fundamental Matrix. The points that you are optimizing over are provided to you, and they are the homogeneous coordinates of corresponding features from two different images of the same scene.

Part 2.1 Estimate Fundamental Matrix (12pts)

Implement the `point_line_distance()` method in fundamental_matrix.py.

$$d(line, point) = \frac{au + bv + c}{\sqrt{a^2 + b^2}} \tag{3}$$

```
from unit_tests.test_fundamental_matrix import verify
from unit_tests.test_fundamental_matrix import TestFundamentalMatrix

test_fundamental_matrix_stereo = TestFundamentalMatrix()
TestFundamentalMatrix.setUp(test_fundamental_matrix_stereo)
print("test_point_line_distance(): " + verify(test_fundamental_matrix_stereo.test_point_li
print("test_point_line_distance_zero(): " + verify(test_fundamental_matrix_stereo.test_poi
```

Implement `signed_point_line_errors()`.

Keep in mind that SciPy does the squaring and summing for you, so all you have to do in `signed_point_line_errors()` is return a list of each individual error. So if there are 9 points, you have to calculate the `point_line_distance()` between each pair from Fx_1 to x_0 and also $F^T x_0$ to x_1, then append all errors to a list, such that you end up returning a list of length 18. SciPy will take the list and square each element and sum everything for you.

`signed_point_line_errors()`:

$$d(Fx_1, x_0)^2 + d(F^T x_0, x_1)^2 \tag{4}$$

You'll also have to make the call to SciPy's least squares optimizer in the `optimize()` method in least_squares_fundamental_matrix.py.

```
from unit_tests.test_fundamental_matrix import TestFundamentalMatrix2, TestFundamentalMatr

print("TestFundamentalMatrix():")
print("test_signed_point_line_errors(): " + verify(test_fundamental_matrix_stereo.test_sig
print("test_least_squares_optimize(): " + verify(test_fundamental_matrix_stereo.test_least

print("TestFundamentalMatrix2():")
test_fundamental_matrix_synthetic = TestFundamentalMatrix2()
TestFundamentalMatrix2.setUp(test_fundamental_matrix_synthetic)
print("test_signed_point_line_errors(): " + verify(test_fundamental_matrix_synthetic.test_
print("test_least_squares_optimize(): " + verify(test_fundamental_matrix_synthetic.test_le

print("TestFundamentalMatrix3():")
test_fundamental_matrix_real = TestFundamentalMatrix3()
TestFundamentalMatrix3.setUp(test_fundamental_matrix_real)
print("test_signed_point_line_errors(): " + verify(test_fundamental_matrix_real.test_signe
print("test_least_squares_optimize(): " + verify(test_fundamental_matrix_real.test_least_s
```

Then run the following cell to find the Fundamental Matrix using least squares. You should see the epipolar lines in the correct places in the image.

```
%matplotlib inline
# Load the data for room images
points_2d_pic_a = np.loadtxt('../data/pts2d-pic_a.txt')
points_2d_pic_b = np.loadtxt('../data/pts2d-pic_b.txt')
img_left = load_image('../data/pic_a.jpg')
```

```
img_right = load_image('../data/pic_b.jpg')

import least_squares_fundamental_matrix as ls

F = ls.solve_F(points_2d_pic_a, points_2d_pic_b)
print(F)

# Draw epipolar lines using the fundamental matrix
draw_epipolar_lines(F, img_left, img_right, points_2d_pic_a, points_2d_pic_b)
```

Part 2.2 Try Fundamental Matrix Estimation Yourself (10pts)

Now you're going to take two images yourself and estimate the fundamental matrix between them. To do this, take two images and save them as "my_image_0.jpg" and "my_image_1.jpg" in the "/data" folder. You will want an image with lots of features at varying depths.

```
# Load the data for room images
my_img_left = load_image('../data/my_image_0_new.jpg')
my_img_right = load_image('../data/my_image_1_new.jpg')
```

To collect your own data points, run the following cell and mouse over features in the image and record the x- and y-coordinates. You'll need at least 9 points because we are trying to optimize for 9 variables, one for each element in the 3x3 fundamental matrix. Think about how you can choose good features for estimating the fundamental matrix.

Store your points in variable "my_image_0_pts" and "my_image_1_pts" respectively.

```
%matplotlib
# plotting image 1
image_0 = plt.figure(); image_0_ax = image_0.add_subplot(111); image_0_ax.imshow(my_img_le
```

```
#my_image_0_pts = np.array([[0, 0]]) # Record your coordinates here
my_image_0_pts = np.array([[336.0, 1900.0], [202.0, 1673.0], [1226.0, 425.0], [3359.0, 731
```

```
# plotting image 2
image_1 = plt.figure(); image_1_ax = image_1.add_subplot(111); image_1_ax.imshow(my_img_ri
#my_image_1_pts = np.array([[0, 0]]) # Record your coordinates here
my_image_1_pts = np.array([[2339.0, 2856.0], [2387.0, 2629.0], [854.0, 11218.0], [2528.0,
```

```
%matplotlib inline
import two_view_data as two_view_data
my_F = ls.solve_F(my_image_0_pts, my_image_1_pts)
print(my_F)

# Draw epipolar lines using the fundamental matrix
draw_epipolar_lines(my_F, my_img_left, my_img_right, my_image_0_pts, my_image_1_pts)
```

Part 2.3 Questions (12pts)

1. Why is it that when you take your own images, you can't just rotate the camera or zoom the image for your two images of the same scene?
2. Why is it that points in one image are projected by the fundamental matrix onto epipolar lines in the other image?
3. What happens to the epipoles and epipolar lines when you take two images where the camera centers are within the images? Why?
4. What does it mean when your epipolar lines are all horizontal across the two images?

Answer 1

Answer 2

Answer 3

Answer 4

Part 3: Fundamental Matrix with RANSAC

Now we will automate the process of finding the fundamental matrix for an image by using SIFTNet to identify interest points and using RANSAC to robustly find true interest point matches between the two images. We will give you the detected features by SIFTNet.

We will implement a workflow using the detected features by SIFTNet, then RANSAC will select a random subset of those points, you will call your function from part 2 to calculate the fundamental matrix for those points, and then you will check how many other points identified by SIFTNet match this fundamental matrix. Then you will repeat this process and select another subset of points using RANSAC until you find the subset of points that produces the best fundamental matrix with the most matching points. Refer to the lecture slides for the RANSAC workflow.

You can also find a simple explanation of RANSAC at https://www.mathworks.com/discovery/ransac.html. See section 8.1.4 in the textbook for a more thorough explanation of how RANSAC works.

3.1 RANSAC Iterations (6pts)

Begin by calculating the number of iterations S RANSAC will need to perform to guarentee a given success rate P knowing the number of points included in the sample k and the probability of an individual point being a true match p. To derive this formula, consider the following: * the probability that any one point has a true match is p * conversely the probability that any one point is not a match is $1 - p$ * the probability that two points are both matches is then $p \cdot p$ * this can be extendeed to k points, for which the probability that they are all true matches is p^k * on the other hand, we want the probability that k points are all true matches to be P (and the probability that they are not to be $1-P$) * by repeatedly sampling k points, we can reduce the probability that all of the samples do not contain k true matches * After S samples we want the probability of failure to equal $1 - P$

Start by setting up this equality

$$1 - P = ...$$

and then rearange it to write a function to solve for S

```
from ransac import calculate_num_ransac_iterations
from unit_tests.test_ransac import test_calculate_num_ransac_iterations

P = 0.999
k = 9
p = 0.9
# call their ransac iterations function
S = calculate_num_ransac_iterations(P, k, p)
# print number of trials they will need to run
print('S =', int(S))

print("Test for calculate_num_ransac_iterations(): " + verify(test_calculate_num_ransac_it
```

3.2 Questions (3pts)

1. How many RANSAC iterations would we need to find the fundamental matrix with 99.9% certainty from your Mount Rushmore SIFTNet results assuming that they had a

90% point correspondence accuracy?

2. One might imagine that if we had more than 9 point correspondences, it would be better to use more of them to solve for the fundamental matrix. Investigate this by finding the number of RANSAC iterations you would need to run with 18 points.
3. If our dataset had a lower point correspondence accuracy, say 70%, what is the minimum number of iterations needed to find the fundamental matrix with 99.9% certainty?

Answer 1

Answer 2

Answer 3

3.3 RANSAC Implementation (12pts)

Next we will implement the RANSAC algorithm. Remember the steps from the link above: 1. Randomly selecting a subset of the data set 1. Fitting a model to the selected subset 1. Determining the number of outliers 1. Repeating steps 1-3 for a prescribed number of iterations

For the application of finding true point pair matches and using them to calculate the fundamental matrix, our subset of the data will be the minimum number of point pairs needed to calculate the fundamental matrix. The model we are fitting is the fundamental matrix. Outliers will be found by using the `point_line_distance()` error function from part 2 and thresholding with a certain margin of error.

```
from ransac import ransac_fundamental_matrix
from unit_tests import test_ransac

points_a = np.load('unit_tests/pointsa.npy')
points_b = points_a

F, _, _ = ransac_fundamental_matrix(points_a, points_b)
print('F= ', F)

print("Test for ransac_find_inliers(): " + verify(test_ransac.test_ransac_find_inliers))
print("Test for ransac_fundamental_matrix(), F matches inliers: " + verify(test_ransac.tes
print("Test for ransac_fundamental_matrix(), F matches all points: " + verify(test_ransac.
```

3.4 Finally we will put it all together (12pts)

The code below will load the detected features by SIFTNet, determine the number RANSAC iterations using your function, and run your RANSAC calculating the fundamental matrix with your function at each pass. You shouldn't have to implement any new functions for this.

```
from feature_matching.utils import load_image, PIL_resize, rgb2gray
import torch
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

# Rushmore
image1 = load_image('../data/2a_rushmore.jpg')
image2 = load_image('../data/2b_rushmore.jpg')

scale_factor = 0.5
image1 = PIL_resize(image1, (int(image1.shape[1]*scale_factor), int(image1.shape[0]*scale_
image2 = PIL_resize(image2, (int(image2.shape[1]*scale_factor), int(image2.shape[0]*scale_
image1_bw = rgb2gray(image1)
image2_bw = rgb2gray(image2)

#convert images to tensor
tensor_type = torch.FloatTensor
torch.set_default_tensor_type(tensor_type)
to_tensor = transforms.ToTensor()
image_input1 = to_tensor(image1_bw).unsqueeze(0)
image_input2 = to_tensor(image2_bw).unsqueeze(0)

# load detected features by SIFTNet and the matches
image1_feats = np.load("../data/2a_rushmore_feats.npz")
image2_feats = np.load("../data/2b_rushmore_feats.npz")
feats_matches = np.load("../data/2a2b_rushmore_matches.npz")
x1, y1 = image1_feats['x'], image1_feats['y']
x2, y2 = image2_feats['x'], image2_feats['y']
matches, confidences = feats_matches['matches'], feats_matches['confidences']

print('{:d} corners in image 1, {:d} corners in image 2'.format(len(x1), len(x2)))
print('{:d} matches from {:d} corners'.format(len(matches), len(x1)))

from feature_matching.utils import show_correspondence_circles, show_correspondence_lines
```

```
# num_pts_to_visualize = len(matches)
num_pts_to_visualize = 100
c2 = show_correspondence_lines(image1, image2,
                    x1[matches[:num_pts_to_visualize, 0]], y1[matches[:num_pts_to_visualiz
                    x2[matches[:num_pts_to_visualize, 1]], y2[matches[:num_pts_to_visualiz
plt.figure(); plt.title('Proposed Matches'); plt.imshow(c2)

from ransac import ransac_fundamental_matrix
# print(image1_features.shape, image2_features.shape)
num_features = min([len(image1_features), len(image2_features)])
x0s = np.zeros((len(matches), 2))
x1s = np.zeros((len(matches), 2))
x0s[:,0] = x1[matches[:, 0]]
x0s[:,1] = y1[matches[:, 0]]
x1s[:,0] = x2[matches[:, 1]]
x1s[:,1] = y2[matches[:, 1]]
# print(image1_pts.shape)
F, matches_x0, matches_x1 = ransac_fundamental_matrix(x0s, x1s)
print(F)
# print(matches_x0)
# print(matches_x1)

from utils import draw_epipolar_lines
# Draw the epipolar lines on the images and corresponding matches
match_image = show_correspondence_lines(image1, image2,
                                   matches_x0[:num_pts_to_visualize, 0], matches_x0[:num_p
                                   matches_x1[:num_pts_to_visualize, 0], matches_x1[:num_p
plt.figure(); plt.title('True Matches'); plt.imshow(match_image)
draw_epipolar_lines(F, image1, image2, matches_x0, matches_x1)
```

We can visualize the fundamental matrix by drawing the epipolar lines as shown above. As we learned these lines are the projections of the points in one image onto the other image. The point where they intersect is where the other picture was taken.