HW3

February 10, 2021

1 CSE 152A Winter 2021 – Assignment 3

1.1 Instructor: Ben Ochoa

- Assignment Published On: **Wed, Feb 10, 2021**
- Due On: **Wed, Feb 24, 2021 11:59 PM (Pacific Time)**

1.2 Instructions

Please answer the questions below using Python in the attached Jupyter notebook and follow the guidelines below:

- This assignment must be completed **individually**. For more details, please follow the Academic Integrity Policy and Collaboration Policy on Canvas.
- All the solutions must be written in this Jupyter notebook.
- After finishing the assignment in the notebook, please export the notebook as a PDF and submit both the notebook and the PDF (i.e. the `.ipynb` and the `.pdf` files) on Gradescope.
- You may use basic algebra packages (e.g. `NumPy`, `SciPy`, etc) but you are not allowed to use the packages that directly solve the problems. Feel free to ask the instructor and the teaching assistants if you are unsure about the packages to use.
- It is highly recommended that you begin working on this assignment early.

Late Policy: Assignments submitted late will receive a 15% grade reduction for each 12 hours late (i.e., 30% per day). Assignments will not be accepted 72 hours after the due date. If you require an extension (for personal reasons only) to a due date, you must request one as far in advance as possible. Extensions requested close to or after the due date will only be granted for clear emergencies or clearly unforeseeable circumstances.

1.3 Problem 1: Theory [20 points]

1.3.1 Problem 1.1: Epipolar Geometry [10 points]

Consider two cameras whose image planes are the z=2 plane, and whose focal points are at (-6, 0, 0) and (6, 0, 0). See Fig 1.1 below. We'll call a point in the first camera (x, y), and a point in the second camera (u, v). Points in each camera are relative to the camera center. So, for example if

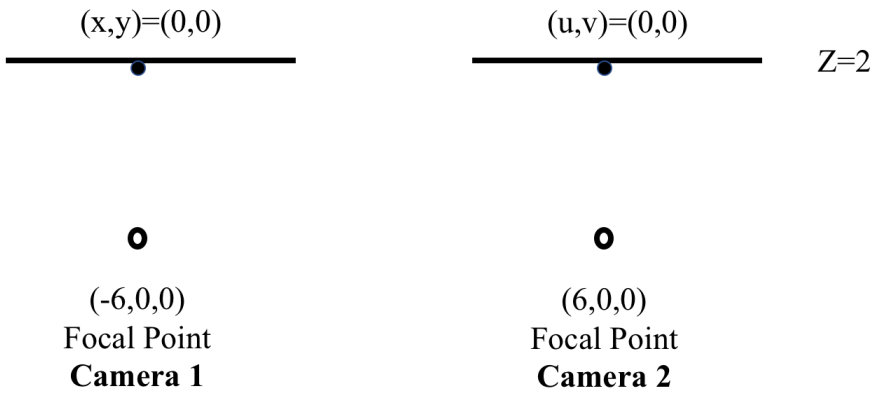


fig 1.1

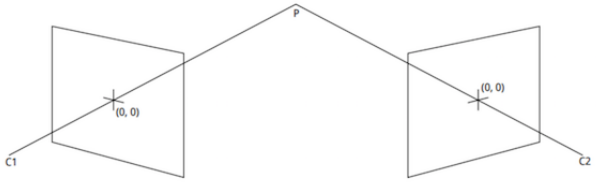


fig 1.2

(x, y) = (0, 0), this is really the point (-6, 0, 2) in world coordinates, while if (u, v) = (0, 0) this is the point (6, 0, 2).

Suppose the point (x, y) = (3, 3) is matched to the point (u, v) = (2, 3). What is the 3D location of this point?

1.3.2 Problem 1.2: The Epipolar Constraint [5 points]

Suppose two cameras fixate on a point P in space such that their principal axes intersect at that point. (See the fig. 1.2 below.) Show that if the image coordinates are normalized so that the coordinate origin (0, 0) coincides with the principal point, then the F_{33} element of the fundamental matrix is zero.

In the figure, $C1$ and $C2$ are the optical centers. The principal axes intersect at point P.

1.3.3 Problem 1.3: Essential Matrix [5 points]

Suppose a stereo rig is formed of two cameras: the rotation matrix and translation vector are given to you. Please write down the essential matrix. Also, compute the rank of the essential matrix using SVD, i.e., the number of nonzero singular values. (Note that if you get a singular value s of a very small number in your calculation, e.g., $s <= 1e-15$, you can treat it as zero singular value).

$$R = \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & 0 \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$t = \begin{bmatrix} 2 \\ 5 \\ 1 \end{bmatrix}$$

1.4 Problem 2: SSD (Sum Squared Distance) and NCC (Normalized Cross-Correlation) Matching [21 points]

In this part, you have to write two functions ssdMatch and nccMatch that implement the computation of the matching score for two given windows with SSD and NCC metrics respectively.

1.4.1 Problem 2.1: SSD (Sum Squared Distance) Matching [5 points]

Complete the function ssdMatch:
SSD = $\sum_{x,y} |W_1(x,y) - W_2(x,y)|^2$

```
In [ ]: import numpy as np
        import cv2
        import matplotlib.pyplot as plt
        import math
        import imageio
        import pickle
```

```
In [ ]: def ssdMatch(img1, img2, c1, c2, R):
            """Compute SSD given two windows.

            Args:
                img1: Image 1.
                img2: Image 2.
                c1: Center (in image coordinate) of the window in image 1.
                c2: Center (in image coordinate) of the window in image 2.
                R: R is the radius of the patch, 2 * R + 1 is the window size

            Returns:
                SSD matching score for two input windows.

            """
            """ ==========
            YOUR CODE HERE
            ========== """


            return matching_score
```

```
In [ ]: # Here is the code for you to test your implementation
        img1 = np.array([[1, 2, 3, 4], [4, 5, 6, 8], [7, 8, 9, 4]])
        img2 = np.array([[1, 2, 1, 3], [6, 5, 4, 4], [9, 8, 7, 3]])
        print(ssdMatch(img1, img2, np.array([1, 1]), np.array([1, 1]), 1))
        # should print 20
        print(ssdMatch(img1, img2, np.array([2, 1]), np.array([2, 1]), 1))
        # should print 30
        print(ssdMatch(img1, img2, np.array([1, 1]), np.array([2, 1]), 1))
        # should print 46
```

1.4.2 Problem 2.2: NCC (Normalized Cross-Correlation) Matching [8 points]

Complete the function nccMatch: NCC = $\sum_{x,y} \tilde{W}_1(x,y) \cdot \tilde{W}_2(x,y)$ where $\tilde{W} = \frac{W - \overline{W}}{\sqrt{\sum_{x,y}(W(x,y) - \overline{W})^2}}$ is a mean-shifted and normalized version of the window and $\overline{W}$ is the mean pixel value in the window W.

```
In [ ]: def nccMatch(img1, img2, c1, c2, R):
            """Compute NCC given two windows.

            Args:
                img1: Image 1.
                img2: Image 2.
                c1: Center (in image coordinate) of the window in image 1.
                c2: Center (in image coordinate) of the window in image 2.
                R: R is the radius of the patch, 2 * R + 1 is the window size

            Returns:
                NCC matching score for two input windows.

            """

            """ ==========
            YOUR CODE HERE
            ========== """

            return matching_score
```

```
In [ ]: # Here is the code for you to test your implementation
        img1 = np.array([[1, 2, 3, 4], [4, 5, 6, 8], [7, 8, 9, 4]])
        img2 = np.array([[1, 2, 1, 3], [6, 5, 4, 4], [9, 8, 7, 3]])
        print(nccMatch(img1, img2, np.array([1, 1]), np.array([1, 1]), 1))
        # should print 0.8546
        print(nccMatch(img1, img2, np.array([2, 1]), np.array([2, 1]), 1))
        # should print 0.8457
        print(nccMatch(img1, img2, np.array([1, 1]), np.array([2, 1]), 1))
        # should print 0.6258
```

1.4.3 Problem 2.3: Naive Matching [8 points]

Given the corner points detected and the NCC matching function, we are ready to start finding correspondences. One naive strategy is to try and find the best match between the two sets of corner points. Write a script that does this, namely, for each corner in image1, find the best match from the detected corners in image2 (or, if the NCC match score is too low, then return no match for that point).

Write a function naive_matching and call it as below. Examine your results for 20 detected corners in each image.

```
In [ ]: def naive_matching(img1, img2, corners1, corners2, R, NCCth):
            """Compute NCC given two windows.

            Args:
                img1: Image 1.
                img2: Image 2.
                corners1: Corners in image 1 (nx2)
                corners2: Corners in image 2 (nx2)
                R: NCC matching radius
                NCCth: NCC matching score threshold

            Returns:
                NCC matching result a list of tuple (c1, c2),
                c1 is the 1x2 corner location in image 1,
                c2 is the 1x2 corner location in image 2.

            """

            """ ==========
            YOUR CODE HERE
            ========== """

            return matching
```

```
In [ ]: def rgb2gray(rgb):
            """ Convert rgb image to grayscale.
            """
            return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])
        # detect corners on warrior and matrix sets
        # you are free to modify code here, create your helper functions, etc.

        nCorners = 20
        smoothSTD = 1
        windowSize = 17

        # read images and detect corners on images

        imgs_mat = []
        crns_mat = []
```

```
imgs_war = []
crns_war = []

for i in range(2):
    img_mat = imageio.imread('p4/matrix/matrix' + str(i) + '.png')
    imgs_mat.append(rgb2gray(img_mat))
    img_war = imageio.imread('p4/warrior/warrior' + str(i) + '.png')
    imgs_war.append(rgb2gray(img_war))
```

In [ ]:
```
# match corners
crnsmatf=open('crns_mat.pkl','rb')
crns_mat=pickle.load(crnsmatf)
crnswarf=open('crns_mat.pkl','rb')
crns_war=pickle.load(crnswarf)
R = 120
NCCth = 0.6  # put your threshold here
matching_mat = naive_matching(imgs_mat[0]/255, imgs_mat[1]/255, crns_mat[0], crns_mat[
matching_war = naive_matching(imgs_war[0]/255, imgs_war[1]/255, crns_war[0], crns_war[
```

In [ ]:
```
# plot matching result
def show_matching_result(img1, img2, matching):
    fig = plt.figure(figsize=(8, 8))
    plt.imshow(np.hstack((img1, img2)), cmap='gray') # two dino images are of differen
    for p1, p2 in matching:
        plt.scatter(p1[0], p1[1], s=35, edgecolors='r', facecolors='none')
        plt.scatter(p2[0] + img1.shape[1], p2[1], s=35, edgecolors='r', facecolors='no
        plt.plot([p1[0], p2[0] + img1.shape[1]], [p1[1], p2[1]])
    plt.savefig('dino_matching.png')
    plt.show()

print("Number of Corners:", nCorners)
show_matching_result(imgs_mat[0], imgs_mat[1], matching_mat)
show_matching_result(imgs_war[0], imgs_war[1], matching_war)
```

1.5 Problem 3: Epipolar Geometry [34 points]

As shown in Problem 2, the naive mathing algorithm is simple. The weakness of this method comes from the high matching complexity. In this problem, we will use the epipolar geometry constraint on the rectified images and updated corner points to build a better matching algorithm.

1.5.1 Problem 3.1: Fundamental matrix [10 points]

Complete the compute_fundamental function below using the 8-point algorithm described in lecture. Note that the normalization of the corner points is handled in the fundamental_matrix function. Hint: When you try to find the non-trival solution to an linear equation system **Af=0**, you can use singular value decomposition (SVD) method:SVD(**A**)=USVT. And **f** is given by the singular vector corresponding to the smallest singular value, which is the last column of V.

```
In [ ]: import numpy as np
        from imageio import imread
        import matplotlib.pyplot as plt
        from scipy.io import loadmat
        from numpy.linalg import svd

        def compute_fundamental(x1, x2):
            """ Computes the fundamental matrix from corresponding points
                (x1,x2 3*n arrays) using the 8 point algorithm.

                Construct the A matrix according to lecture
                and solve the system of equations for the entries of the fundamental matrix.

                Returns:
                Fundamental Matrix (3x3)
            """

            """ ==========
            YOUR CODE HERE
            ========== """


            # constrain F
            # make rank 2 by zeroing out last singular value
            U,S,V = np.linalg.svd(F)
            S[2] = 0
            F = np.dot(U,np.dot(np.diag(S),V))

            return F/F[2,2]

In [ ]: def fundamental_matrix(x1,x2):
            # Normalization of the corner points is handled here
            n = x1.shape[1]
            if x2.shape[1] != n:
                raise ValueError("Number of points don't match.")

            # normalize image coordinates
            x1 = x1 / x1[2]
            mean_1 = np.mean(x1[:2],axis=1)
            S1 = np.sqrt(2) / np.std(x1[:2])
            T1 = np.array([[S1,0,-S1*mean_1[0]],[0,S1,-S1*mean_1[1]],[0,0,1]])
            x1 = np.dot(T1,x1)

            x2 = x2 / x2[2]
            mean_2 = np.mean(x2[:2],axis=1)
            S2 = np.sqrt(2) / np.std(x2[:2])
            T2 = np.array([[S2,0,-S2*mean_2[0]],[0,S2,-S2*mean_2[1]],[0,0,1]])
            x2 = np.dot(T2,x2)
```

```
    # compute F with the normalized coordinates
    F = compute_fundamental(x1,x2)

    # reverse normalization
    F = np.dot(T1.T,np.dot(F,T2))

    return F/F[2,2]
```

```
In [ ]: # Here is the code for you to test your implementation
        cor1 = np.load("./p4/"+'dino'+"/cor1.npy")
        cor2 = np.load("./p4/"+'dino'+"/cor2.npy")
        print(fundamental_matrix(cor1,cor2))
        # should print
        #[[ 4.00502510e-07  3.09619039e-06 -2.86966053e-03]
        #[-2.69900666e-06 -1.00972419e-08  6.70452915e-03]
        #[ 1.37819769e-03 -7.29675791e-03  1.00000000e+00]]
```

1.5.2 Problem 3.2: Epipoles [8 points]

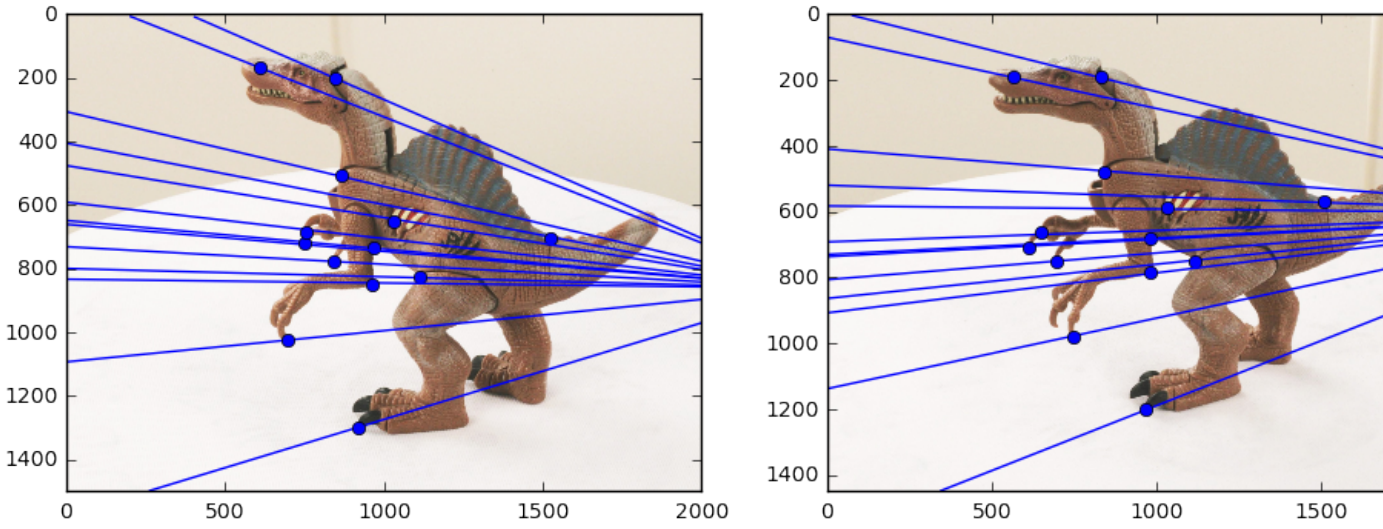
In this part, you are supposed to complete the function compute_epipole to calculate the epipoles for a given fundamental matrix.

```
In [ ]: def compute_epipole(F):
            """
            This function computes the epipoles for a given fundamental matrix.

            input:
              F --> fundamental matrix
            output:
              e1 --> corresponding epipole in image 1
              e2 --> epipole in image2
            """

            """ ==========
            YOUR CODE HERE
            ========== """

            return e1,e2
```

```
In [ ]: # Here is the code for you to test your implementation
        F_test = np.array([[1, 2, 1], [6, 5, 4], [9, 8, 1]])
        print(compute_epipole(F_test))
        # should print
        #(array([-65.3659783 ,  15.85984739,   1.        ]),
        #array([-41.86658577,  46.87378417,   1.        ]))
```

1.5.3 Problem 3.3: Plot Epipolar Lines [16 points]

Using this fundamental matrix, plot the epipolar lines in both images for each image pair. For this part, you will want to complete the function plot_epipolar_lines. Show your result for matrix and warrior as exemplified by the figure below.




```
In [ ]: def plot_epipolar_lines(img1, img2, cor1, cor2):
            """Plot epipolar lines on image given image, corners

            Args:
                img1: Image 1.
                img2: Image 2.
                cor1: Corners in homogeneous image coordinate in image 1 (3xn)
                cor2: Corners in homogeneous image coordinate in image 2 (3xn)

            """


            assert cor1.shape[0] == 3
            assert cor2.shape[0] == 3
            assert cor1.shape == cor2.shape

            F = fundamental_matrix(cor1, cor2)
            e1, e2 = compute_epipole(F)

            """ ==========
            YOUR CODE HERE
            ========== """


In [ ]: # replace images and corners with those of matrix and warrior
        imgids = ["dino", "matrix", "warrior"]
        for imgid in imgids:
```

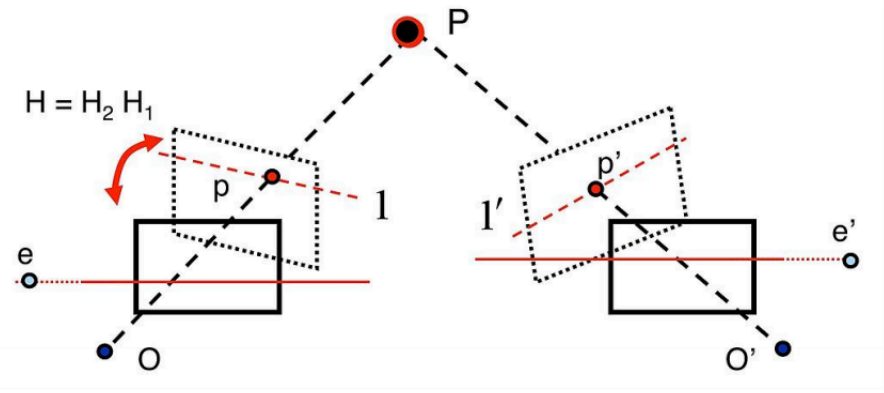
```
I1 = imageio.imread("./p4/"+imgid+"/"+imgid+"0.png")
I2 = imageio.imread("./p4/"+imgid+"/"+imgid+"1.png")
cor1 = np.load("./p4/"+imgid+"/cor1.npy")
cor2 = np.load("./p4/"+imgid+"/cor2.npy")
plot_epipolar_lines(I1,I2,cor1,cor2)
```

1.6 Problem 4: Image Rectification [18 points]

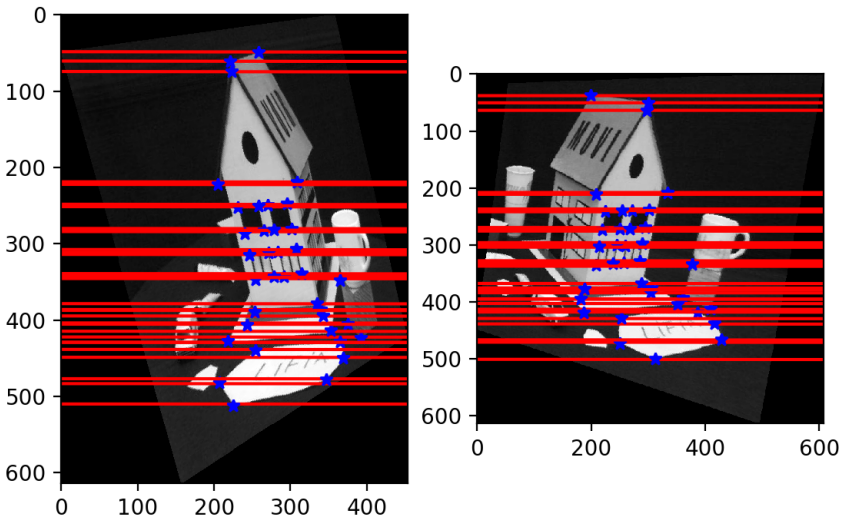
An interesting case for epipolar geometry occurs when two images are parallel to each other. In this case, there is no rotation component involved between the two images and the essential matrix is $\mathtt{E} = [T_x]R = [T_x]$. Also if you observe the epipolar lines l and l' for parallel images, they are horizontal and consequently, the corresponding epipolar lines share the same vertical coordinate. Therefore the process of making images parallel becomes useful while discerning the relationships between corresponding points in images.

Rectifying a pair of images can also be done for uncalibrated camera images (i.e. we do not require the camera matrix of intrinsic parameters). Using the fundamental matrix we can find the pair of epipolar lines l_i and l_i' for each of the correspondences. The intersection of these lines will give us the respective epipoles e and e'. Now to make the epipolar lines to be parallel we need to map the epipoles to infinity. Hence, we need to find a homography that maps the epipoles to infinity.

The method to find the homography has already been implemented for you. You can read more about the method used to estimate the homography in the paper "Theory and Practice of Projective Rectification" by Richard Hartley.



Using the `compute_epipole` function from the previous part and the given `compute_matching_homographies` function, find the rectified images and plot the parallel epipolar lines using the `plot_epipolar_lines` function from above. You need to do this for both the `matrix` and the `warrior` images. Below is an example of the output for this part:



In [ ]:
```
from math import floor, ceil

    def compute_matching_homographies(e2, F, im2, points1, points2):
        """This function computes the homographies to get the rectified images.

        input:
          e2 --> epipole in image 2
          F --> the fundamental matrix (think about what you should be passing: F or F.T!)
          im2 --> image2
          points1 --> corner points in image1
          points2 --> corresponding corner points in image2

        output:
          H1 --> homography for image 1
          H2 --> homography for image 2
        """
        # calculate H2
        width = im2.shape[1]
        height = im2.shape[0]

        T = np.identity(3)
        T[0][2] = -1.0 * width / 2
        T[1][2] = -1.0 * height / 2

        e = T.dot(e2)
        e1_prime = e[0]
        e2_prime = e[1]
        if e1_prime >= 0:
            alpha = 1.0
        else:
            alpha = -1.0
```

```
            R = np.identity(3)
            R[0][0] = alpha * e1_prime / np.sqrt(e1_prime**2 + e2_prime**2)
            R[0][1] = alpha * e2_prime / np.sqrt(e1_prime**2 + e2_prime**2)
            R[1][0] = - alpha * e2_prime / np.sqrt(e1_prime**2 + e2_prime**2)
            R[1][1] = alpha * e1_prime / np.sqrt(e1_prime**2 + e2_prime**2)

            f = R.dot(e)[0]
            G = np.identity(3)
            G[2][0] = - 1.0 / f

            H2 = np.linalg.inv(T).dot(G.dot(R.dot(T)))

            # calculate H1
            e_prime = np.zeros((3, 3))
            e_prime[0][1] = -e2[2]
            e_prime[0][2] = e2[1]
            e_prime[1][0] = e2[2]
            e_prime[1][2] = -e2[0]
            e_prime[2][0] = -e2[1]
            e_prime[2][1] = e2[0]

            v = np.array([1, 1, 1])
            M = e_prime.dot(F) + np.outer(e2, v)

            points1_hat = H2.dot(M.dot(points1.T)).T
            points2_hat = H2.dot(points2.T).T

            W = points1_hat / points1_hat[:, 2].reshape(-1, 1)
            b = (points2_hat / points2_hat[:, 2].reshape(-1, 1))[:, 0]

            # least square problem
            a1, a2, a3 = np.linalg.lstsq(W, b)[0]
            HA = np.identity(3)
            HA[0] = np.array([a1, a2, a3])

            H1 = HA.dot(H2).dot(M)
            return H1, H2
```

In [ ]:
```
def warp_image(image, H):
    """Returns an inverse warp of the image, given a homography.
    PASS IN THE HOMOGRAPHY FROM INPUT TO OUTPUT.

    Performs the warp of the full image content.
    Calculates bounding box by piping four corners through the transformation.
    """
    """ ==========
    YOUR CODE HERE
    ========== """
```

```
In [ ]: def image_rectification(im1, im2, points1, points2):
            """This function provides the rectified images along with the new corner points as
            images with corner correspondences

            input:
              im1 --> image1
              im2 --> image2
              points1 --> corner points in image1
              points2 --> corner points in image2

            output:
              rectified_im1 --> rectified image 1
              rectified_im2 --> rectified image 2
              new_cor1 --> new corners in the rectified image 1
              new_cor2 --> new corners in the rectified image 2
            """

            """ ==========
            YOUR CODE HERE
            ========== """


            return rectified_im1, rectified_im2, new_cor1, new_cor2
```

```
In [ ]: # Plot the parallel epipolar lines using plot_epipolar_lines
        for subj in ('matrix', 'warrior'):
            I1 = imread("./p4/%s/%s0.png" % (subj, subj))
            I2 = imread("./p4/%s/%s1.png" % (subj, subj))

            cor1 = np.load("./p4/%s/cor1.npy" % subj)
            cor2 = np.load("./p4/%s/cor2.npy" % subj)

            rectified_im1, rectified_im2, new_cor1, new_cor2 = image_rectification(I1, I2, cor
            plot_epipolar_lines(rectified_im1, rectified_im2, new_cor1, new_cor2)
```

1.6.1 Submission Instructions

Remember to submit a PDF version of this notebook to Gradescope. You can create a PDF via **File > Download as > PDF via LaTeX** (preferred, if possible), or by downloading as an HTML page and then "printing" the HTML page to a PDF (by opening the print dialog and then choosing the "Save as PDF" option).

```
In [ ]:
```