CSE 152A Homework 2 writeup

Zhizhen Yu

TOTAL POINTS

98 / 98

QUESTION 1

Edge detection and Corner detection

23 pts

1.1 Edge Detection **8 / 8**

✓ **- 0 pts** *Correct*

1.2 Corner Detection **15 / 15**

✓ **- 0 pts** *Correct*

QUESTION 2

Theory 20 pts

2.1 Epipolar Geometry **10 / 10**

✓ **- 0 pts** *Correct*

2.2 Epipolar Constraint **5 / 5**

✓ **- 0 pts** *Correct*

2.3 Essential Matrix **5 / 5**

✓ **- 0 pts** *Correct*

QUESTION 3

SSD (Sum Squared Distance) and NCC (Normalized Cross-Correlation) Matching 21 pts

3.1 SSD (Sum Squared Distance) Matching **5 / 5**

✓ **- 0 pts** *Correct*

3.2 NCC (Normalized Cross-Correlation) Matching **8 / 8**

✓ **- 0 pts** *Correct*

3.3 Naive Matching **8 / 8**

✓ **- 0 pts** *Correct*

QUESTION 4

Epipolar Geometry 34 pts

4.1 Fundamental Matrix **10 / 10**

✓ **- 0 pts** *Correct*

4.2 Epipoles **8 / 8**

✓ **- 0 pts** *Correct*

4.3 Plot Epipolar Lines **16 / 16**

✓ **- 0 pts** *Correct*

1.1 Edge Detection **8 / 8**

✓ **- 0 pts** *Correct*

1.2 Corner Detection **15 / 15**

✓ **- 0 pts** *Correct*

2.1 Epipolar Geometry **10 / 10**

✓ **- 0 pts** *Correct*

2.2 Epipolar Constraint **5 / 5**

✓ **- 0 pts** *Correct*

2.3 Essential Matrix **5 / 5**

✓ **- 0 pts** *Correct*

3.1 SSD (Sum Squared Distance) Matching **5 / 5**

✓ **- 0 pts** *Correct*

3.2 NCC (Normalized Cross-Correlation) Matching **8 / 8**

✓ **- 0 pts** *Correct*

3.3 Naive Matching **8 / 8**

✓ **- 0 pts** *Correct*

4.1 Fundamental Matrix **10 / 10**

✓ **- 0 pts** *Correct*

4.2 Epipoles **8 / 8**

✓ **- 0 pts** *Correct*

4.3 Plot Epipolar Lines **16 / 16**

✓ **- 0 pts** *Correct*

HW2

October 31, 2022

1 CSE 152A Fall 2022 – Assignment 2

- Assignment Published On: **Mon, Oct 17, 2022**
- Due On: **Sat, Oct 29, 2022 11:59 PM (Pacific Time)**

[189]:
```
# Setup
import numpy as np
from time import time
from skimage import io
%matplotlib inline
import matplotlib.pyplot as plt
```

1.1 Problem 1: Edge & Cornder Detection [23 pts]

1.1.1 Problem 1.1: Edge Detection [8 pts]

In this problem, you will write a function to perform edge detection. The following steps need to be implemented.

- **Smoothing [2 pt]:** First, we need to smooth the images to prevent noise from being considered edges. For this problem, use a 9x9 Gaussian kernel filter with $\sigma = 1.4$ to smooth the images.
- **Gradient Computation [3+3 pts]:** After you have finished smoothing, find the image gradient in the horizontal and vertical directions. Compute the gradient magnitude image as $|G| = \sqrt{G_x^2 + G_y^2}$ and gradient direction as $tan^{-1}(G_y/G_x)$.

Compute the images after each step. Show each of the intermediate steps and label your images accordingly.

In total, there should be four output images (original, smoothed, gradient magnitude, gradient direction).

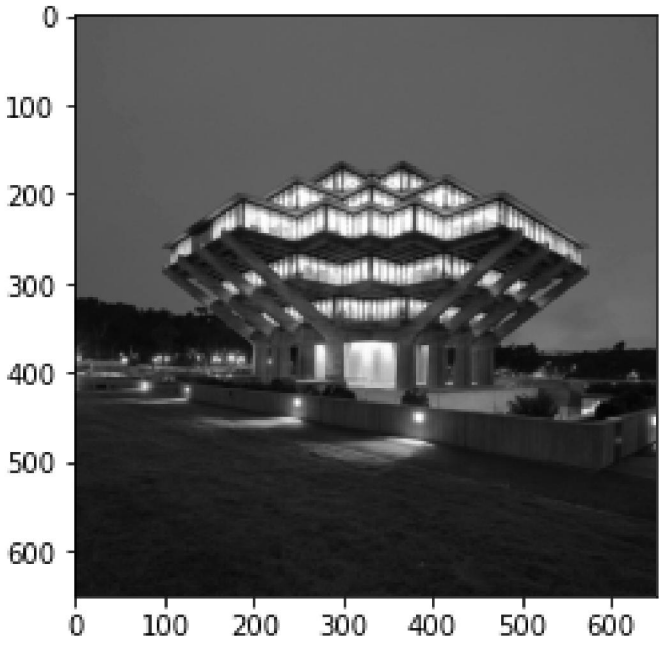
For this question, use the image `geisel.jpeg`.

[190]:
```
import numpy as np
from skimage import io
from collections import defaultdict
from scipy.signal import convolve
import matplotlib.pyplot as plt
```

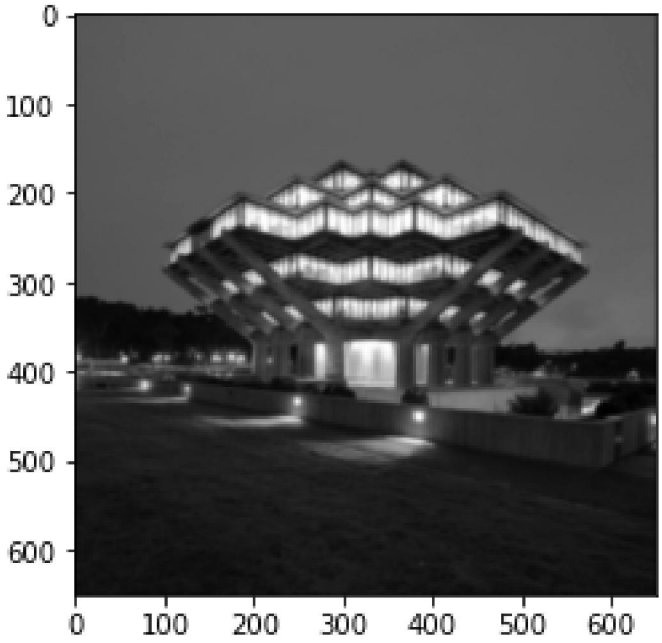
```
import scipy
import math
%matplotlib inline

def gaussian2d(filter_size=9, sig=1.0):
    """
    Creates 2D Gaussian kernel with side length `filter_size` and a sigma of␣
 ↪`sig`.
    Source: https://stackoverflow.com/a/43346070
    """
    ax = np.arange(-filter_size // 2 + 1., filter_size // 2 + 1.)
    xx, yy = np.meshgrid(ax, ax)
    kernel = np.exp(-0.5 * (np.square(xx) + np.square(yy)) / np.square(sig))
    return kernel / np.sum(kernel)


def smooth(image):
    smooth_image = np.zeros_like(image)

    ### YOUR CODE HERE
    kernal = gaussian2d(9, 1.4)
    smooth_image = scipy.signal.convolve(image, kernal, mode='same')
#     print(smoothed)

    ### END YOUR CODE

    return smooth_image

def gradient(image):

    g_mag = np.zeros_like(image)
    g_theta = np.zeros_like(image)

    ### YOUR CODE HERE
#     Gx= [[1, 0, -1],
#          [2, 0, -2],
#          [1, 0, -1]]
#     Gy = [[1,2,1],
#           [0,0,0],
#           [-1,-2,-1]]
    Gx= [[-1/2, 0, 1/2],
         [-1/2, 0, 1/2],
         [-1/2, 0, 1/2]]
    Gy = [[-1/2, -1/2, -1/2],
          [0,0,0],
          [1/2,1/2,1/2]]
    Gx = np.flip(Gx)
```

```
    Gy = np.flip(Gy)
    G_x = scipy.signal.convolve(image, Gx, mode='same')
    G_y = scipy.signal.convolve(image, Gy, mode='same')
    g_mag = np.sqrt(np.square(G_x) + np.square(G_y))
    g_theta = np.arctan2(G_y,G_x)
#     print(G_x.shape)

    ### END YOUR CODE


    return g_mag, g_theta

def edge_detect(image):
    """Perform edge detection on the image."""
    smoothed = smooth(image)
    g_mag, g_theta = gradient(smoothed)
    return smoothed, g_mag, g_theta

# Load image in grayscale
image = io.imread('geisel.jpeg', as_gray=True)
smoothed, g_mag, g_theta = edge_detect(image)

# print(image)


print('Original:')
plt.imshow(image, cmap='gray')
plt.show()

print('Smoothed:')
plt.imshow(smoothed, cmap='gray')
plt.show()

print('Gradient magnitude:')
plt.imshow(g_mag, cmap='gray')
plt.show()

print('Gradient direction:')
plt.imshow(g_theta, cmap='gray')
plt.show()
```

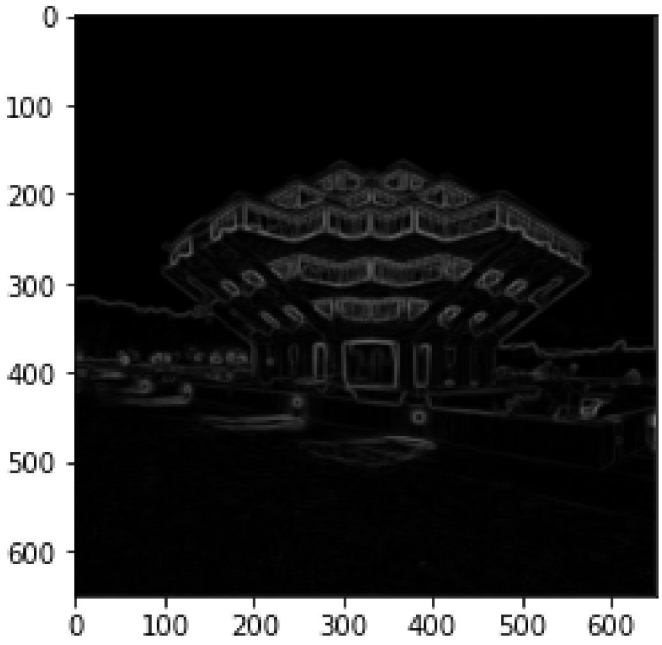
```
Original:
```



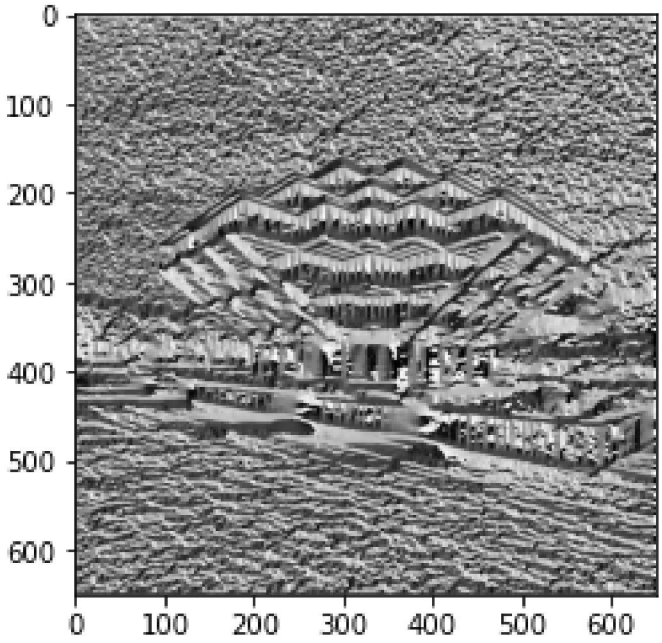
Smoothed:



Gradient magnitude:



Gradient direction:



1.1.2 Problem 1.2: Corner Detection [15 pts]

Next, you will implement a corner detector to detect photo-identifiable features in the image.

This should be done according to the easier method of looking at regions with significant value of the minimum eigenvalue. You should fill in the function corner_detect with inputs image, nCorners, smoothSTD, windowSize, where smoothSTD is the standard deviation of the smoothing kernel and windowSize is the window size for Gaussian smoothing, corner detection and non-maximum suppression. Instead of using a hard threshold, return the nCorners strongest corners after non-maximum suppression. This way you can control exactly how many corners are returned. Your function should also return the matrix of minimum eigen values that you computed.

For each image, detect 100 corners with a Gaussian standard deviation of 2.0 and a window size of 13. Display the corners using the show_corners_result function and plot the minimum eigen value images using the show_eigen_images function.

For this question, we will use images `almond0.jpg` and `almond1.jpg`.

[191]:
```
import numpy as np
import matplotlib.pyplot as plt
import imageio
```

[192]:
```
def rgb2gray(rgb):
    """ Convert rgb image to grayscale.
    """
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])
```

[201]:
```
def corner_detect(image, nCorners, smoothSTD, windowSize):
    """Detect corners on a given image.

    Args:
        image: Given a grayscale image on which to detect corners.
        nCorners: Total number of corners to be extracted.
        smoothSTD: Standard deviation of the Gaussian smoothing kernel.
        windowSize: Window size for Gaussian smoothing kernel, corner detector,␣
 ↪and non maximum suppression.

    Returns:
        Detected corners (in image coordinate) in a numpy array (n*2).
        The minor eigen value image having the same shape as the image
    """



    corners = []
    minor_eig_image = np.zeros_like(image)

    ### YOUR CODE
    half = (int) (windowSize/2)
```

```
    img_padded = np.pad(image, [
        (half, half),
        (half, half)
    ], mode='edge')
#     Gy_padded = np.pad(G_y, half)

    kernal = gaussian2d(windowSize, smoothSTD)
    img_padded = scipy.signal.convolve(img_padded, kernal, mode='same')

    Gx= [[-1/2, 0, 1/2],
        [-1/2, 0, 1/2],
        [-1/2, 0, 1/2]]
    Gy = [[-1/2, -1/2, -1/2],
         [0,0,0],
         [1/2,1/2,1/2]]
    Gx = np.flip(Gx)
    Gy = np.flip(Gy)
    G_x = scipy.signal.convolve(img_padded, Gx, mode='same')
    G_y = scipy.signal.convolve(img_padded, Gy, mode='same')


#     for i in range(half, image.shape[0]+half):
#         for j in range(half, image.shape[1]+half):
#             Gx_win = Gx_padded[i-half:i+half+1, j-half:j+half+1]
#             Gy_win = Gy_padded[i-half:i+half+1, j-half:j+half+1]
#             C =np.array([[np.sum(np.square(Gx_win)), np.sum(Gx_win*Gy_win)],
#                 [np.sum(Gx_win*Gy_win), np.sum(np.square(Gy_win))]])

#             minor_eig_image[i-half, j-half] = min(np.linalg.eigvals(C))
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            Gx_win = G_x[i:i+windowSize, j:j+windowSize]
            Gy_win = G_y[i:i+windowSize, j:j+windowSize]
            C =np.array([[np.sum(Gx_win*Gx_win), np.sum(Gx_win*Gy_win)],
                [np.sum(Gx_win*Gy_win), np.sum(Gy_win*Gy_win)]])
            eigenv = min(np.linalg.eigvals(C))
            if eigenv > 250000:
                continue
            minor_eig_image[i, j] = eigenv

#     for i in range(0, image.shape[0]):
#         for j in range(0, image.shape[1]):
#             start_i, end_i = i-half, i+half
#             start_j, end_j = j-half, j+half
#             start_i = 0 if start_i <0 else start_i
#             start_j = 0 if start_j <0 else start_j
#             end_i = image.shape[0]-1 if end_i > image.shape[0] else end_i
```

```
#             end_j = image.shape[1]-1 if end_j > image.shape[1] else end_j

#             ix_m = G_x[start_i: end_i, start_j: end_j]
#             iy_m = G_y[start_i: end_i, start_j: end_j]
#             ix2 = np.sum(ix_m * ix_m)
#             iy2 = np.sum(iy_m * iy_m)
#             ixy = np.sum(ix_m * iy_m)

#             C = np.array([[ix2, ixy], [ixy, iy2]])

#             minor_eig_image[i, j] = min(np.linalg.eigvals(C)) #unfin

    #minor_eig_image = unsorted
    #corners : divide into cells, pick the largest, and sort to find the 100␣
 ↪max corners.

            #create a window, while find a max
            #loop through the window
            #if the pixel matches the max,
    cornerNMS = []


    for i in range(half,image.shape[0]-windowSize,windowSize):
        for j in range(half,image.shape[1]-windowSize,windowSize):

            start_i, end_i = i, i+windowSize
            start_j, end_j = j, j+windowSize
            end_i = image.shape[0] if end_i > image.shape[0] else end_i
            end_j = image.shape[1] if end_j > image.shape[1] else end_j

            grid = minor_eig_image[start_i: end_i, start_j: end_j]
            maximum = np.max(grid)
            for k in range(0,grid.shape[0]):
                for l in range(0, grid.shape[1]):
                    if grid[k][l] == maximum:
                        cornerNMS.append((maximum, (start_j+l, start_i+k)))
                        break
                break

#     corners = np.array([i[1] for i in cornerNMS][:nCorners])

    cornerNMS = np.array(cornerNMS)
#     print(cornerNMS.shape)
    cornerSorted=sorted(cornerNMS,key=lambda x: x[0], reverse=True)

    corners = np.zeros((nCorners, 2), dtype='int32')
    for i in range(0, nCorners):
```

```
        corners[i, :] = cornerSorted[i][1]
#         print(cornerNMS[i][1])

    #Check
    return corners, minor_eig_image
```

[202]:
```
def show_eigen_images(imgs):
    print("Minor Eigen value images")
    fig = plt.figure(figsize=(16, 16))
    # Plot image 1
    plt.subplot(1,2,1)
    plt.imshow(imgs[0], cmap='gray')
    plt.title('almond 1')

    # Plot image 2
    plt.subplot(1,2,2)
    plt.imshow(imgs[1], cmap='gray')
    plt.title('almond 2')

    plt.show()

def show_corners_result(imgs, corners):
    print("Detected Corners")
    fig = plt.figure(figsize=(16, 16))
    ax1 = fig.add_subplot(221)
    ax1.imshow(imgs[0], cmap='gray')
    ax1.scatter(corners[0][:, 0], corners[0][:, 1], s=35, edgecolors='r',␣
 ↪facecolors='none')

    ax2 = fig.add_subplot(222)
    ax2.imshow(imgs[1], cmap='gray')
    ax2.scatter(corners[1][:, 0], corners[1][:, 1], s=35, edgecolors='r',␣
 ↪facecolors='none')
    plt.show()
```

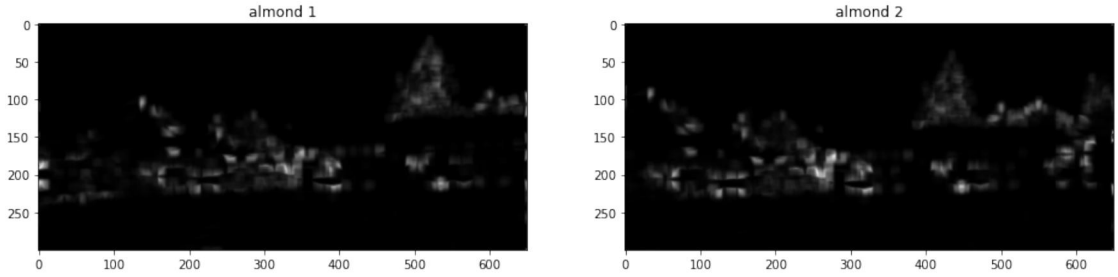
[203]:
```
# detect corners on the two provided images
# adjust your corner detection parameters here
nCorners = 100
smoothSTD = 2
windowSize = 13

# read images and detect corners on images
imgs = []
eig_imgs = []
corners = []
for i in range(2):
    img = io.imread('almond' + str(i) + '.jpg')
```

```
    imgs.append(rgb2gray(img))
    corners_vals, minor_eig_image = corner_detect(imgs[-1], nCorners,␣
↪smoothSTD, windowSize)
    eig_imgs.append(minor_eig_image)
    corners.append(corners_vals)
```

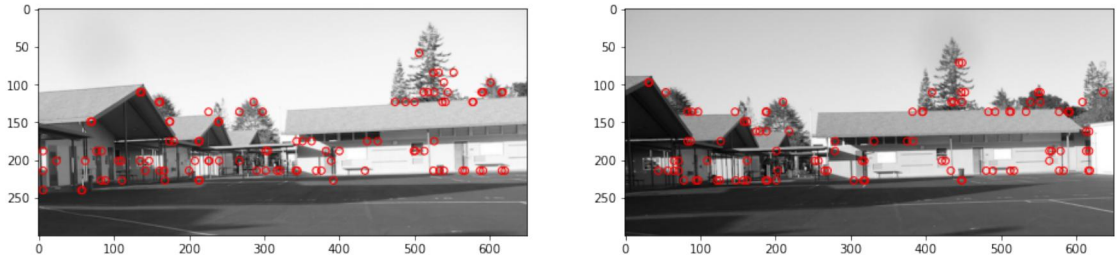
```
/var/folders/39/npycxr3x43ngh80h1wlwmj540000gn/T/ipykernel_1052/1905249500.py:10
9: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences
(which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths
or shapes) is deprecated. If you meant to do this, you must specify
'dtype=object' when creating the ndarray.
  cornerNMS = np.array(cornerNMS)
```

[204]:
```
show_eigen_images(eig_imgs)
show_corners_result(imgs, corners)
```

```
Minor Eigen value images
```




```
Detected Corners
```

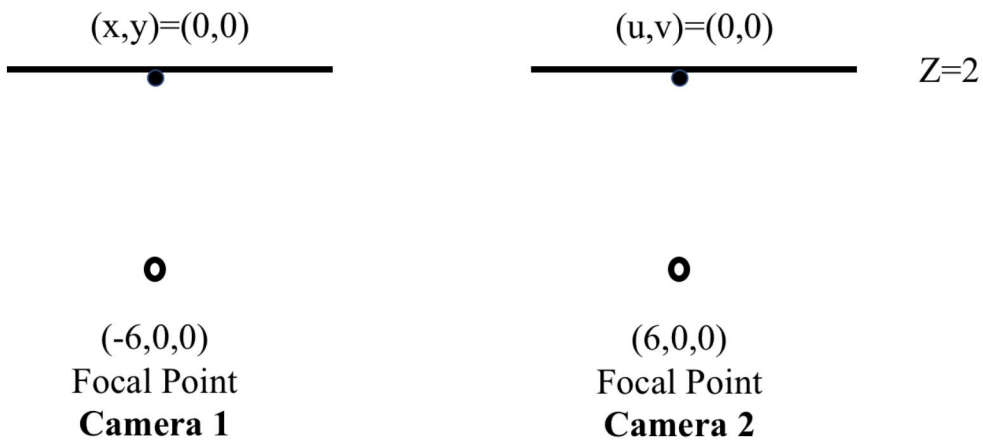


1.2 Problem 2: Theory [20 points]

1.2.1 Problem 2.1: 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

(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).



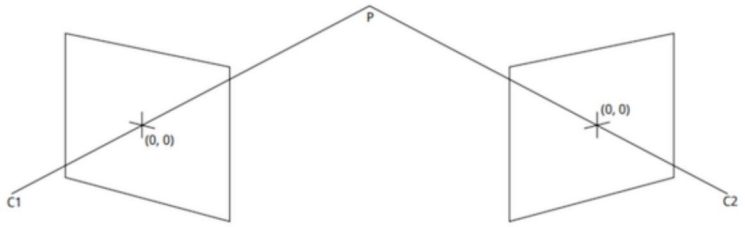
Suppose the point (x, y) = (3, 3) is matched to the point (u, v) = (2, 3). What is the 3D location of this point?

[197]:
```
#This means these two pixels are the projections of the same 3D point in the␣
 ↪world coordinate.
import numpy as np
Z=2
p1 = [3-6,3+0,Z]
p2=[2+6,3+0, Z]

f1 = [-6,0,0]
f2 = [6,0,0]

#compute the lines, or camera rays, connecting the focal ponts and the points
lineRation1 = np.subtract(p1,f1)
lineRation2 = np.subtract(p2,f2)
    # print(lineRation1, lineRation2)

#Line1 = (x+3)/3 = (y-3)/3=(z-2)/2
#Line2 = (x-8)/2 = (y-3)/3=(z-2)/2
# computing the intersection of two lines by Line1 = Line2
# 3x-24 = 2x+6 ---> x=30
# (30+3)/3 = (y-3)/3 ---> y=36
# (30+3)/3 = (z-2)/2 ---> z=24

Final_answer = (30,36,24)
Final_answer
```

[197]: (30, 36, 24)

1.2.2 Problem 2.2: 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.

[198]:
```
#Answer:
# F is given by x'.T * F *x = 0
# x' = x = (0, 0, 1)

# Expanding the linear equation:
# 0+0+0+0+0+0+0+0+0+f_33 = 0
# --> f_33 = 0
```

1.2.3 Problem 2.3: 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).

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

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

[227]:
```
import numpy as np
import math
t = np.array([[2],[5],[1]])
```

```
print("Version one: citation: wikepedia E = R *tx")
tx = [[0, -1,5],[1,0,-2],[-5,2,0]]
R = [[math.sqrt(2)/2,-math.sqrt(2)/2,0],
     [math.sqrt(2)/2,math.sqrt(2)/2,0],
     [0,0,1]]
E = np.matmul(R,tx)
print("E:", E)

# E1 = np.cross(t,R)
# print("E1:", E1)
#

U,S,V = np.linalg.svd(E)
# print("S:", S)
# S = [5.47722558e+00 5.47722558e+00 1.41503082e-16]
# 1.41503082e-16 <=1 -15
# so rank = 3-1 = 2
rank = 2
print("rank:", rank)

print("Version two: by Professor's slides")
E = np.matmul(tx, R)
print("E:", E)

# E1 = np.cross(t,R)
# print("E1:", E1)
#

U,S,V = np.linalg.svd(E)
# print("S:", S)
# S = [5.47722558e+00 5.47722558e+00 3.64131468e-17]
# 3.64131468e-17 <=1 -15
# so rank = 3-1 = 2
rank = 2
print("rank:", rank)
```

```
Version one: citation: wikepedia E = R *tx
E: [[-0.70710678 -0.70710678  4.94974747]
 [ 0.70710678 -0.70710678  2.12132034]
 [-5.          2.          0.        ]]
rank: 2
Version two: by Professor's slides
E: [[-0.70710678 -0.70710678  5.        ]
 [ 0.70710678 -0.70710678 -2.        ]
 [-2.12132034  4.94974747  0.        ]]
rank: 2
```

1.3 Problem 3: 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.

1.3.1 Problem 3.1: SSD (Sum Squared Distance) Matching [5 points]

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

[200]:
```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import math
import imageio
import pickle
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Input In [200], in <cell line: 2>()
      1 import numpy as np
----> 2 import cv2
      3 import matplotlib.pyplot as plt
      4 import math

ModuleNotFoundError: No module named 'cv2'
```

[205]:
```
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
    ========== """
    #for the image window
    # get the magnitude each pixel in img1 - each pixel in img2 and square
```

```
    # add to the sum

    matching_score = 0
#     print(c1)
    for i,i2 in zip(range(c1[0]-R, c1[0]+R+1), range(c2[0]-R, c2[0]+R+1)):
        for j,j2 in zip(range(c1[1]-R,c1[1]+R+1), range(c2[1]-R, c2[1]+R+1)):
            if (j >= img1.shape[0] or i >= img1.shape[1]):
                img1 = np.pad(img1, R)
            if (j2 >= img2.shape[0] or i2 >= img2.shape[1]):
                img2=np.pad(img2, R)

            mag = pow(abs(img1[j][i]-img2[j2][i2]),2)
            matching_score += mag

    return matching_score
```

[206]:
```
# 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
```

```
20
30
46
```

1.3.2 Problem 3.2: 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.

[207]:
```
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
    ========== """
    matching_score = 0
    mean = 0
    mean2 =0
    std = 0
    std2=0

#     for i,i2 in zip(range(c1[0]-R, c1[0]+R+1), range(c2[0]-R, c2[0]+R+1)):
#         for j,j2 in zip(range(c1[1]-R,c1[1]+R+1), range(c2[1]-R, c2[1]+R+1)):
#             mean +=img1[j][i]
#             mean2 +=img2[j2][i2]
#     mean = mean/(2*R+1)
#     mean2 = mean2/(2*R+1)


#     img11 = img1[c1[1]-R:c1[1]+R+1, c1[0]-R:c1[0]+R+1]
#     img22=img2[c2[1]-R:c2[1]+R+1, c2[0]-R: c2[0]+R+1]

#     img1 = np.pad(img1, 1)
#     img2=np.pad(img2, 1)

    img11 = img1[c1[1]-R:c1[1]+R+1, c1[0]-R:c1[0]+R+1]
    img22=img2[c2[1]-R:c2[1]+R+1, c2[0]-R:c2[0]+R+1]

    mean = np.mean(img11)
    mean2 = np.mean(img22)

#             mag = pow(abs(img1[i][j]-img2[i2][j2]),2)
#             matching_score += mag
    for i,i2 in zip(range(c1[1]-R, c1[1]+R+1), range(c2[1]-R, c2[1]+R+1)):
        for j,j2 in zip(range(c1[0]-R,c1[0]+R+1), range(c2[0]-R, c2[0]+R+1)):
            tmp = (img1[i][j]-mean) ** 2
            std += tmp
            tmp2 = (img2[i2][j2]-mean2) ** 2
            std2 +=tmp2

    std = math.sqrt(std)
    std2 = math.sqrt(std2)


#     for i,i2 in zip(range(c1[0]-R, c1[0]+R+1), range(c2[0]-R, c2[0]+R+1)):
```

```
#         for j,j2 in zip(range(c1[1]-R,c1[1]+R+1), range(c2[1]-R, c2[1]+R+1)):
# #     for i in range(img11.shape[0]):
# #         for j in range(img11.shape[1]):
#             matching_score += ((img1[j][i]-mean)/std)*((img2[j2][i2]-mean2)/
 ↪std2)

#     std = img11.std()
#     std2 = img22.std()

    img1_norm = (img11-mean)/std
    img2_norm = (img22-mean2)/std2
    matching_score = np.sum(img1_norm*img2_norm)


    return matching_score
```

[208]:
```
# 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
```

```
0.8546547739343037
0.8457615282174419
0.6258689611426174
```

1.3.3 Problem 3.3: 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.

[220]:
```
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
    ========== """
    matching = []
    for i in range(corners1.shape[0]):
        max1 = None
        pair = None
        for j in range(corners2.shape[0]):
            score = nccMatch(img1, img2, corners1[i], corners2[j], R)
            if(max1 is None) or (score > max1):
                max1 = score
                pair = (corners1[i], corners2[j])


#         score = nccMatch(img1, img2, corners1[i], corners2[i], R)
#         pair = (corners1[i], corners2[i])
        if max1 >= NCCth:
            matching.append(pair)

    #Check
    return matching
```

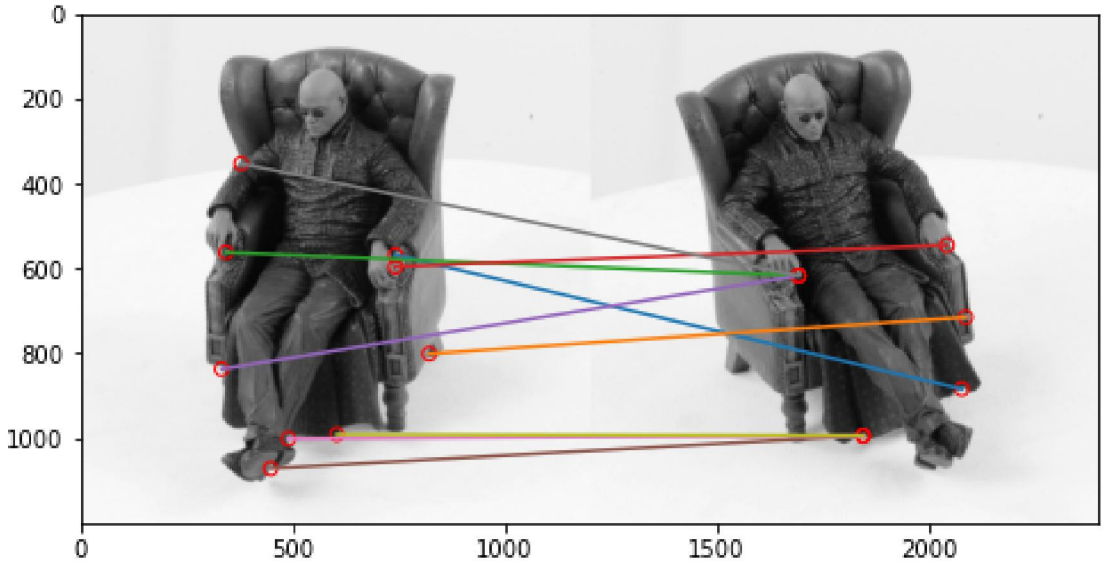
[221]:
```
def rgb2gray(rgb):
    """ Convert rgb image to grayscale.
    """
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])
# detect corners on warrior and matrix sets
# you are free to modify code here, create your helper functions, etc.

nCorners = 20
smoothSTD = 1
windowSize = 17

# read images and detect corners on images
#
imgs_mat = []
```

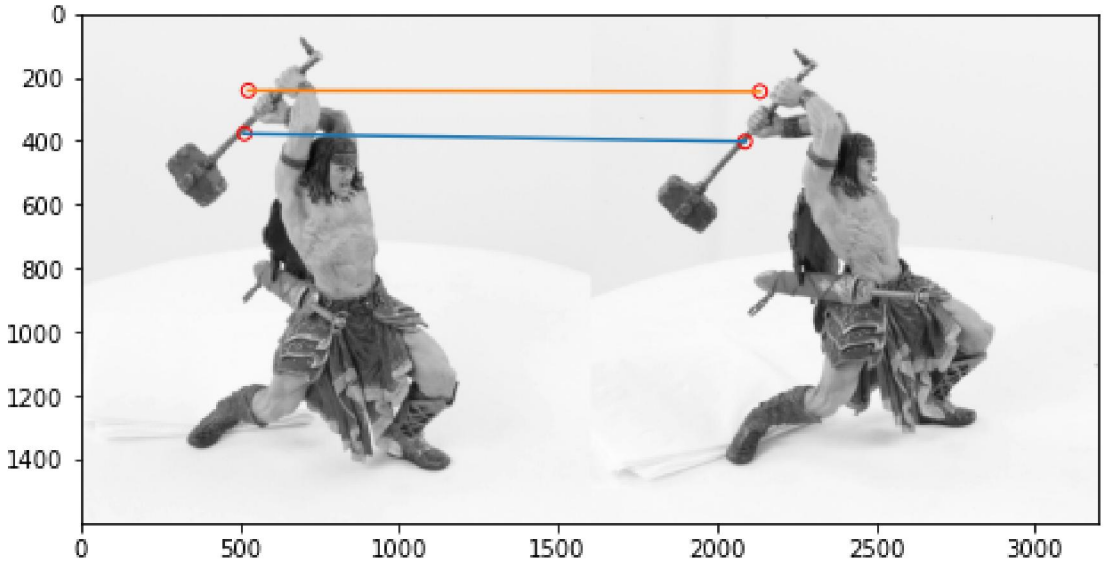
```
crns_mat = []
imgs_war = []
crns_war = []

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))
```

[222]:
```
import pickle
# 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[1], R, NCCth)
matching_war = naive_matching(imgs_war[0]/255, imgs_war[1]/255, crns_war[0],␣
 ↪crns_war[1], R, NCCth)
```

[223]:
```
# 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␣
 ↪different sizes, resize one before use
    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='none')
        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)
```

```
Number of Corners: 20
```





1.4 Problem 4: 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 explore how to visualize epipolar geometry constraint in the form of epipolar lines. Although it is outside the scope of this assignment, we can further use this constraint to rectify the images to build a better matching algorithm.

1.4.1 Problem 4.1: 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.

[213]:
```
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
    ========== """
#     A =  [x2[0,:].T*x1[0,:].T, x2[0,:].T*x1[1,:].T, x2[0,:].T*x1[2,:].T,
#             x2[1,:].T*x1[0,:].T, x2[1,:].T*x1[1,:].T, x2[1,:].T*x1[2,:].T,
#             x2[2,:].T*x1[0,:].T, x2[2,:].T*x1[1,:].T, x2[2,:].T*x1[2,:].T ]
    A = np.zeros((x1.shape[1],9))
    for i in range(x1.shape[1]):
        A[i] = [x1[0,i]*x2[0,i], x1[0,i]*x2[1,i], x1[0,i]*x2[2,i],
                x1[1,i]*x2[0,i], x1[1,i]*x2[1,i], x1[1,i]*x2[2,i],
                x1[2,i]*x2[0,i], x1[2,i]*x2[1,i], x1[2,i]*x2[2,i] ]
          #(n, 9), np.ones((n,1))

    U,S,V = np.linalg.svd(A)

    F = np.reshape(V[-1],(3,3))
    # 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]
```

[214]:
```
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]
```

[215]:
```
# 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]]
```

```
[[ 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]]
```

1.4.2 Problem 4.2: Epipoles [8 points]

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

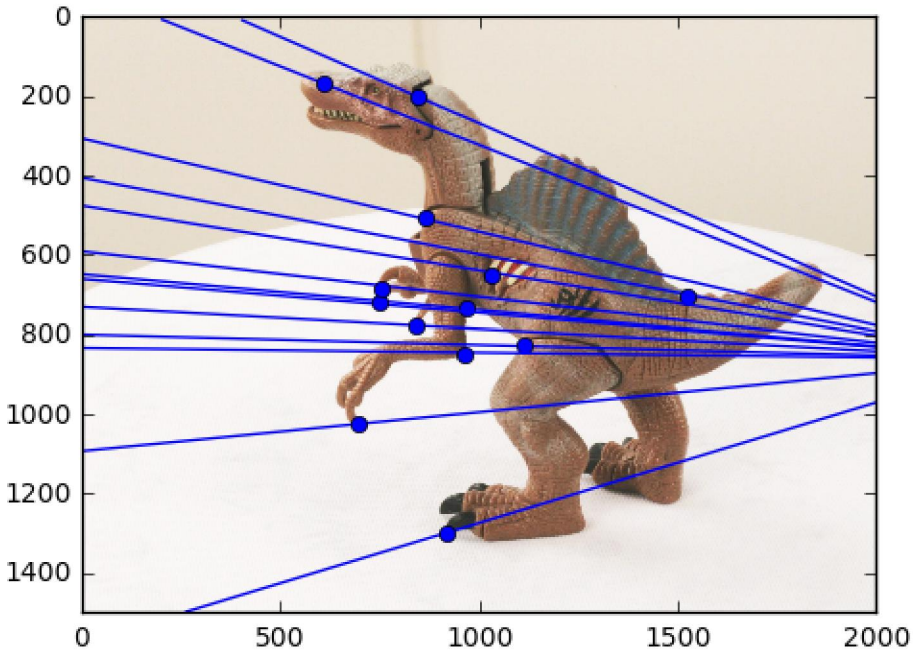
[216]:
```
def compute_epipole(F):
    """
    This function computes the epipoles for a given fundamental matrix.

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

    """ ==========
    YOUR CODE HERE
    ========== """
    U,S,V1 = svd(F.T)
    U,S,V2 = svd(F)
    e1 = V1[-1]
    e2 = V2[-1]
    #convert into cardesian coordinates
    e1 = e1[:]/e1[:][-1]
    e2 = e2[:]/e2[:][-1]


    return e1,e2
```

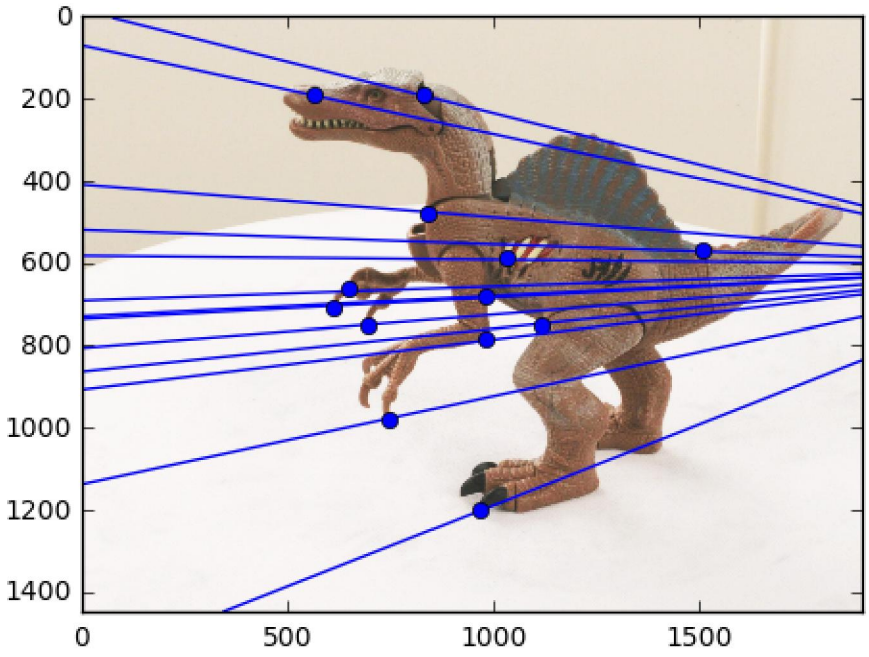
[217]:
```
# Here is the code for you to test your implementation
F_test = np.array([[1, 2, 1], [6, 5, 4], [9, 8, 1]])
print(compute_epipole(F_test))
# should print
#(array([-65.3659783 ,  15.85984739,   1.        ]),
#array([-41.86658577,  46.87378417,   1.        ]))
```

```
(array([-65.3659783 ,  15.85984739,   1.        ]), array([-41.86658577,
46.87378417,   1.        ]))
```

1.4.3 Problem 4.3: 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.



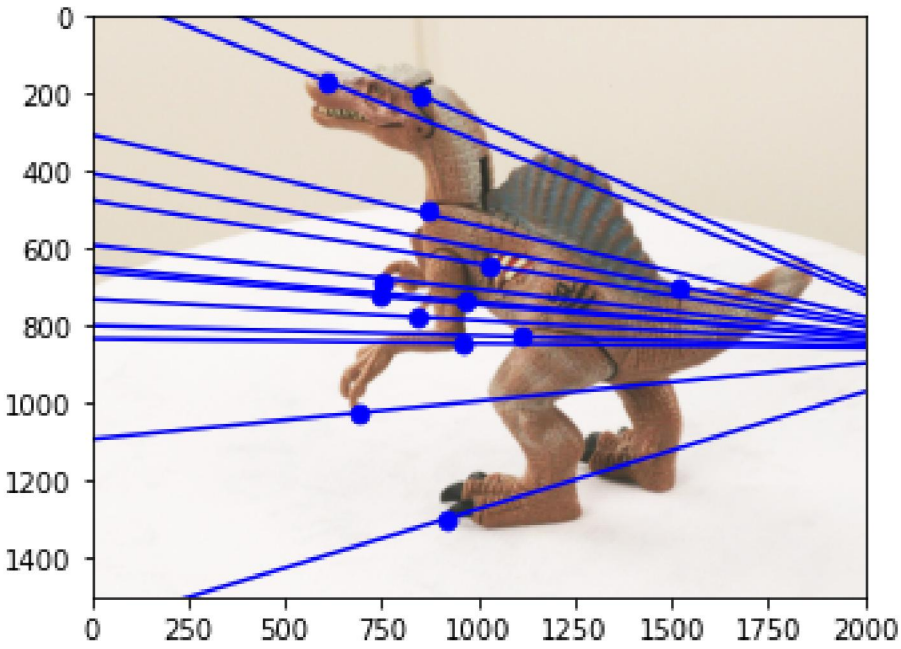


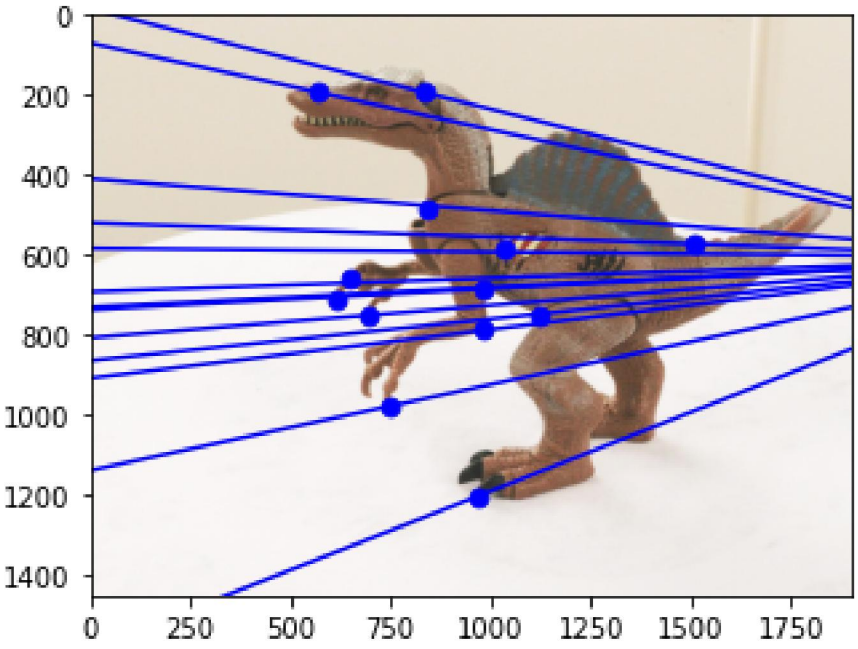
[218]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm


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

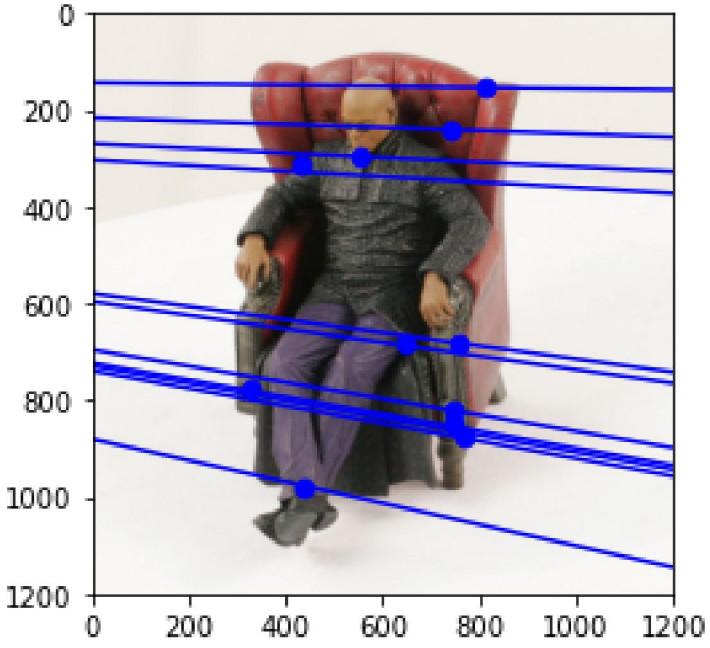
```
    Args:
        img1: Image 1.
        img2: Image 2.
        cor1: Corners in homogeneous image coordinate in image 1 (3xn)
        cor2: Corners in homogeneous image coordinate in image 2 (3xn)

    """


    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
    ========== """


    fig = plt.figure()
    plt.imshow(img1, cmap = 'gray')
    for i in range(cor1.shape[1]):
        # Corners have last coordinate=1
        plt.scatter(cor1[0][i], cor1[1][i], s=45, edgecolors='b',␣
↪facecolors='b')
    for i in range(cor2.shape[1]):

        # e2--> e --> x ---> l=Fx
        epiL_img2 = np.dot(F,cor2[:,i])
        # resulting line vector from (Fx) representing ax + by + c = 0

        # connect to find the epipolar line
        # Using the equation Ax+By+C = 0
        # y = (-C-Ax) / B
        x1, x2 = 0, img1.shape[1]
        y1 = (-epiL_img2[2])/(epiL_img2[1])
        y2 = (x2*epiL_img2[0]+epiL_img2[2])/(-epiL_img2[1])

        plt.plot([x1, x2], [y1, y2], color = 'b')
        plt.axis([0,img1.shape[1],img1.shape[0],0])
    plt.show()

    plt.imshow(img2, cmap = 'gray')
    for i in range(cor2.shape[1]):
        # Corners have last coordinate=1
```

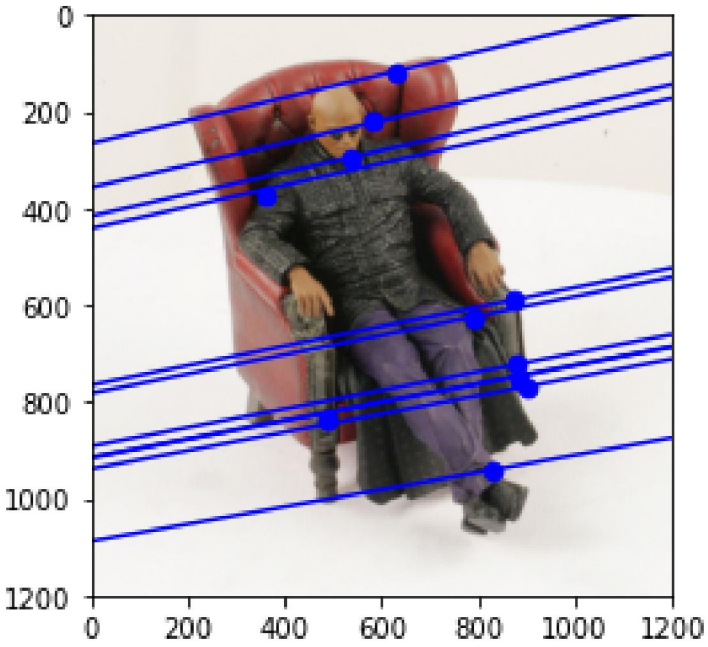
```
        plt.scatter(cor2[0][i], cor2[1][i], s=45, edgecolors='b',␣
↪facecolors='b')
    for i in range(cor1.shape[1]):
        epiL_img1 = np.dot(F.T,cor1[:,i])
        # resulting line vector from (Fx) representing ax + by + c = 0

        # connect to find the epipolar line
        # Using the equation Ax+By+C = 0
        # y = (-C-Ax) / B
        x1, x2 = 0, img2.shape[1]
        y1 = (-epiL_img1[2])/(epiL_img1[1])
        y2 = (x2*epiL_img1[0]+epiL_img1[2])/(-epiL_img1[1])

        plt.plot([x1, x2], [y1, y2], color = 'b')
        plt.axis([0,img2.shape[1],img2.shape[0],0])
    plt.show()
```

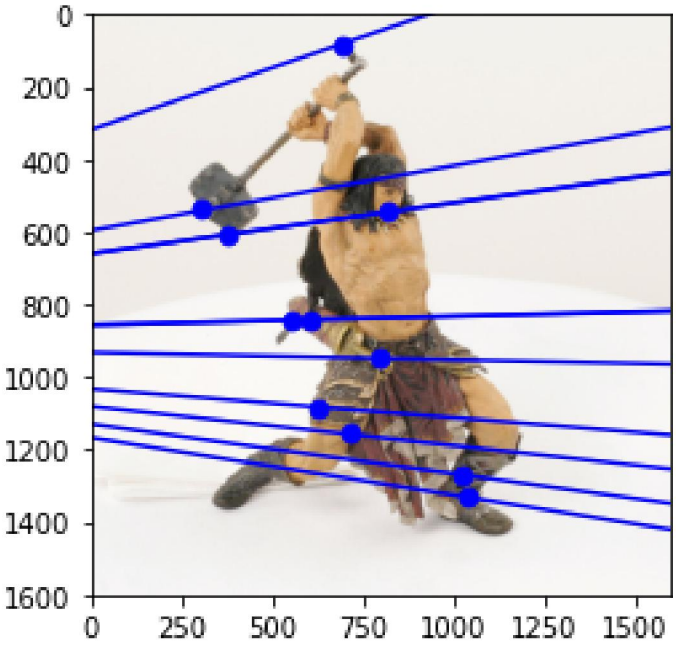
[219]:
```
# replace images and corners with those of matrix and warrior
imgids = ["dino", "matrix", "warrior"]
for imgid in imgids:
    I1 = imageio.imread("./p4/"+imgid+"/"+imgid+"0.png")
    I2 = imageio.imread("./p4/"+imgid+"/"+imgid+"1.png")
    cor1 = np.load("./p4/"+imgid+"/cor1.npy")
    cor2 = np.load("./p4/"+imgid+"/cor2.npy")
    plot_epipolar_lines(I1,I2,cor1,cor2)
```

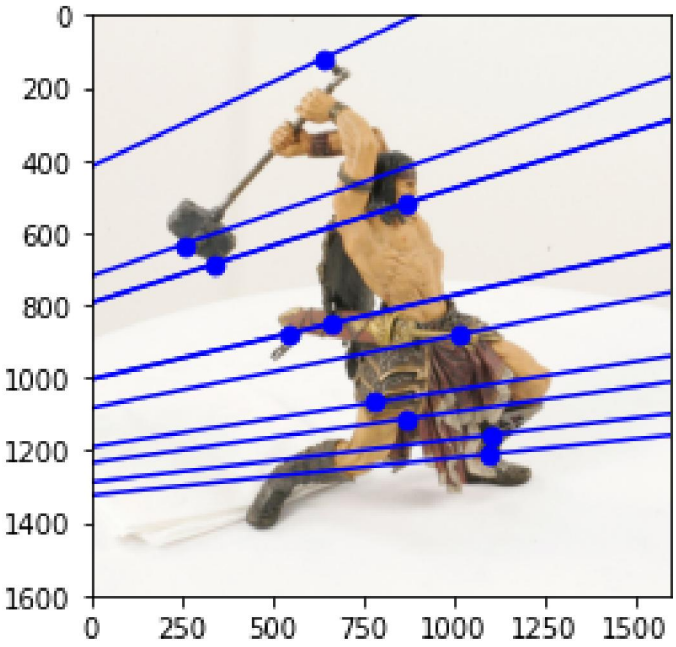



0
200
400
600
800
1000
1200
1400
0
250
500
750
1000
1250
1500
1750


0
200
400
600
800
1000
1200
0
200
400
600
800
1000
1200


0
200
400
600
800
1000
1200
0
200
400
600
800
1000
1200


0
200
400
600
800
1000
1200
1400
1600
0
250
500
750
1000
1250
1500


0
200
400
600
800
1000
1200
1400
1600
0
250
500
750
1000
1250
1500