CSE 152A Winter 2023 – Assignment 2

- Assignment Published On: **Tuesday, October 24, 2023**
- Due On: **Saturday, November 4 11:59 PM (Pacific Time)**

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 posted on lecture slides.
- 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 open source codes 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.
- Make sure that you read hints for questions (wherever given).

Late Policy: Assignments submitted late will receive a 25% grade reduction for each 12 hours late (that is, 50% per day).

In [ ]:
```
# Setup
import numpy as np
from time import time
from skimage import io
%matplotlib inline
import matplotlib.pyplot as plt
import imageio
import cv2
import math
import pickle
```

Problem 1: Edge & Corner Detection [23 pts]

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`.

!! Hint !!

G_x and G_y are the image gradients along x and y directions respectively. To get the image gradients, it is recommended that you use convolutions(similar to the approach that you used in HW1). Learn about them here: https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.convolve.html

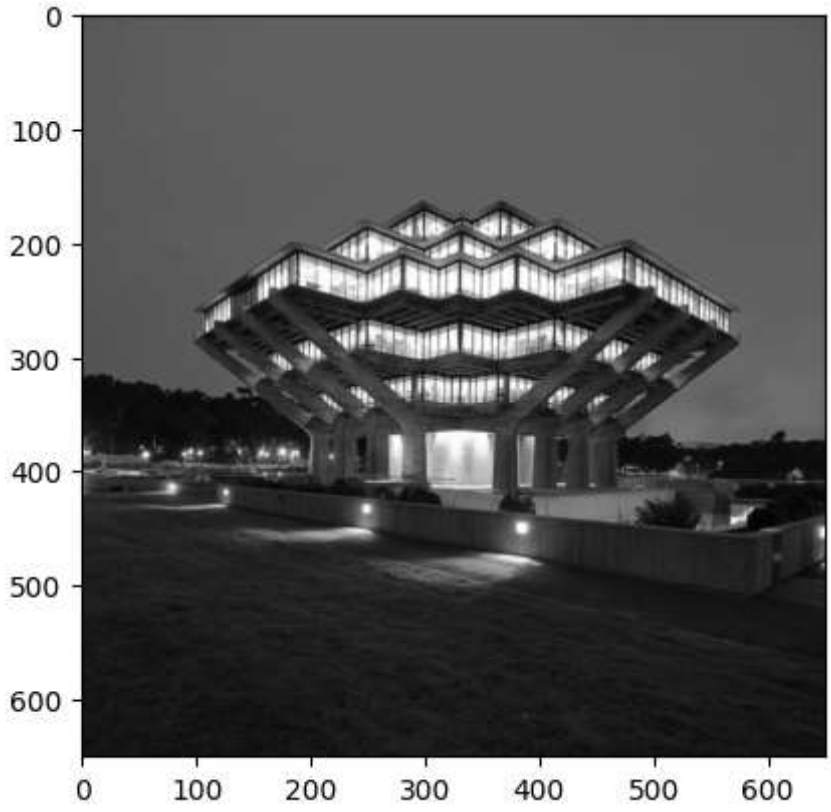
Convolution can also be used to smooth the image.

For gradient direction, try utilizing the arctan2 function to obtain the angle: https://numpy.org/doc/stable/reference/generated/numpy.arctan2.html

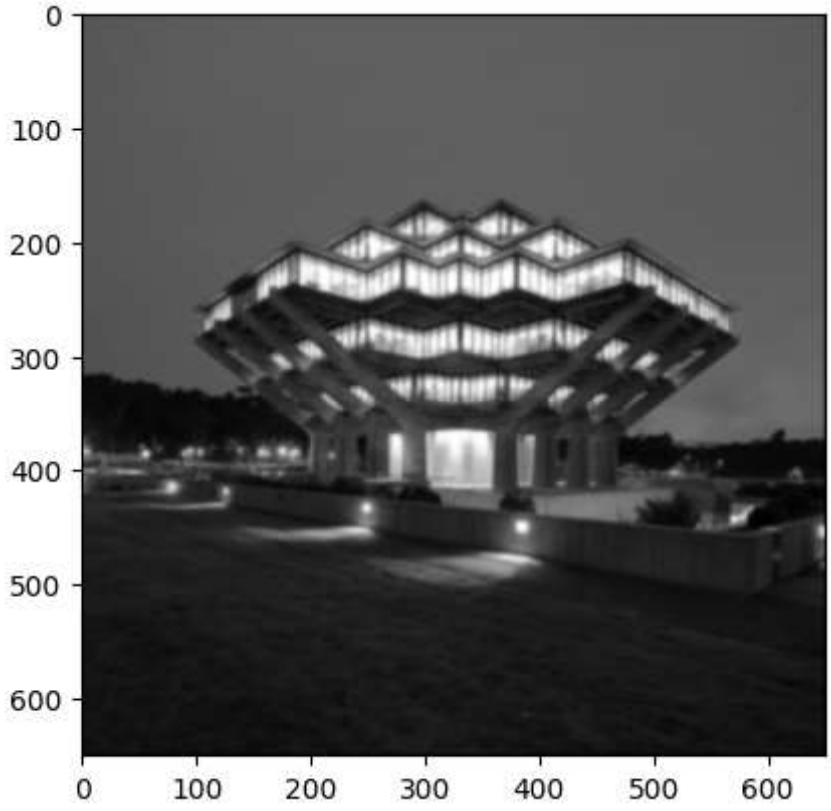
In [ ]:
```python
import numpy as np
from skimage import io
from collections import defaultdict
from scipy.signal import convolve
import matplotlib.pyplot as plt
%matplotlib inline

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):
    """
    Args:
        image: input image (h, w)

    Returns:
        smooth_image: smoothened version of the input image (h, w)
    """
```

```
    ### YOUR CODE HERE
    kernel = gaussian2d(9, 1.4)
    smooth_image = convolve(image, kernel, mode='same')
    ### END YOUR CODE

    return smooth_image

def gradient(image):
    """
    Args:
        image: input image (h, w)

    Returns:
        g_mag: gradient magnitude (h, w)
        g_theta: gradient direction (h, w)
    """

    ### YOUR CODE HERE
    kernel_x = np.array([[0, 0, 0], [1, 0, -1], [0, 0, 0]]) * 0.5
    kernel_y = np.array([[0, 1, 0], [0, 0, 0], [0, -1, 0]]) * 0.5

    gx = convolve(image, kernel_x)
    gy = convolve(image, kernel_y)

    g_mag = np.sqrt(gx**2 + gy**2)

    g_theta = np.arctan2(gy,gx)
    ### 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('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()
```

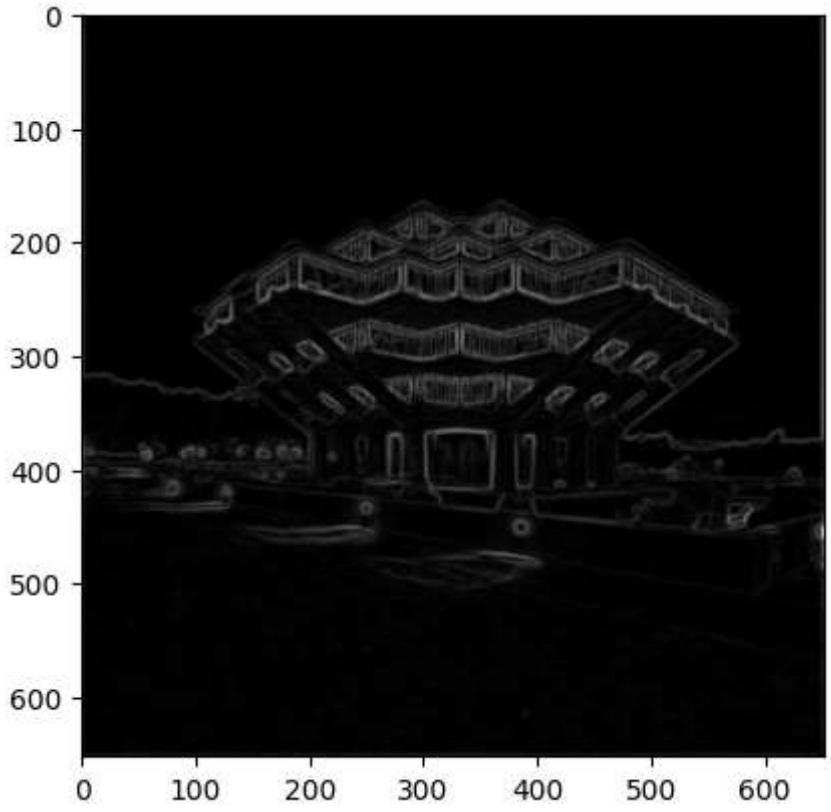
```
Original:
```



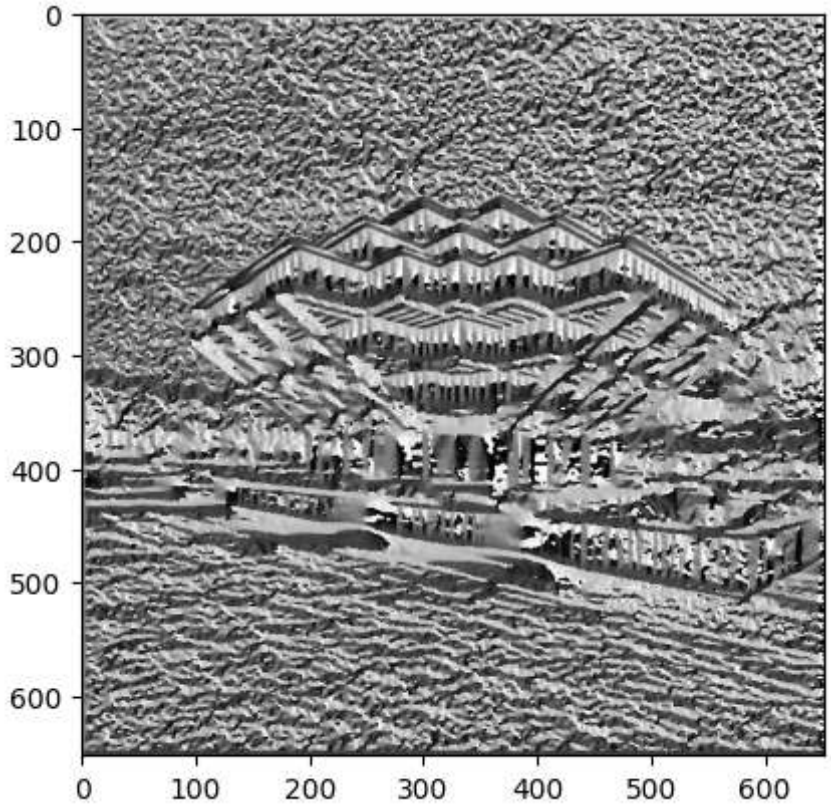
Smoothed:



Gradient magnitude:



Gradient direction:



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`.

!! Hint !!

A window in an image is usually represented by its center pixel. So in the example below, the window size = 3, image shape = 6 x 6 and the shown window correponds to the pixel location (1,1) or i5. Think about how you need to iterate over the image dimensions to get the different windows of an image. **Be wary of the image boundaries.**

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

In [ ]:
```
def nonmaximal_suppression(R, grid_size):
    R_prime = np.zeros_like(R)
    height, width = R.shape
    for i in range(grid_size, height - grid_size, grid_size):
        for j in range(grid_size, width - grid_size, grid_size):
            cell = R[i : i + grid_size, j : j + grid_size]

            max_value = np.max(cell)

            max_index = np.unravel_index(np.argmax(cell), cell.shape)

            R_prime[i + max_index[0], j + max_index[1]] = max_value
    return R_prime


def top_n_coordinates(R, n):
    coords = []
    indices = np.argsort(R, axis=None)[::-1]
    for i in range(n):
        coord = np.unravel_index(indices[i], R.shape)
        coords.append(coord[::-1])
    coords = np.array(coords)
    return coords
```

In [ ]:
```
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

    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 = np.zeros((nCorners, 2))
    minor_eig_image = np.zeros_like(image)

    ### YOUR CODE HERE

    #1. Smooth the image
    gaussian_kernel = gaussian2d(windowSize, smoothSTD)
    smooth = convolve(image, gaussian_kernel, mode="same")

    #2. Compute the derivatives
    Ix = cv2.Sobel(smooth, cv2.CV_64F, 1, 0, ksize=3)
    Iy = cv2.Sobel(smooth, cv2.CV_64F, 0, 1, ksize=3)

    #3. Compute the squared derivatives + gaussian smoothing
    Ixx = convolve(Ix**2, gaussian_kernel, mode="same")
    Iyy = convolve(Iy**2, gaussian_kernel, mode="same")
    Ixy = convolve(Ix*Iy , gaussian_kernel, mode="same")

    offset = windowSize // 2
    # Itérer sur l'image pour détecter les coins
    for y in range(windowSize, smooth.shape[0] - windowSize):
        for x in range(windowSize, smooth.shape[1] - windowSize):
            Sx = np.sum(
                Ixx[y - offset : y + 1 + offset, x - offset : x + 1 + offset]
            )
            Sy = np.sum(
                Iyy[y - offset : y + 1 + offset, x - offset : x + 1 + offset]
            )
            Sxy = np.sum(
                Ixy[y - offset : y + 1 + offset, x - offset : x + 1 + offset]
            )

            C = np.array([[Sx, Sxy], [Sxy, Sy]])
            eigenvalues = np.linalg.eigvals(C)
            min_eig = np.min(eigenvalues)
            if min_eig > 0.85 :
                minor_eig_image[y, x] = min(eigenvalues)

    corners = nonmaximal_suppression(minor_eig_image, 30)
    corners = top_n_coordinates(corners, nCorners)

    ### END YOUR CODE

    return corners, minor_eig_image
```

In [ ]:

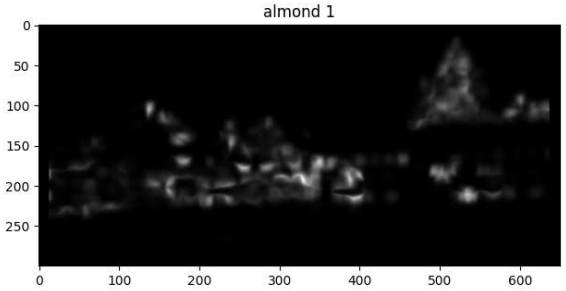
```
def show_eigen_images(imgs):
    print("Minor Eigen value images")
    fig = plt.figure(figsize=(16, 16))
    # Plot image 1
    plt.subplot(1, 2, 1)
    plt.imshow(imgs[0], cmap="gray")
    plt.title("almond 1")

    # Plot image 2
```

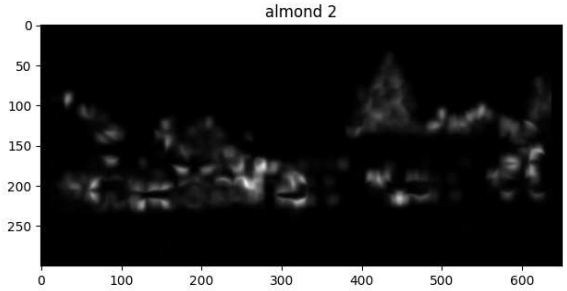
```
    plt.subplot(1, 2, 2)
    plt.imshow(imgs[1], cmap="gray")
    plt.title("almond 2")

    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()
```

In [ ]:
```
# 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)
```

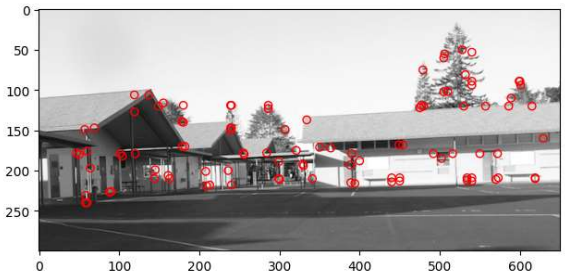
In [ ]:
```
show_eigen_images(eig_imgs)
show_corners_result(imgs, corners)
```

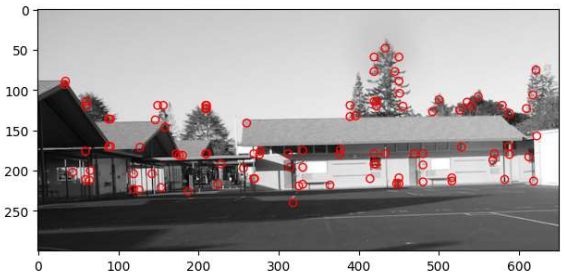
```
Minor Eigen value images
```






```
Detected Corners
```





Problem 2: Theory [20 points]

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).

fig 1.1

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

!! Hint !!

Think about how to convert the points (x, y) and (u, v) in the world coordinates. How would you use them and the camera centers in the world coordinates to get the 3D location of the point?

We know that the 3D location of this point is defined by (X_0, Y_0, Z_0) with :

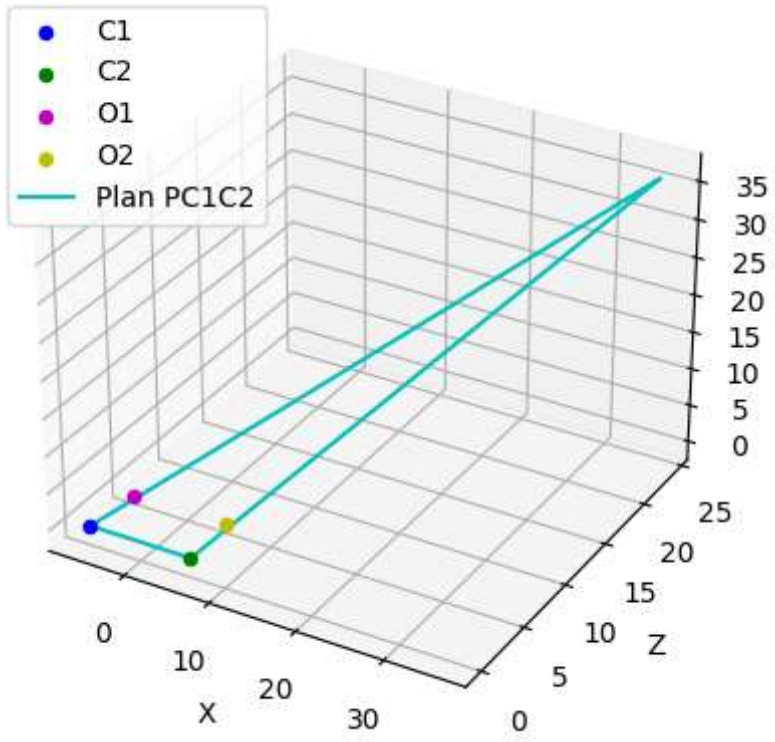
- $X_0 = \frac{d \cdot X_l}{X_l - X_r}$
- $Y_0 = \frac{d \cdot Y_l}{X_l - X_r}$
- $Z_0 = \frac{d \cdot f}{X_l - X_r}$

Here, $X_l = 3, Y_l = 3, f = 2, X_r = 2, Y_r = 3, d = 12$

That is why the 3D location of the point is $(36, 36, 24)$, you can find a 3D representation below :

In [ ]:
```
P = np.array([36, 36, 24])
C1 = np.array([-6, 0, 0])
C2 = np.array([6, 0, 0])
O1 = np.array([-3, 2, 3])
O2 = np.array([8, 2, 3])

fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

ax.scatter(*C1, color="b", label="C1")
```

```
ax.scatter(*C2, color="g", label="C2")
ax.scatter(*O1, color="m", label="O1")
ax.scatter(*O2, color="y", label="O2")

x_vals = np.array([P[0], C1[0], C2[0], P[0]])
y_vals = np.array([P[1], C1[1], C2[1], P[1]])
z_vals = np.array([P[2], C1[2], C2[2], P[2]])

ax.plot(x_vals, z_vals, y_vals, color="c", label="Plan PC1C2")
ax.legend()
ax.set_xlabel("X")
ax.set_ylabel("Z")
ax.set_zlabel("Y")

plt.show()
```



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.

fig 1.2

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

!! Hint !!

You will be required to convert the points into homogeneous coordinates.

- The fundamental matrix F is defined by

$$\mathbf{x}'^{\mathsf{T}}\mathbf{F}\mathbf{x} = 0$$

for any pair of matches x and x' in two images.

- Let $\mathbf{x} = (u,v,1)^{\mathsf{T}}$ and $\mathbf{x}' = (u',v',1)^{\mathsf{T}}$, $\mathbf{F} = \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix}$
- Each match gives a linear equation:

$$uu'f_{11} + vu'f_{12} + u'f_{13} + uv'f_{21} + vv'f_{22} + v'f_{23} + uf_{31} + vf_{32} + f_{33} = 0$$

Here $x = (0, 0, 1)$ and $x' = (0, 0, 1)$, according to the linear equation above : F_{33} = 0.

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).

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

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

!! Hint !!

You may find the following implementation of SVD useful:
https://numpy.org/doc/stable/reference/generated/numpy.linalg.svd.html

$E = [t]_X \cdot R$

with $[t]_X = \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix}$

In [ ]:
```
t = [2, 5, 1]
R = np.array(
    [
        [np.sqrt(2) / 2, -np.sqrt(2) / 2, 0],
```

```
        [np.sqrt(2) / 2, np.sqrt(2) / 2, 0],
        [0, 0, 1],
    ]
)
tx = np.array([[0, -t[2], t[1]], [t[2], 0, -t[0]], [-t[1], t[0], 0]])
E = tx @ R
print("Essential Matrix : \n", E)
```

```
Essential Matrix :
 [[-0.70710678 -0.70710678  5.        ]
 [ 0.70710678 -0.70710678 -2.        ]
 [-2.12132034  4.94974747  0.        ]]
```

Compute the rank (sum of nonzero singular values S)

In [ ]:
```
_, S, _ = np.linalg.svd(E)
rank = np.sum(S > 1e-15)
print(f"Rank : {rank}")
```

```
Rank : 2
```

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.

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

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

In [ ]:
```
import numpy as np


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

    Args:
        img1: Image 1.
        img2: Image 2.
        c1: Center (in image coordinates) of the window in image 1.
        c2: Center (in image coordinates) of the window in image 2.
        R: R is half the side length of the square window.

    Returns:
        SSD matching score for two input windows (a scalar value).

    """
    ### YOUR CODE HERE
    w1 = img1[c1[1] - R : c1[1] + R + 1, c1[0] - R : c1[0] + R + 1]
    w2 = img2[c2[1] - R : c2[1] + R + 1, c2[0] - R : c2[0] + R + 1]

    matching_score = np.sum((w1 - w2) ** 2)
    ### END YOUR CODE
```

```
    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
```

```
20
30
46
```

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.

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 (a scalar value).

    """

    ### YOUR CODE HERE
    w1 = img1[c1[1] - R : c1[1] + R + 1, c1[0] - R : c1[0] + R + 1]
    w2 = img2[c2[1] - R : c2[1] + R + 1, c2[0] - R : c2[0] + R + 1]

    mean_w1 = np.mean(w1)
    mean_w2 = np.mean(w2)

    tilde_w1 = (w1 - mean_w1) / np.sqrt(np.sum((w1 - mean_w1) ** 2))
    tilde_w2 = (w2 - mean_w2) / np.sqrt(np.sum((w2 - mean_w2) ** 2))

    NCC = np.sum(tilde_w1 * tilde_w2)
    matching_score = np.trunc(NCC * 10000) / 10000
    ### END YOUR CODE

    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
```

```
0.8546
0.8457
0.6258
```

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.

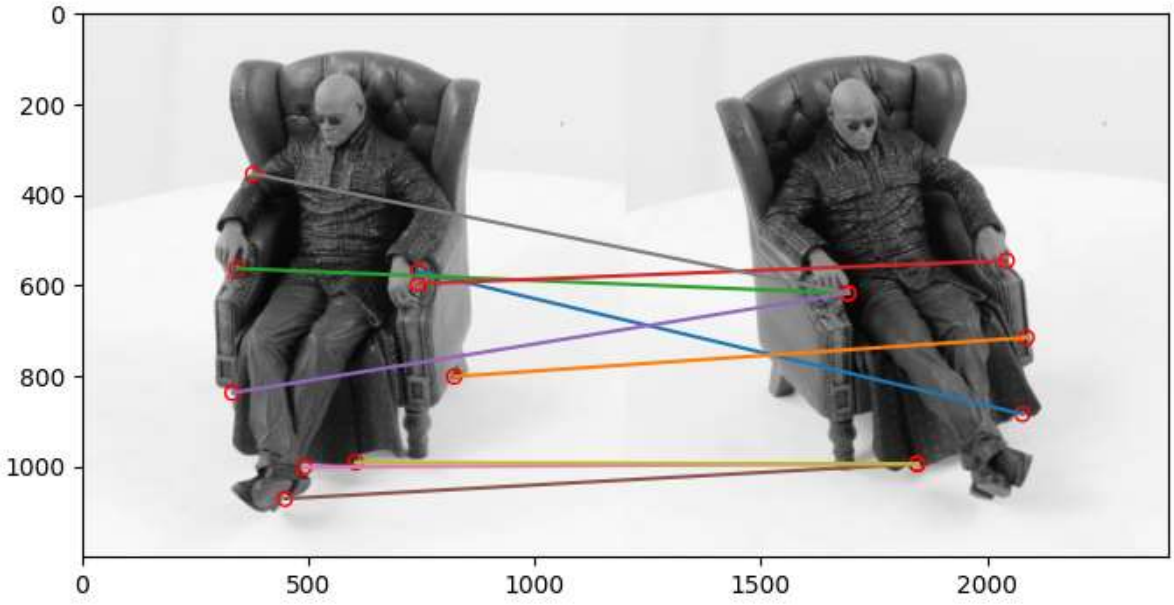
In [ ]:
```
def naive_matching(image1, image2, image1_corners, image2_corners, R, NCCth):
    """Compute NCC given two windows.

    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 returns a list of tuple (c1, c2),
        c1 is the 1x2 corner location in image 1,
        c2 is the 1x2 corner location in image 2.

    """
    matches = []

    for corner1 in image1_corners:
        best_match = None
        best_score = -1

        for corner2 in image2_corners:
            score = nccMatch(image1, image2, corner1, corner2, R)

            if score > best_score:
                best_score = score
                best_match = corner2

        if best_score >= NCCth:
            matches.append((corner1, best_match))

    return matches
```

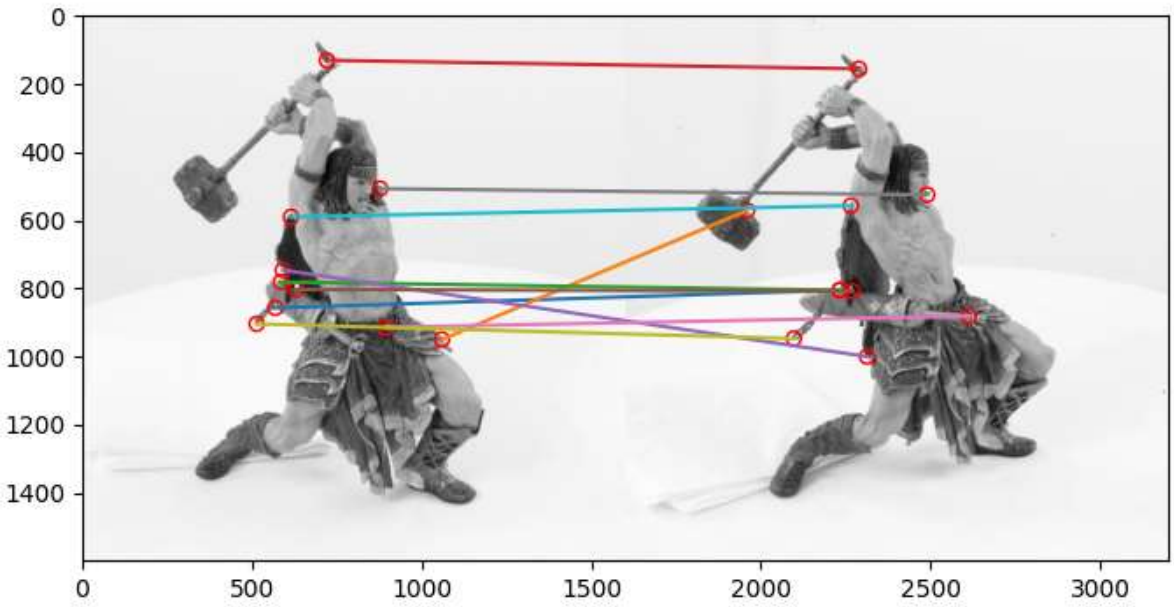
In [ ]:
```
def rgb2gray(rgb):
    """Convert rgb image to grayscale."""
    return np.dot(rgb[..., :3], [0.299, 0.587, 0.114])


# 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.v2.imread("p4/matrix/matrix" + str(i) + ".png")
    imgs_mat.append(rgb2gray(img_mat))
    img_war = imageio.v2.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_war.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
)
```

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 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
```





Problem 4: Epipolar Geometry [13 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.

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**)=USV^T. And **f** is given by the singular vector corresponding to the smallest singular value, which is the last column of V.

In [ ]:
```python
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 = 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],
        ]

    _, _, V = np.linalg.svd(A)
    F = V[-1].reshape(3, 3)

    U, S, V = np.linalg.svd(F)
    S[2] = 0
    F = np.dot(U, np.dot(np.diag(S), V))
    ### END YOUR CODE

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

In [ ]:
```python
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]]
```

```
[[ 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]]
```

Problem 4.2: Plot Epipolar Lines [3 points]

You do not need to code anything here. Run the cells below and look at your results

Using this fundamental matrix, we can plot the epipolar lines in both images for each image pair. For this part, you just have to use the function `plot_epipolar_lines` and check for the correctness of `compute_fundamental` that you have already written in Q4.1). Show your result for matrix and warrior as exemplified by the figure below.

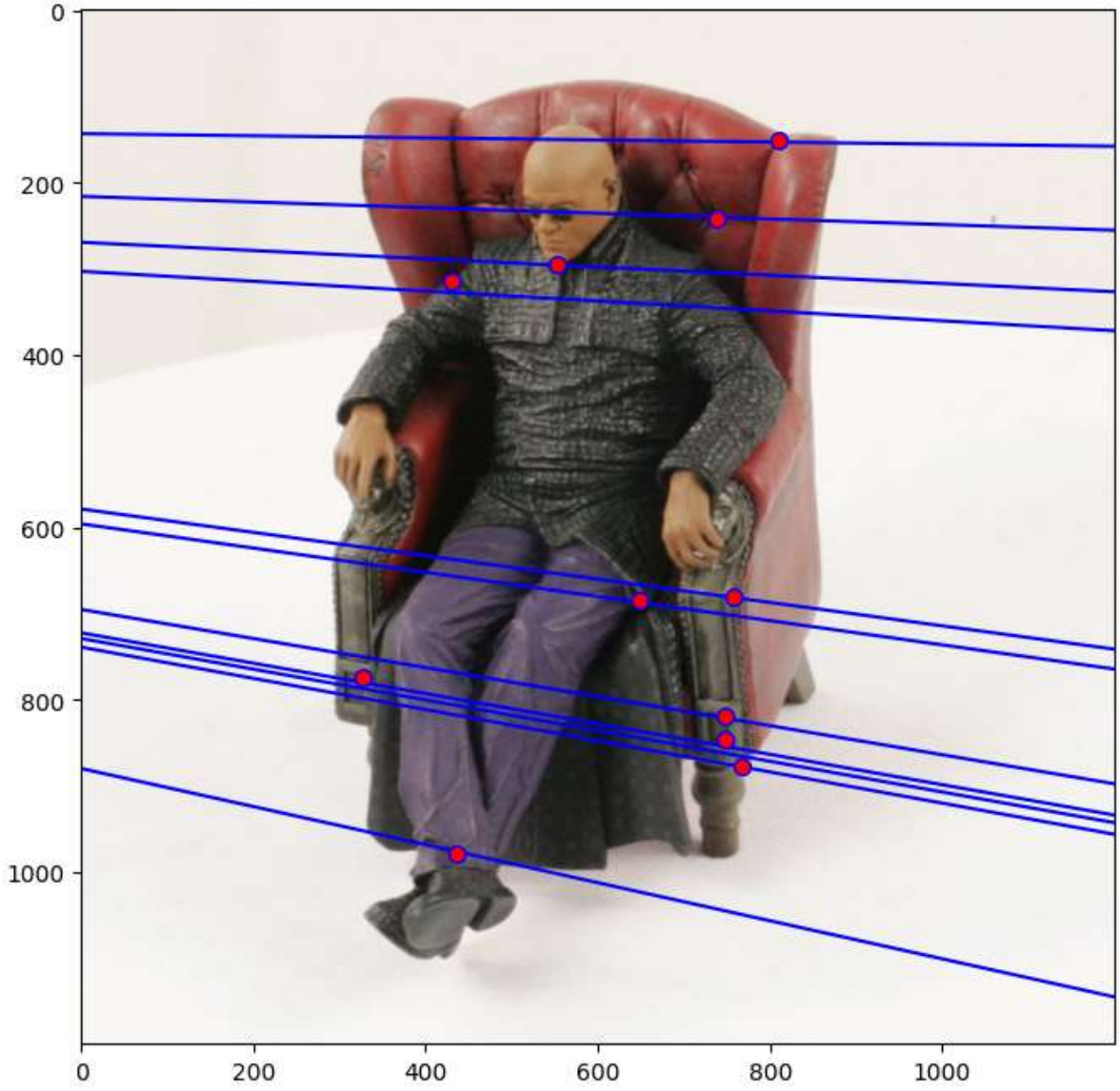
 Dino Epipolar  Dino Epipolar

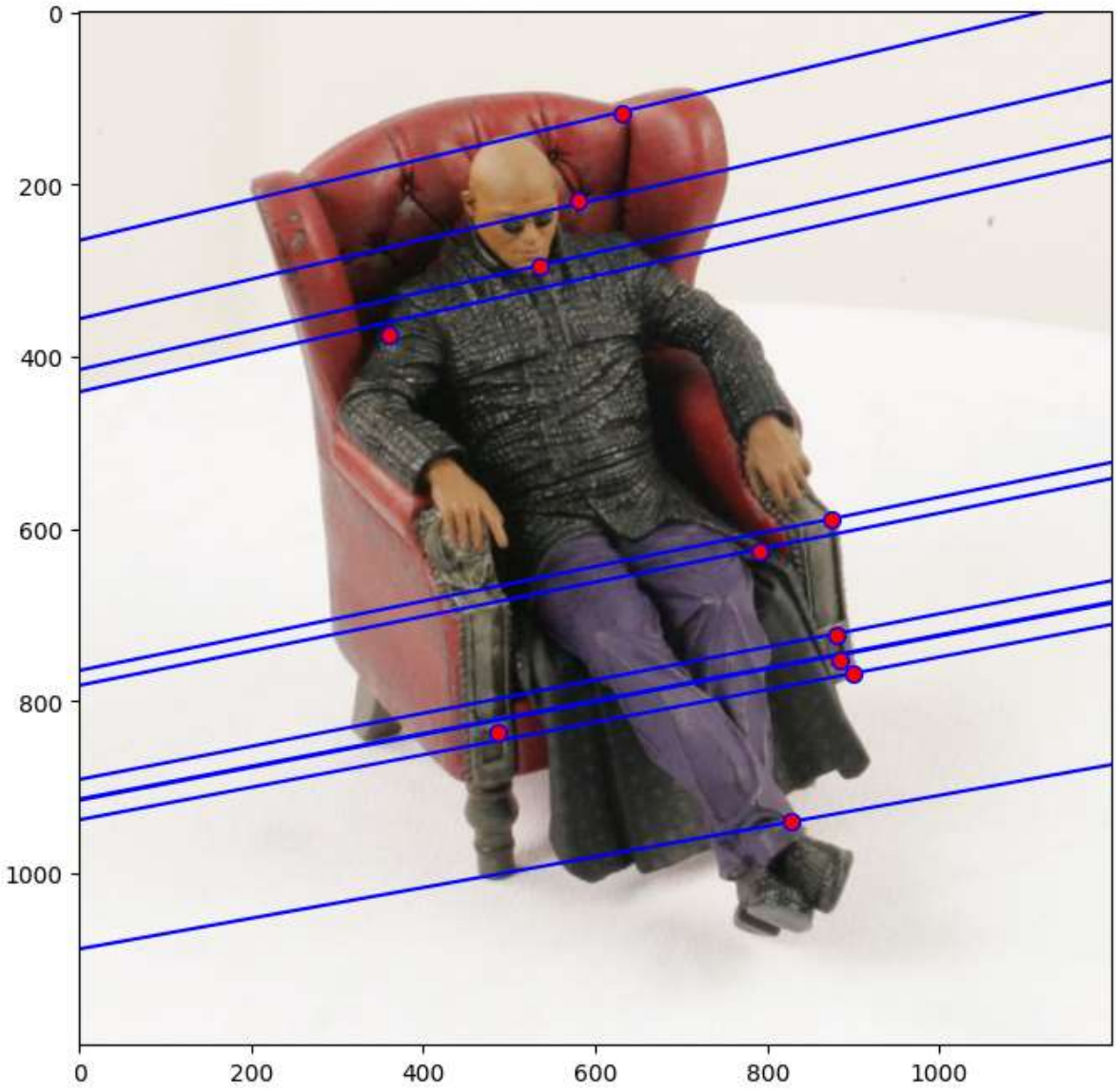
In [ ]:
```
def plot_epipolar_lines(img1, img2, cor1, cor2):
    """Plot epipolar lines on image given image, corners

    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)

    # epipole in image 1 is the solution to F^T e = 0
    U, S, V = np.linalg.svd(F.T)
    e1 = V[-1]
    e1 /= e1[-1]

    # epipole in image 2 is the solution to Fe = 0
    U, S, V = np.linalg.svd(F)
    e2 = V[-1]
    e2 /= e2[-1]

    plot_epipoles = False

    # Plot epipolar lines in the first image
    # There is an epipolar line for each corner
    fig = plt.figure(figsize=(8, 8))
```

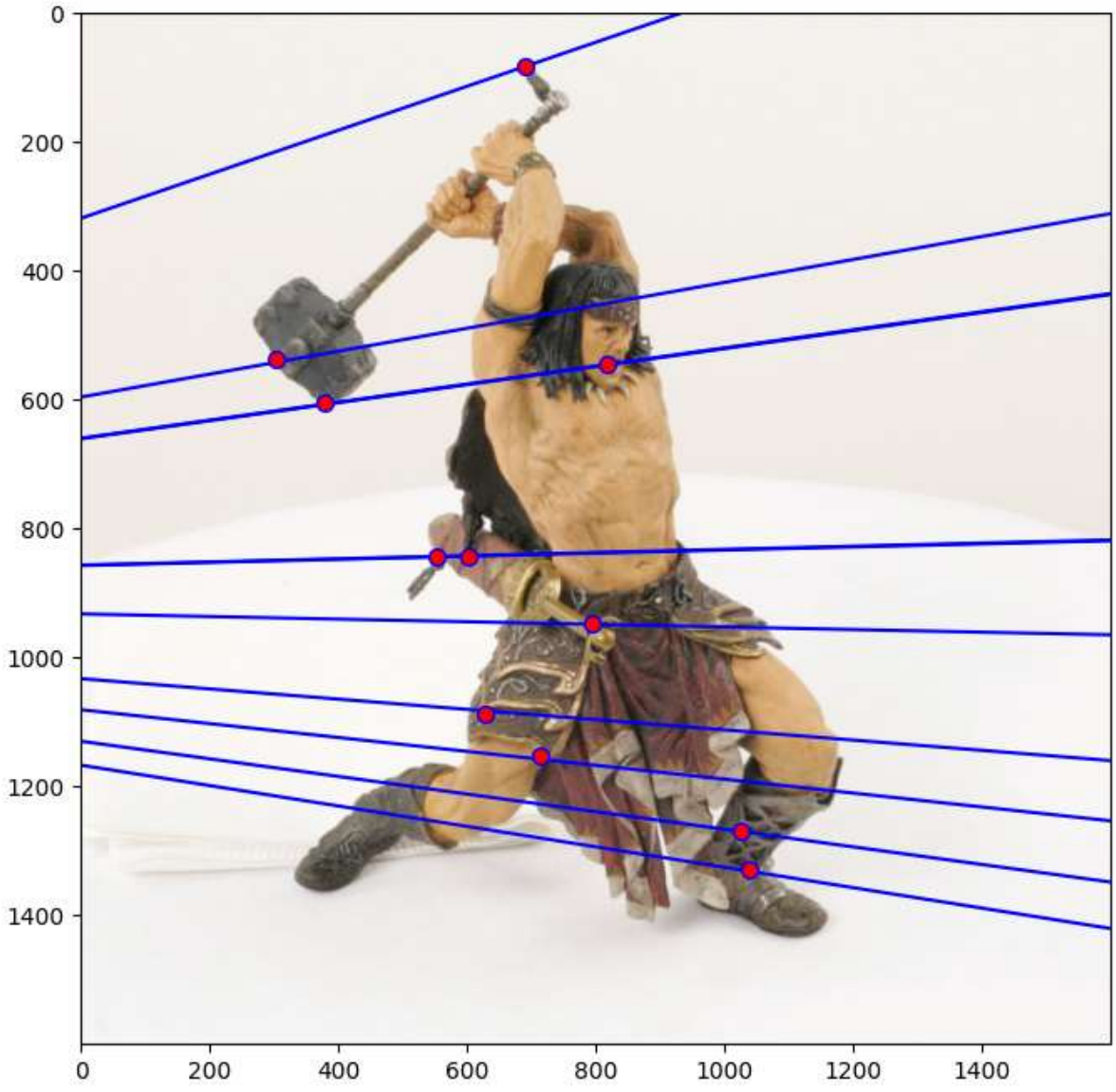
```
    plt.imshow(img1, cmap="gray")
    h, w = img1.shape[:2]
    for c2 in cor2.T:
        # epipolar line is (F * c2) dot (x, y, 1) = 0
        epi_line = np.dot(F, c2)
        a, b, c = epi_line  # ax + by + c = 0, y = -a/b * x - c/b
        x = np.arange(w)
        y = (-a / b) * x - (c / b)
        x = np.array([x[i] for i in range(x.size) if y[i] >= 0 and y[i] < h - 1])
        y = np.array([y[i] for i in range(y.size) if y[i] >= 0 and y[i] < h - 1])
        plt.plot(x, y, "b", zorder=1)

    plt.scatter(cor1[0], cor1[1], s=50, edgecolors="b", facecolors="r", zorder=2)

    if plot_epipoles:
        plt.scatter([e1[0]], [e1[1]], s=75, edgecolors="g", facecolors="y", zorder=
    plt.show()

    # Plot epipolar lines in the second image
    fig = plt.figure(figsize=(8, 8))
    plt.imshow(img2, cmap="gray")
    h, w = img2.shape[:2]

    for c1 in cor1.T:
        # epipolar line is (F^T * c1) dot (x, y, 1) = 0
        epi_line = np.dot(F.T, c1)
        a, b, c = epi_line
        x = np.arange(w)
        y = (-a / b) * x - (c / b)
        x = np.array([x[i] for i in range(x.size) if y[i] >= 0 and y[i] < h - 1])
        y = np.array([y[i] for i in range(y.size) if y[i] >= 0 and y[i] < h - 1])
        plt.plot(x, y, "b", zorder=1)

    plt.scatter(cor2[0], cor2[1], s=50, edgecolors="b", facecolors="r", zorder=2)

    if plot_epipoles:
        plt.scatter([e2[0]], [e2[1]], s=75, edgecolors="g", facecolors="y", zorder=
    plt.show()
```

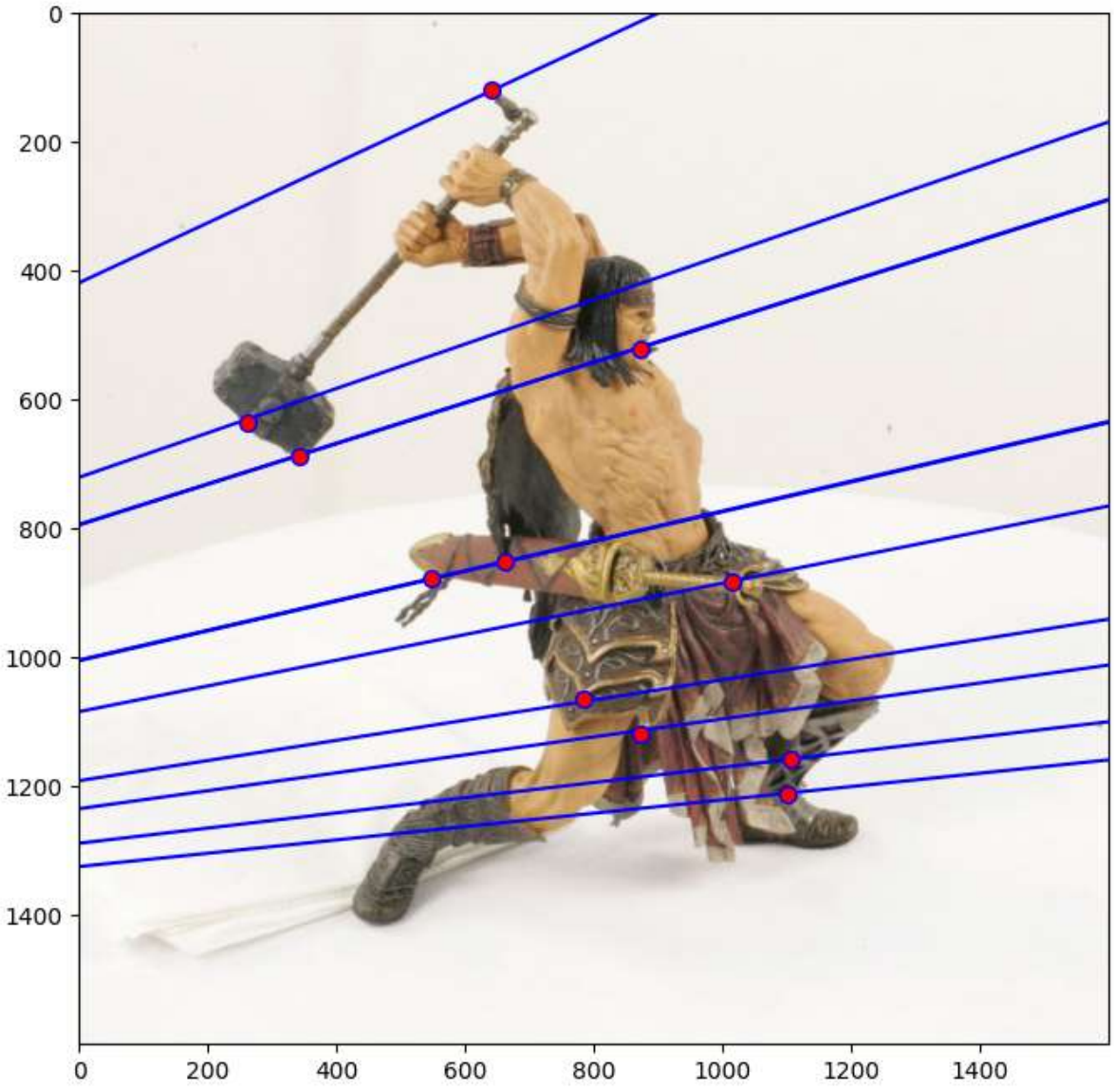
In [ ]:

```
# replace images and corners with those of matrix and warrior
imgids = ["matrix", "warrior"]
for imgid in imgids:
    I1 = imageio.v2.imread("./p4/" + imgid + "/" + imgid + "0.png")
    I2 = imageio.v2.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
0
200
400
600
800
1000


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


0
200
400
600
800
1000
1200
1400
0
200
400
600
800
1000
1200
1400


0
200
400
600
800
1000
1200
1400
0
200
400
600
800
1000
1200
1400