

# HW4

November 19, 2019

## 1 CSE 252A Computer Vision I Fall 2019 - Homework 4

### 1.1 Instructor: Ben Ochoa

1.1.1 Assignment published on: Tuesday, November 05, 2019

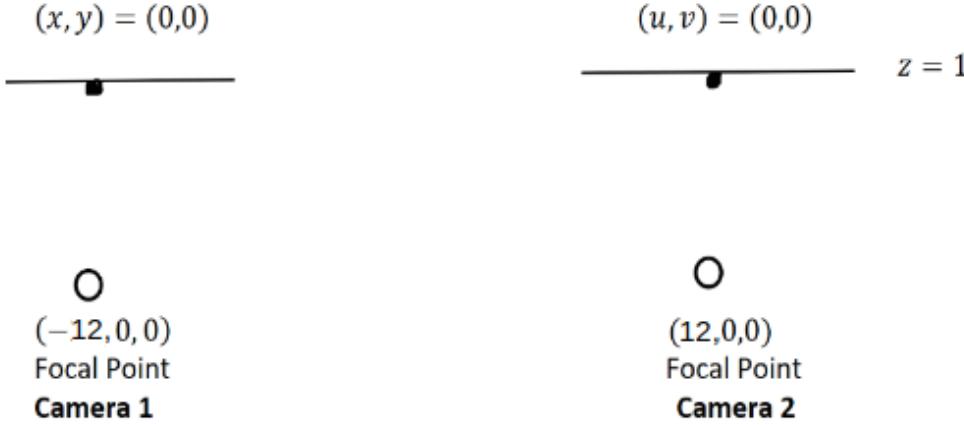
1.1.2 Due on: Tuesday, November 19, 2019 11:59 pm

### 1.2 Instructions

- Review the academic integrity and collaboration policies on the course website.
  - This assignment must be completed individually.
- All solutions must be written in this notebook.
  - This includes the theoretical problems, for which you **must** write your answers in Markdown cells (using LaTeX when appropriate).
  - Programming aspects of the assignment must be completed using Python in this notebook.
- If you want to modify the skeleton code, you may do so. It has only been provided as a framework for your solution.
- You may use Python packages (such as NumPy and SciPy) for basic linear algebra, but you may not use packages that directly solve the problem.
  - If you are unsure about using a specific package or function, then ask the instructor and/or teaching assistants for clarification.
- You must submit this notebook exported as a PDF. You must also submit this notebook as .ipynb file.
  - Submit both files (.pdf and .ipynb) on Gradescope.
  - **You must mark the PDF pages associated with each question in Gradescope. If you fail to do so, we may dock points.**
- 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: Epipolar Geometry [4 pts]

Consider two cameras whose image planes are the  $z=1$  plane, and whose focal points are at  $(-12, 0, 0)$  and  $(12, 0, 0)$ . 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  $(-12, 0, 1)$  in world coordinates, while if  $(u, v) = (0, 0)$  this is the point  $(12, 0, 1)$ .



- a) Suppose the points  $(x, y) = (8, 7)$  is matched to the point  $(u, v) = (2, 7)$ . What is the 3D location of this point?
- b) Compute the Essential Matrix.
- c) Consider points that lie on the line  $x + z = 2, y = 0$ . Use the same stereo set up as before. Write an analytic expression giving the disparity of a point on this line after it projects onto the two images, as a function of its position in the right image. So your expression should only involve the variables  $u$  and  $d$  (for disparity). Your expression only needs to be valid for points on the line that are in front of the cameras, i.e. with  $z > 1$ .

### 1.4 Problem 2: Epipolar Rectification [4 pts]

In stereo vision, image rectification is a common preprocessing step to simplify the problem of finding matching points between images. The goal is to warp image views such that the epipolar lines are horizontal scan lines of the input images. Suppose that we have captured two images  $I_A$  and  $I_B$  from identical calibrated cameras separated by a rigid transformation

$${}^B_A T = \begin{bmatrix} R & t \\ 0^T & 1 \end{bmatrix} \text{ and } t = [t_x, t_y, t_z]$$

Without loss of generality assume that camera A's optical center is positioned at the origin and that its optical axis is in the direction of the  $z$ -axis.

From the lecture, a rectifying transform for each image should map the epipole to a point infinitely far away in the horizontal direction  $H_A e_A = H_B e_B = [1, 0, 0]^T$ . Consider the following special cases:

- a) Pure horizontal translation  $t = [t_x, 0, 0]^T, R = I$
- b) Pure translation orthogonal to the optical axis  $t = [t_x, t_y, 0]^T, R = I$

### Problem 1

a)

$$(x,y) = (8,7) \text{ is matched to the point } \begin{bmatrix} 1 & 0 & 0 & -12 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 8 \\ 7 \\ 1 \end{bmatrix} = \begin{bmatrix} -4 \\ 7 \\ 1 \end{bmatrix} \text{ in world}$$

coordinate, and the same applies to  $(u,v) = (2,7)$  which is  $\begin{pmatrix} 14 & 7 & 1 \end{pmatrix}$  in world coordinate.

Therefore we could have line

$$OP_1 : \frac{x+12}{8} = \frac{y}{7} = \frac{z}{1}$$

and

$$OP_2 : \frac{x-12}{2} = \frac{y}{7} = \frac{z}{1}$$

$$\begin{pmatrix} 20 & 28 & 4 \end{pmatrix}$$

and the intersection of the two line is the 3D location of that point, which is

b)

$$\text{We have } t = \begin{pmatrix} -24 & 0 & 0 \end{pmatrix}, R = I,$$

$$E = [t]R$$

Therefore the essential matrix is

$$E = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -24 \\ 0 & 24 & 0 \end{bmatrix} R = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -24 \\ 0 & 24 & 0 \end{bmatrix}$$

c)

Since the focal point is at  $y=0$ , and all the points on the line are at  $y=0$ , we can consider only the  $xz$ -plane.

For the porjection of point  $x$  in camera1 we have

$$X_L = \frac{f(x+12)}{z}$$

and the projection point  $x$  in camera2 which is also  $u$  is

$$X_R = \frac{f(x-12)}{z} = u = \frac{x-12}{2-x}$$

$$x = \frac{12+2u}{1+u}$$

Therefore the disparity  $d$  is

$$d = \frac{f(x+12)}{z} - \frac{f(x-12)}{z} = \frac{24}{z} = \frac{24}{2-x} = \frac{24}{2 - \frac{12+2u}{1+u}} = -\frac{12}{5}(1+u)$$

## Problem 2

a) Pure Horizontal Translation:  $t = [t_x, 0, 0]^T$ ,  $R = I$

i. For essential matrix E we have

$$E = [t]R = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix}$$

and

$$E^T = [t]R = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & t_x \\ 0 & -t_x & 0 \end{bmatrix}$$

Since  $E^T e_A = 0$  and  $E e_B = 0$ , we have the epipoles  $e_A = [1, 0, 0]^T$ ,  $e_B = [1, 0, 0]^T$

ii. Set  $P = [x_A, y_A, 1]^T$  and  $P' = [x_B, y_B, 1]^T$ , we have

$$P^T EP' = [x_A, y_A, 1] \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix} \begin{bmatrix} x_B \\ y_B \\ 1 \end{bmatrix} = t_x(y_B - y_A) = 0$$

Therefore the epipolar line  $l_B : y_B = y_A$

iii. First construct 3 mutually orthogonal unit vectors  $e_1, e_2, e_3$ , set the first vector the direction

of translation  $e_1 = \frac{t}{|t|} = [1, 0, 0]^T$ , and set the second vector the normalized cross product of

$e_1$  with the direction vector of the optical axis  $e_2 = \frac{1}{\sqrt{t_x^2 + t_y^2}} [-t_y, t_x, 0]^T = [0, 1, 0]^T$ , then set

the third vector the normalized cross product of the first two vector  $e_3 = e_1 \times e_2 = [0, 0, 1]$ . Thus

$$R_{rect} = \begin{bmatrix} e_1^T \\ e_2^T \\ e_3^T \end{bmatrix} = I$$

we have the rectification matrix  $H_A = R_{rect} = I$  and  $H_B = R \cdot R_{rect} = I$ .

b) Pure Translation Orthogonal to the Optical Axis:  $t = [t_x, t_y, 0]^T$ ,  $R = I$

i. For essential matrix we have

$$E = [t]R = \begin{bmatrix} 0 & 0 & t_y \\ 0 & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix}$$

Since  $E^T e_A = 0$  and  $Ee_B = 0$ , we have the epipoles  $e_A = [t_x, t_y, 0]^T$  and  $e_B = [t_x, t_y, 0]^T$

ii. Set  $P = [x_A, y_A, 1]^T$  and  $P' = [x_B, y_B, 1]^T$ , we have

$$P^T EP' = [x_A, y_A, 1] \begin{bmatrix} 0 & 0 & t_y \\ 0 & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix} \begin{bmatrix} x_B \\ y_B \\ 1 \end{bmatrix} = t_x(y_B - y_A) + t_y(x_A - x_B) = 0$$

Therefore the epipolar line  $l_B : t_x(y_B - y_A) + t_y(x_A - x_B) = 0$

$$e_1 = \frac{t}{|t|} = \left[ -\frac{t_x^2}{\sqrt{t_x^2 + t_y^2}}, \frac{t_y^2}{\sqrt{t_x^2 + t_y^2}}, 0 \right]^T$$

iii. Set the first vector

$$e_2 = \frac{1}{\sqrt{t_x^2 + t_y^2}} [-t_y, t_x, 0]^T = \left[ \frac{-t_y^2}{\sqrt{t_x^2 + t_y^2}}, \frac{t_x^2}{\sqrt{t_x^2 + t_y^2}}, 0 \right]^T$$

the second vector

and the third vector  $e_3 = e_1 \times e_2 = [0, 0, 1]$

$$R_{rect} = \begin{bmatrix} e_1^T \\ e_2^T \\ e_3^T \end{bmatrix} = \begin{bmatrix} \frac{t_x^2}{\sqrt{t_x^2 + t_y^2}} & \frac{t_y^2}{\sqrt{t_x^2 + t_y^2}} & 0 \\ \frac{-t_y^2}{\sqrt{t_x^2 + t_y^2}} & \frac{t_x^2}{\sqrt{t_x^2 + t_y^2}} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Thus we have the rectification matrix

therefor the solution to rectifying transforms is

$$H_A = H_B = R_{rect} = \begin{bmatrix} \frac{t_x^2}{\sqrt{t_x^2 + t_y^2}} & \frac{t_y^2}{\sqrt{t_x^2 + t_y^2}} & 0 \\ \frac{-t_y^2}{\sqrt{t_x^2 + t_y^2}} & \frac{t_x^2}{\sqrt{t_x^2 + t_y^2}} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

c) Pure Translation Along the Optical Axis  $t = [0, 0, t_z]^T$ ,  $R = I$

i. For essential matrix we have

$$E = [t]R = \begin{bmatrix} 0 & -t_z & 0 \\ t_z & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Since  $E^T e_A = 0$  and  $E e_B = 0$ , we have the epipoles  $e_A = [0, 0, 1]^T$  and  $e_B = [0, 0, 1]^T$

ii. Set  $P = [x_A, y_A, 1]^T$  and  $P' = [x_B, y_B, 1]^T$ , we have

$$P^T EP' = [x_A, y_A, 1] \begin{bmatrix} 0 & -t_z & 0 \\ -t_z & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_B \\ y_B \\ 1 \end{bmatrix} = t_z(y_A x_B - y_B x_A) = 0$$

Therefore the epipolar line  $l_B : (y_A x_B - y_B x_A) = 0$

$$\text{iii. Set the first vector } e_1 = \frac{t}{|t|} = [0, 0, 1]^T,$$

$$\text{the second vector } e_2 = \frac{1}{\sqrt{0^2 + 0^2}} [-t_y, t_x, 0]^T,$$

We found that  $e_2$  doesn't exist, therefore we are not able to find its rectifying transform matrices, the epipolar rectification is impossible.

d) Pure Rotation  $t = [0, 0, 0]^T$ , R is an arbitrary rotation matrix

i. For essential matrix we have

$$E = [t]R = 0$$

Since  $E^T e_A = 0$  and  $E e_B = 0$ , the epipoles  $e_A$  and  $e_B$  could be any existing matrix.

ii. Set  $P = [x_A, y_A, 1]^T$  and  $P' = [x_B, y_B, 1]^T$ , we have

$$P^T 0 P' = 0$$

Therefore  $l_B$  does not exist.

iii. For  $e_1 = \frac{t}{|t|} = \frac{0}{|0|}$ ,  $e_1$  doesn't exist, therefore we are not able to find its rectifying transform matrices, the epipolar rectification is impossible.

- c) Pure translation along the optical axis  $t = [0, 0, t_z]^T$ ,  $R = I$
- d) Pure rotation  $t = [0, 0, 0]^T$ ,  $R$  is an arbitrary rotation matrix

For each of these cases, determine whether or not epipolar rectification is possible. Include the following information for each case:

- (i) The epipoles  $e_A$  and  $e_B$
- (ii) The equation of the epipolar line  $l_B$  in  $I_B$  corresponding to the point  $[x_A, y_A, 1]^T$  in  $I_A$  (if one exists)
- (iii) A plausible solution to the rectifying transforms  $H_A$  and  $H_B$  (if one exists) that attempts to minimize distortion (is as close as possible to a 2D rigid transformation). Note that the above 4 cases are special cases; a simple solution should become apparent by looking at the epipolar lines.

One or more of the above rigid transformations may be a degenerate case where rectification is not possible or epipolar geometry does not apply. If so, explain why.

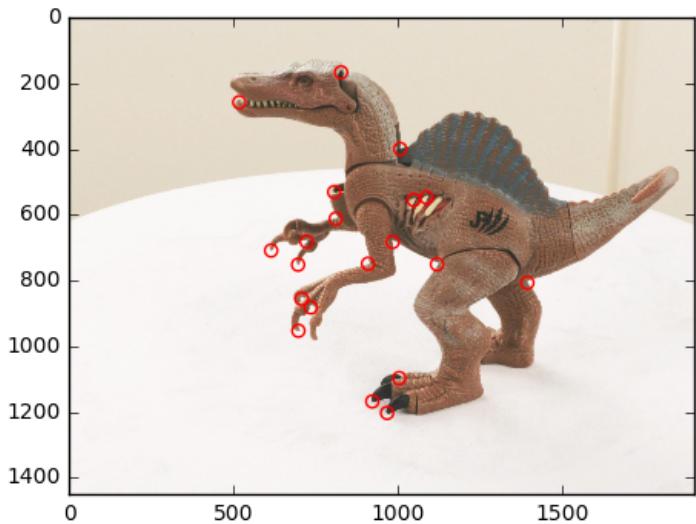
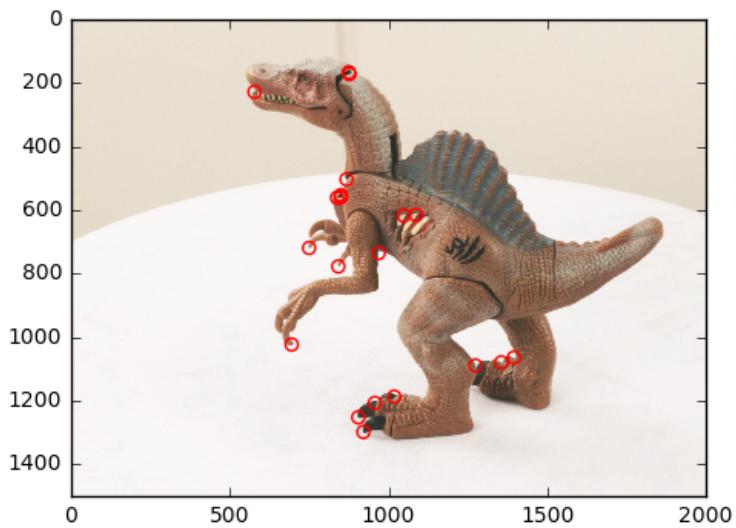
## 1.5 Problem 3: Sparse Stereo Matching [32 pts]

In this problem we will play around with sparse stereo matching methods. You will work on two image pairs, a warrior figure and a figure from the Matrix movies. These files both contain two images, two camera matrices, and two sets of corresponding points (extracted by manually clicking the images). For illustration, I have run my code on a third image pair (dino1.png, dino2.png). This data is also provided for you to debug your code, but **you should only report results on warrior and matrix**. In other words, where I include one (or a pair) of images in the assignment below, you will provide the same thing but for BOTH matrix and warrior. Note that the matrix image pair is harder, in the sense that the matching algorithms we are implementing will not work quite as well. You should expect good results, however, on warrior.

### 1.5.1 Corner Detection [8 pts]

The first thing we need to do is to build a corner detector. This should be done according to <http://cseweb.ucsd.edu/classes/fa19/cse252A-a/lec7.pdf>. You should fill in the function `corner_detect` below, and take as input `corner_detect(image, nCorners, smoothSTD, windowSize)` where `smoothSTD` is the standard deviation of the smoothing kernel and `windowSize` is the window size for corner detector and non maximum suppression. In the lecture the corner detector was implemented using a hard threshold. Do not do that but instead return the `nCorners` strongest corners after non-maximum suppression. This way you can control exactly how many corners are returned. Run your code on all four images (with `nCorners = 20`) and show outputs as shown below. You may find `scipy.ndimage.filters.gaussian_filter` easy to use for smoothing. In this problem, try the following different standard deviation ( $\sigma$ ) parameters for the Gaussian smoothing kernel: 0.5, 1, 2 and 4. For a particular  $\sigma$ , you should take the kernel size to be  $6 \times \sigma$  (add 1 if the kernel size is even). So for example if  $\sigma = 2$ , corner detection kernel size should be 13. This should be followed throughout all experiments in this assignment.

There will be a total of 16 images as outputs : (4 choices of `smoothSTD` x 2 matrix imgs x 2 warrior imgs).



Comment on your results and observations (3/8 points). You don't need to comment per output, just **discuss** any trends you see for the detected corners as you change the windowSize and increase the smoothing w.r.t the two pair of images warrior and matrix. Also discuss if you are able to find corresponding corners for same pair of images.

## 2 trend

When we increase the window size, the corner points are more scattered and sparse since there's a larger area for NMS. When the standard deviation of the smoothing kernel is small, the noisy pattern in the middle of the image might be considered corners. When we increase the smoothing standard deviation, the image has much less noise, but it also blends the edge and corners, so some of the corners are not detected.

```
[1]: import numpy as np
# from scipy.misc import imread
import matplotlib.pyplot as plt
from scipy.ndimage.filters import gaussian_filter
import imageio
from PIL import Image

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

[3]: import math
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 corner detector and non maximum suppression.

    Returns:
        Detected corners (in image coordinate) in a numpy array (n*2).

    """
    Put your awesome numpy powered code here:
    """
    image = gaussian_filter(image, smoothSTD)
    image_dx, image_dy = np.gradient(image)

    Clamb = np.zeros((image.shape[0], image.shape[1]))
    for i in range(0, image.shape[0] - windowSize):
        for j in range(0, image.shape[1] - windowSize):
            Ix = image_dx[i : i+windowSize + 1, j : j+windowSize + 1]
            Iy = image_dy[i : i+windowSize + 1, j : j+windowSize + 1]
            Ix2 = np.sum(Ix * Ix)
            Iy2 = np.sum(Iy * Iy)
            Ixy = np.sum(Ix * Iy)
            Cxy = np.array([[Ix2, Ixy], [Ixy, Iy2]])
            tr = Ix2 + Iy2
            det = Ix2 * Iy2 - Ixy*Ixy
            root = tr * tr - 4 * det
            if root < 0:
                Clamb[i][j] = tr / 2
            else:
```

```

        Clamb[i][j] = (tr - math.sqrt(root)) /2

Csupress = []
for i in range(0, image.shape[0] - windowSize, windowSize):
    for j in range(0, image.shape[1] - windowSize, windowSize):
        patch = Clamb[i: i+windowSize+1, j : j+windowSize+1]
        max_lamb = max(patch.flatten())
        for k in range(0, windowSize):
            for l in range(0, windowSize):
                if patch[k][l] ==max_lamb:
                    Csupress.append((max_lamb, (j + l, i + k)))
            continue

Csupress.sort()
Csupress.reverse()
corners = np.zeros((nCorners, 2), dtype = 'int32')
for i in range(0, corners.shape[0]):
    corners[i, :] = Csupress[i][1]
return corners

```

[4]:

```

def show_corners_result(imgs, corners):
    fig = plt.figure(figsize=(15, 15))
    ax1 = fig.add_subplot(221)
    ax1.imshow(imgs[0], cmap='gray')
    ax1.scatter(corners[0][:, 0], corners[0][:, 1], s=36, edgecolors='r', facecolors='none')

    ax2 = fig.add_subplot(222)
    ax2.imshow(imgs[1], cmap='gray')
    ax2.scatter(corners[1][:, 0], corners[1][:, 1], s=36, edgecolors='r', facecolors='none')
    plt.show()

```

[5]:

```

# detect corners on warrior and matrix image sets
# adjust your corner detection parameters here
nCorners = 20
smoothSTDs = [0.5, 1, 2, 4]
imgs_mat = []
imgs_war = []
grayimgs_mat = []
grayimgs_war = []
# Read the two images and convert it to Greyscale
for i in range(2):
    img_mat = imageio.imread('p4/matrix/matrix' + str(i) + '.png')
    imgs_mat.append(img_mat)
    grayimgs_mat.append(rgb2gray(img_mat))

    img_war = imageio.imread('p4/warrior/warrior' + str(i) + '.png')

```

```









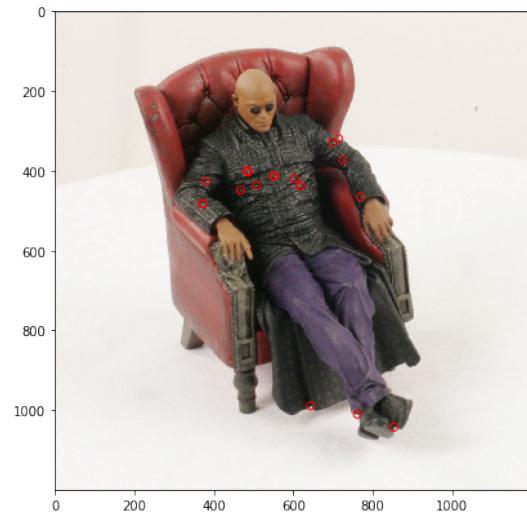
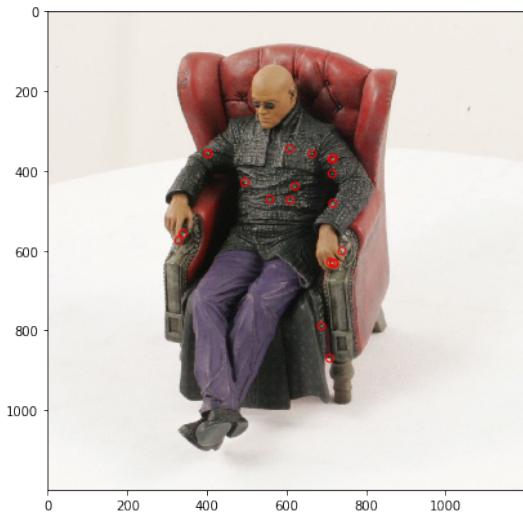


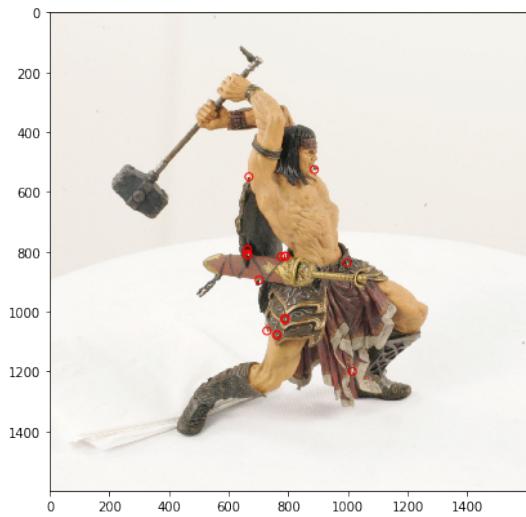
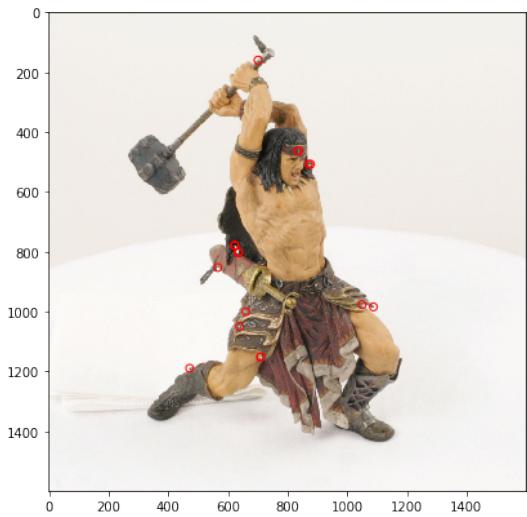




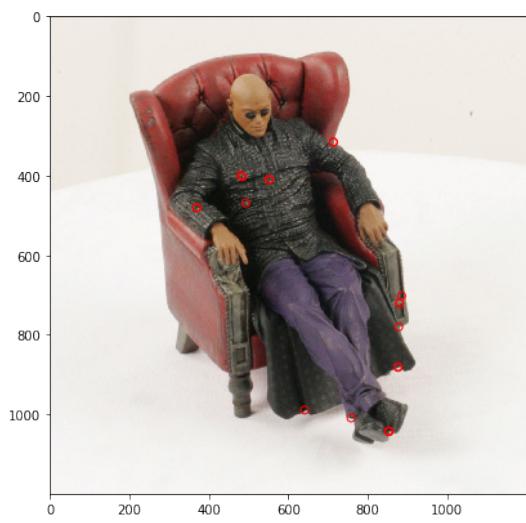
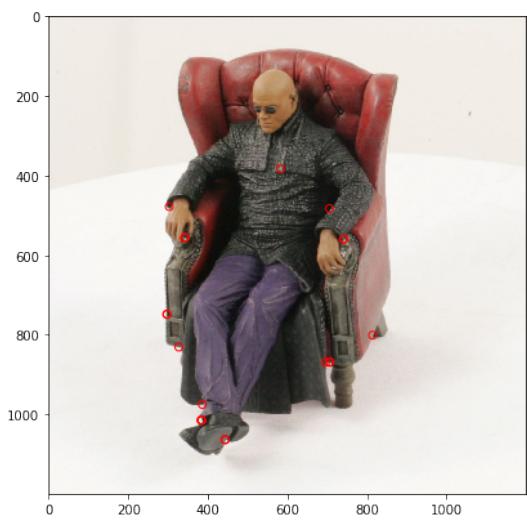
```

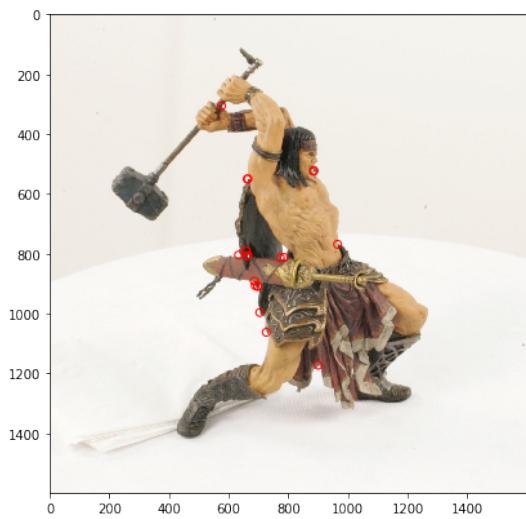
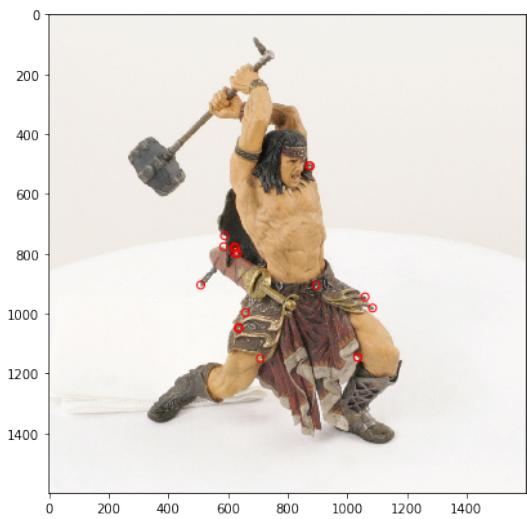
SmoothSTD: 0.5 WindowSize: 3



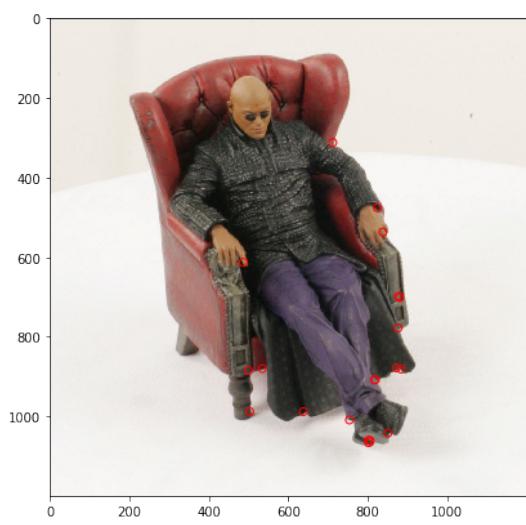
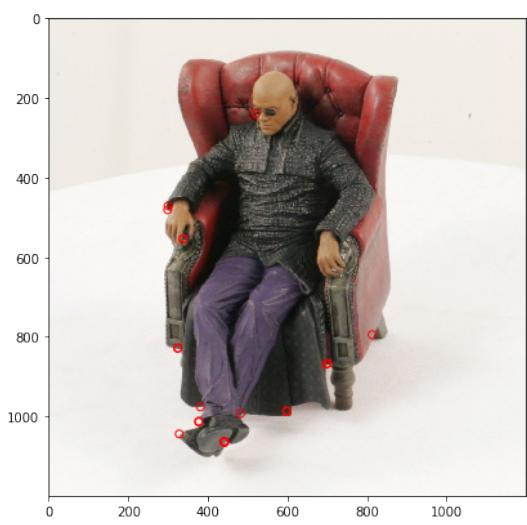


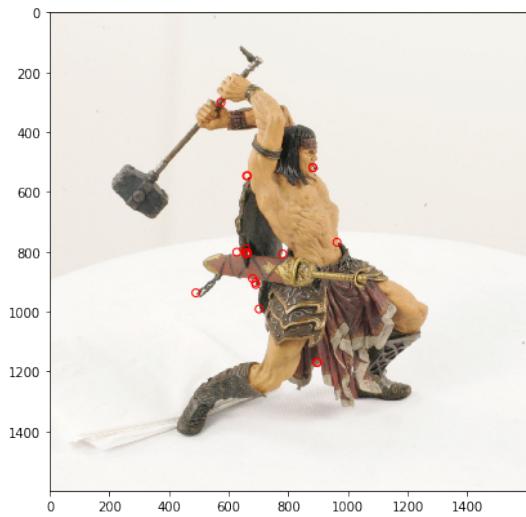
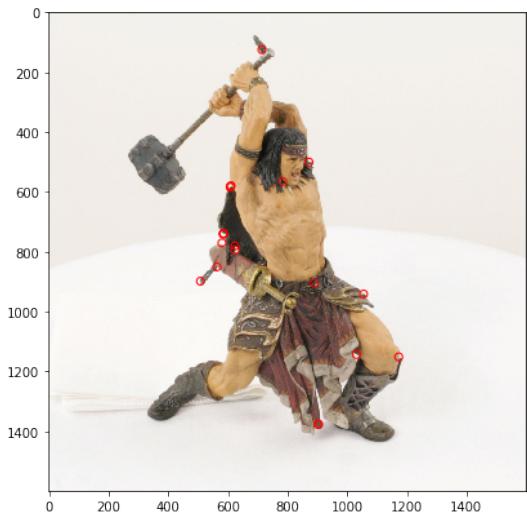
SmoothSTD: 1 WindowSize: 7



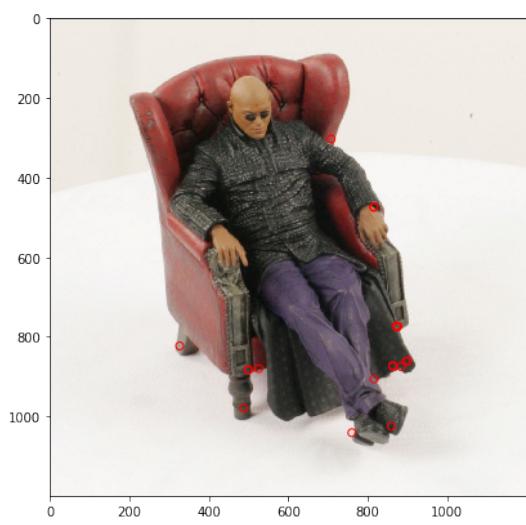
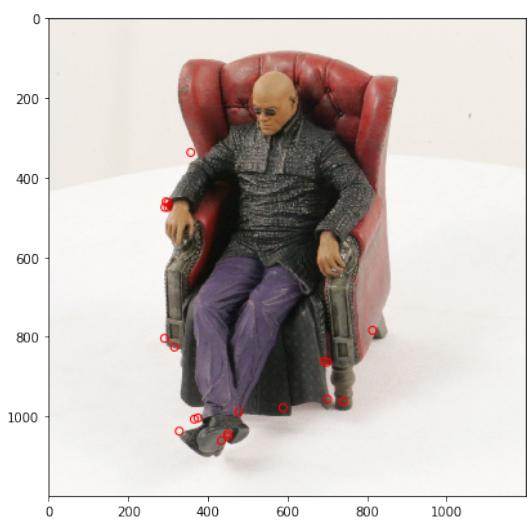


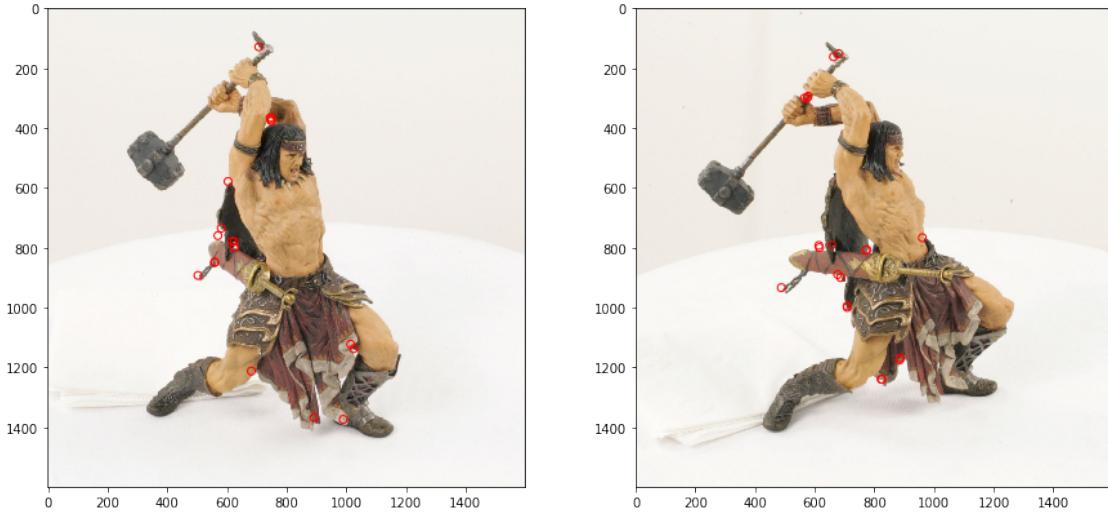
SmoothSTD: 2 WindowSize: 13





SmoothSTD: 4 WindowSize: 25





### 2.0.1 NCC (Normalized Cross-Correlation) Matching [2 pts]

Write a function `ncc_match` that implements the NCC matching algorithm for two input windows.  $NCC = \sum_{i,j} \tilde{W}_1(i,j) \cdot \tilde{W}_2(i,j)$  where  $\tilde{W} = \frac{W - \bar{W}}{\sqrt{\sum_{k,l} (W(k,l) - \bar{W})^2}}$  is a mean-shifted and normalized version of the window and  $\bar{W}$  is the mean pixel value in the window  $W$ .

```
[6]: def ncc_match(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:
    """
    def Wtilde(W):
        mean = np.mean(W)
        W = W - mean
        W = W / math.sqrt(np.sum(W*W))
        return W
```

```

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]
W1 = Wtilde(W1)
W2 = Wtilde(W2)

matching_score = np.sum(W1 * W2)
return matching_score

```

[7]: # test NCC match

```

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 (ncc_match(img1, img2, np.array([1, 1]), np.array([1, 1]), 1))
# should print 0.8546
print (ncc_match(img1, img2, np.array([2, 1]), np.array([2, 1]), 1))
# should print 0.8457
print (ncc_match(img1, img2, np.array([1, 1]), np.array([2, 1]), 1))
# should print 0.6258

```

```

0.8546547739343037
0.8457615282174419
0.6258689611426174

```

## 2.0.2 Naive Matching [4 pts]

Equipped with the corner detector and the NCC matching function, we are ready to start finding correspondances. 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). You will have to figure out a good threshold (NCCth) value by experimentation. Write a function naive\_matching and call it as below. Examine your results for 10, 20, and 30 detected corners in each image. Choose number of detected corners to maximize the number of correct matching pairs. naive\_matching will call your NCC matching code. **Properly label or mention which output corresponds to which choice of number of corners. Total number of output is 6 images** (3 choice of number of corners for each matrix and warrior), where one image is like below:

Number of Corners: 10

[8]: `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]):
    corn1 = corners1[i]
    score_max = -1000
    for j in range(corners2.shape[0]):
        corn2 = corners2[i]
        score = ncc_match(img1, img2, corn1, corn2, R)
        if score > score_max:
            score_max = score
            match = (corn1, corn2)
    if score_max >= NCCth:
        matching.append(match)
return matching
```

[9]: # You are free to modify code here, create your helper functions etc.

```
# detect corners on warrior and matrix sets
nCorners = 10 # Do this for 10, 20 and 30 corners
smoothSTD = 2
windowSize = 13

# 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))
    # downsize your image in case corner_detect runs slow in test
#    imgs_mat.append(rgb2gray(img_mat)[::2, ::2])
    crns_mat.append(corner_detect(imgs_mat[i], nCorners, smoothSTD, windowSize))

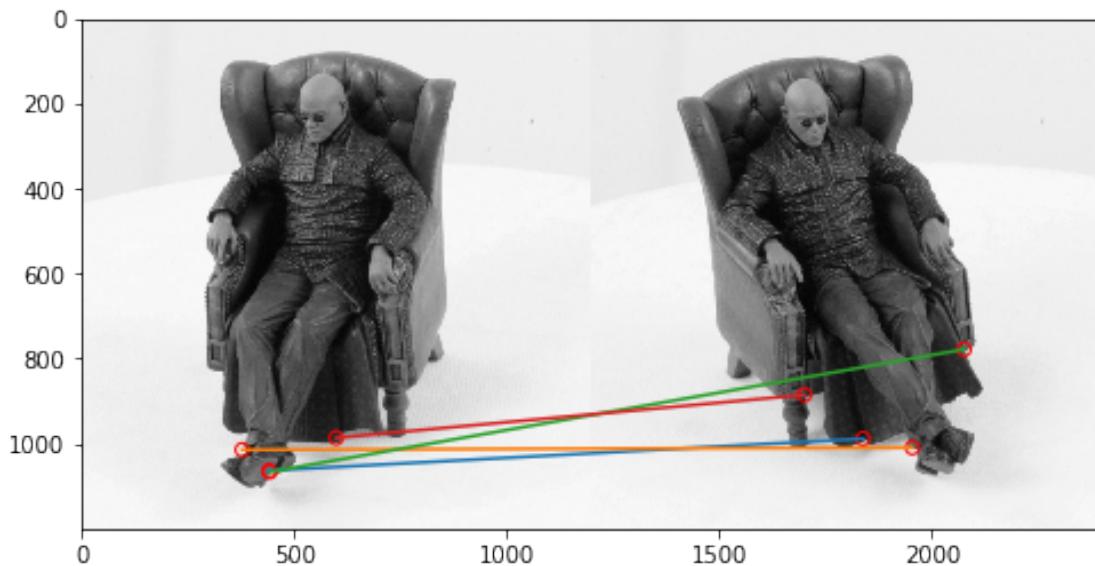
    img_war = imageio.imread('p4/warrior/warrior' + str(i) + '.png')
    imgs_war.append(rgb2gray(img_war))
#    imgs_war.append(rgb2gray(img_war)[::2, ::2])
    crns_war.append(corner_detect(imgs_war[i], nCorners, smoothSTD, windowSize))
```

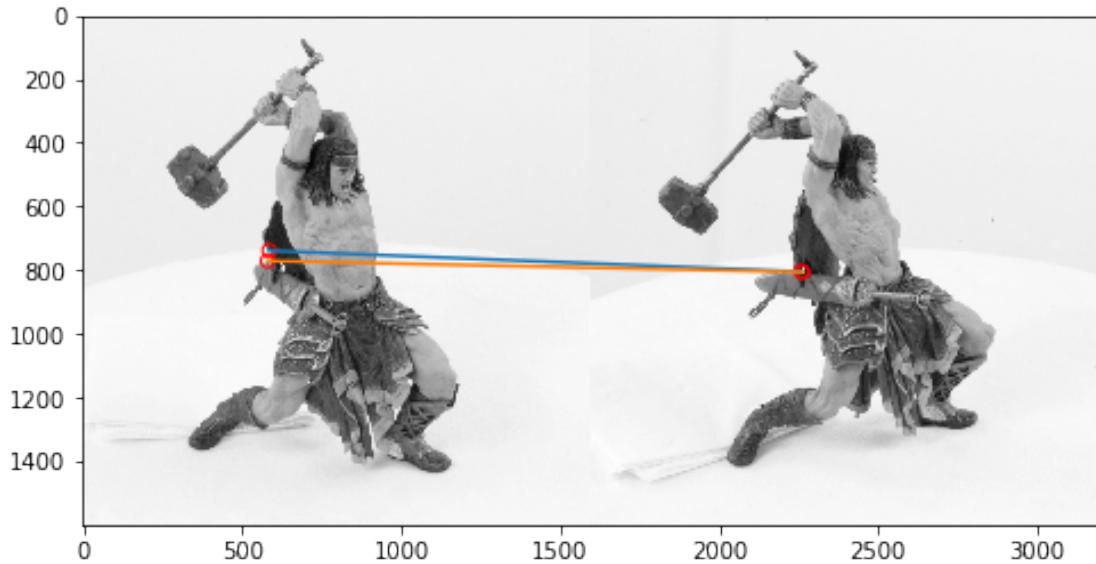
```
[10]: # match corners
R = 15
NCCth_mat = 0.2 # Put your threshold
NCCth_war = 0.001
matching_mat = naive_matching(imgs_mat[0]/255, imgs_mat[1]/255, crns_mat[0], ↵
→crns_mat[1], R, NCCth_mat)
matching_war = naive_matching(imgs_war[0]/255, imgs_war[1]/255, crns_war[0], ↵
→crns_war[1], R, NCCth_war)

[11]: # 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()

# Uncomment to show output
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: 10





```
[16]: # You are free to modify code here, create your helper functions etc.
# detect corners on warrior and matrix sets
nCorners = 20 # Do this for 10, 20 and 30 corners
smoothSTD = 2
windowSize = 13

# 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))
    # downsize your image in case corner_detect runs slow in test
#    imgs_mat.append(rgb2gray(img_mat)[::2, ::2])
    crns_mat.append(corner_detect(imgs_mat[i], nCorners, smoothSTD, windowSize))

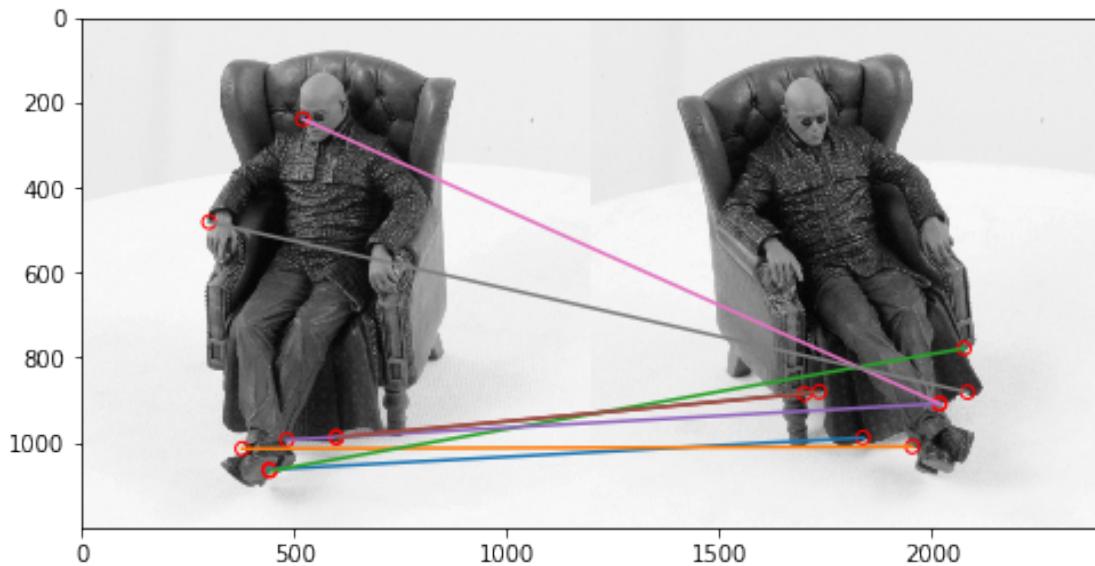
    img_war = imageio.imread('p4/warrior/warrior' + str(i) + '.png')
    imgs_war.append(rgb2gray(img_war))
#    imgs_war.append(rgb2gray(img_war)[::2, ::2])
    crns_war.append(corner_detect(imgs_war[i], nCorners, smoothSTD, windowSize))
```

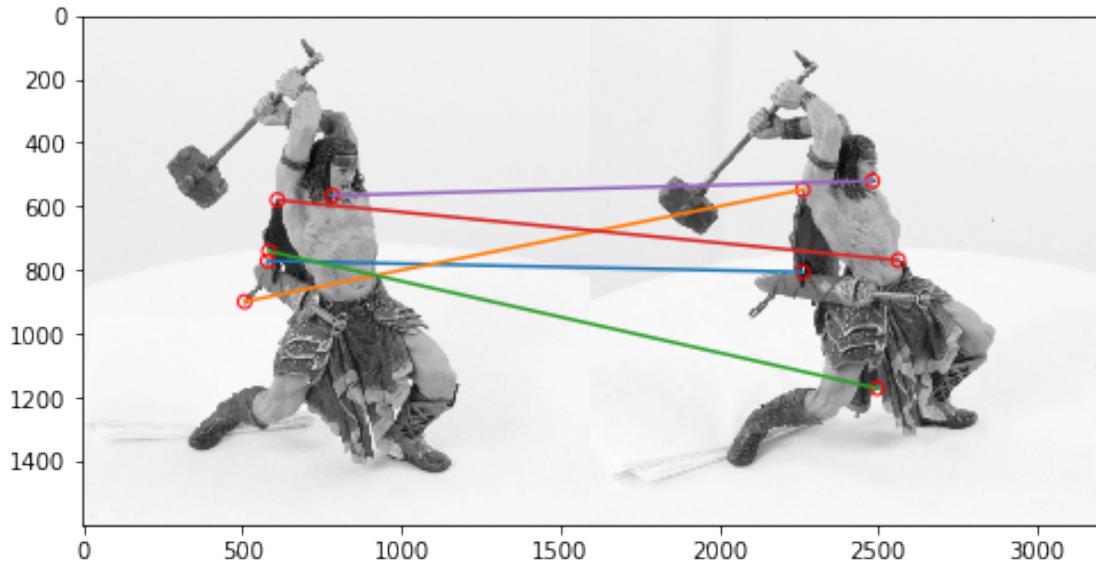
```
[25]: # match corners
R = 15
NCCth_mat = 0.35 # Put your threshold
NCCth_war = 0.25
matching_mat = naive_matching(imgs_mat[0]/255, imgs_mat[1]/255, crns_mat[0], crns_mat[1], R, NCCth_mat)
```

```
matching_war = naive_matching(imgs_war[0]/255, imgs_war[1]/255, crns_war[0],  
→crns_war[1], R, NCCth_war)
```

```
[26]: # 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()  
  
# Uncomment to show output  
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





```
[27]: # You are free to modify code here, create your helper functions etc.
# detect corners on warrior and matrix sets
nCorners = 30 # Do this for 10, 20 and 30 corners
smoothSTD = 2
windowSize = 13

# 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))
    # downsize your image in case corner_detect runs slow in test
#    imgs_mat.append(rgb2gray(img_mat)[::2, ::2])
    crns_mat.append(corner_detect(imgs_mat[i], nCorners, smoothSTD, windowSize))

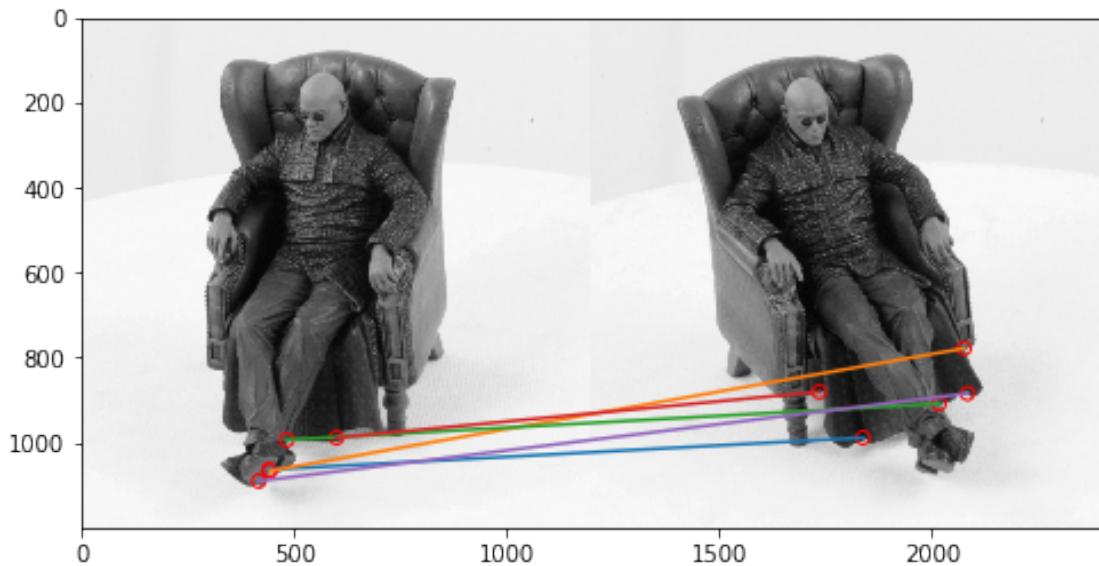
    img_war = imageio.imread('p4/warrior/warrior' + str(i) + '.png')
    imgs_war.append(rgb2gray(img_war))
#    imgs_war.append(rgb2gray(img_war)[::2, ::2])
    crns_war.append(corner_detect(imgs_war[i], nCorners, smoothSTD, windowSize))
```

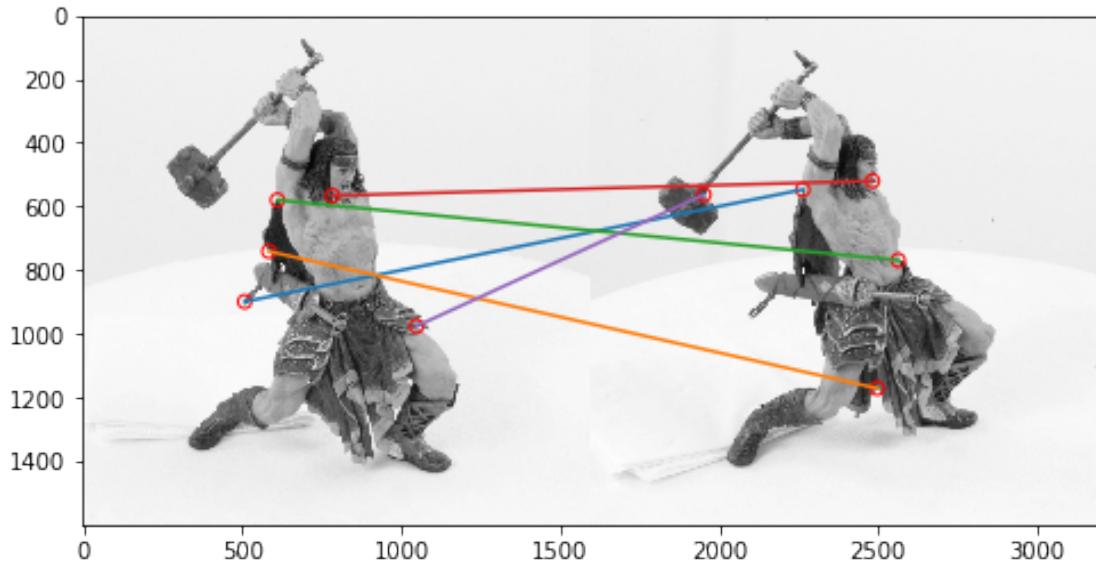
```
[28]: # match corners
R = 15
NCCth_mat = 0.5 # Put your threshold
NCCth_war = 0.33
matching_mat = naive_matching(imgs_mat[0]/255, imgs_mat[1]/255, crns_mat[0], crns_mat[1], R, NCCth_mat)
```

```
matching_war = naive_matching(imgs_war[0]/255, imgs_war[1]/255, crns_war[0],  
→crns_war[1], R, NCCth_war)
```

```
[29]: # 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()  
  
# Uncomment to show output  
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: 30





### 2.0.3 Epipolar Geometry [4 pts]

Complete the compute\_fundamental function below using 8 point algorithm described in [Lecture 8](#). Using the fundamental\_matrix function and the corresponding points provided in cor1.npy and cor2.npy, calculate the fundamental matrix for the set of matrix and warrior image. Note that the normalization of the corner point is handled in the fundamental\_matrix function.

```
[30]: import numpy as np
# from scipy.misc import imread
import matplotlib.pyplot as plt
from scipy.io import loadmat

def compute_fundamental(x1,x2):
    """
        Computes the fundamental matrix from corresponding points
        (x1,x2 3*n arrays) using the 8 point algorithm.
        Each row in the A matrix below is constructed as
        [x'*x, x'*y, x', y'*x, y'*y, y', x, y, 1]

    Returns:
        Fundamental Matrix (3x3)
    """

    n = x1.shape[1]
    if x2.shape[1] != n:
        raise ValueError("Number of points don't match.")
    # return your F matrix
    A = np.zeros((n, 9))
```

```

for i in range(n):
    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]]
    U, S, 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))
    return F/F[2, 2]

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)

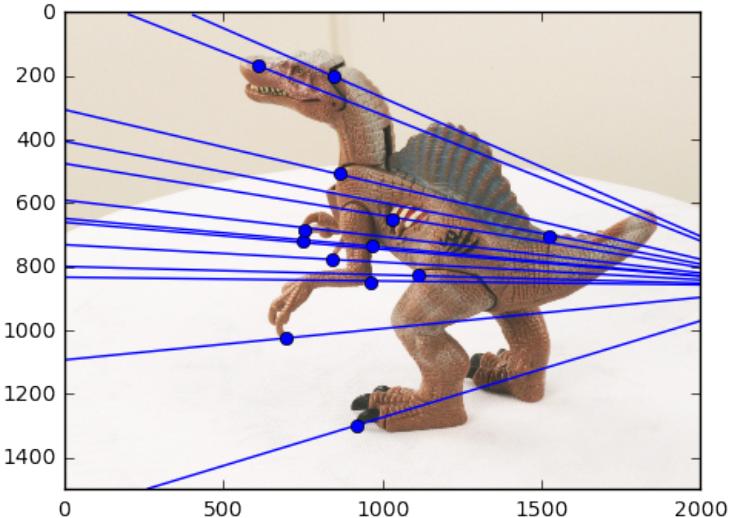
    # reverse normalization
    F = np.dot(T1.T,np.dot(F,T2))

    return F/F[2,2]

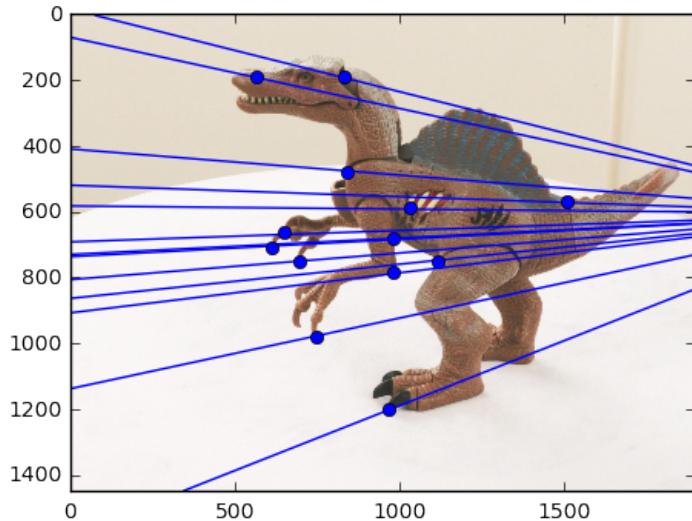
```

#### 2.0.4 Plot Epipolar Lines [5 pts]

Using this fundamental matrix, plot the epipolar lines in both image pairs across all images. For this part you may want to complete the function `plot_epipolar_lines`. Show your result for ma-



trix and warrior as the figure below.



Also, write the script to calculate the epipoles for a given Fundamental matrix and corner point correspondences in the two images.

```
[31]: def compute_epipole(F):
    """
    This function computes the epipoles for a given fundamental matrix
    and corner point correspondences
    input:
    F--> Fundamental matrix
    output:
    e1--> corresponding epipole in image 1
    e2--> epipole in image2
    """
    #your code here
    e1 = np.linalg.svd(F.T)[-1]
    e1 = e1[-1, :] / e1[-1, :][-1]
    e2 = np.linalg.svd(F)[-1]
```

```

e2 = e2[-1, :]/e2[-1, :][-1]
return e1,e2

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)

"""

"""

Your code here:
"""

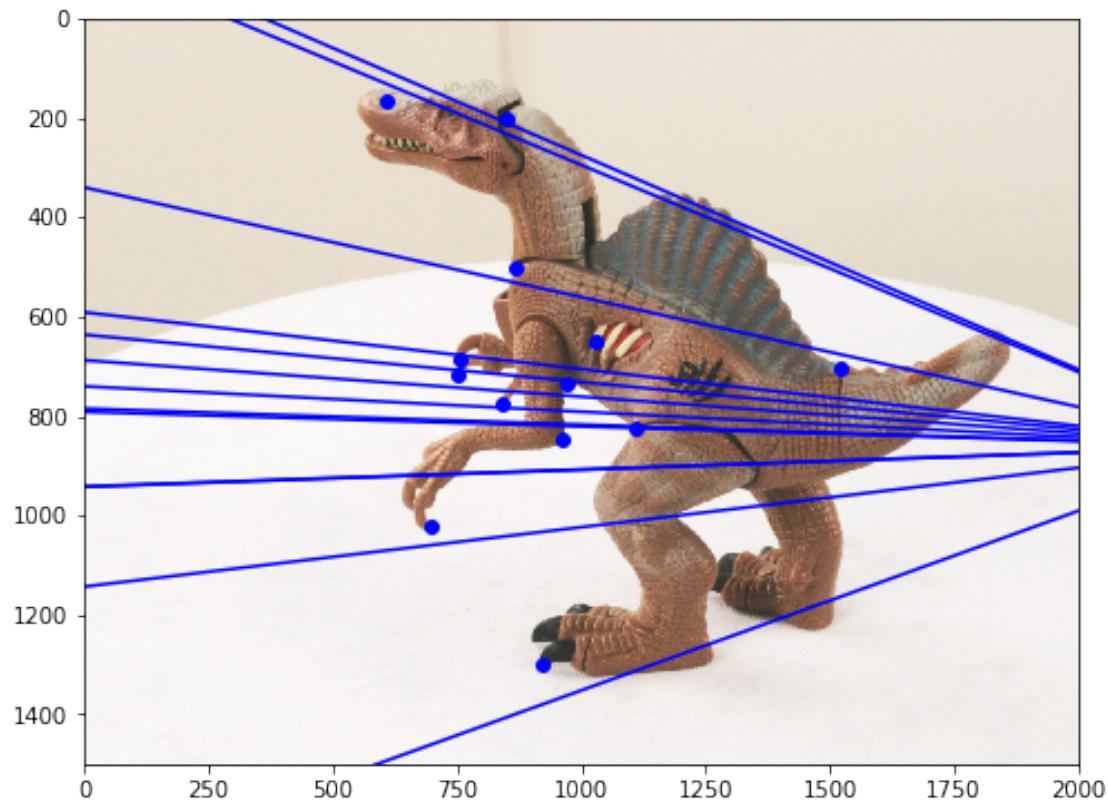
F = fundamental_matrix(cor1, cor2)
fig1 = plt.figure(figsize = (8, 8))
plt.imshow(img1, cmap = 'gray')
for i in range(cor1.shape[1]):
    plt.scatter(cor1[0][i], cor1[1][i], s=35, edgecolors='b', facecolors='b')
for i in range(cor2.shape[1]):
    l1 = np.dot(F, cor1[:, i])
    start = l1[2]/(-l1[1])
    end = (img1.shape[1]*l1[0] + l1[2])/(-l1[1])
    plt.plot([0, img1.shape[1]], [start, end], color = 'b')
    plt.axis([0, img1.shape[1], img1.shape[0], 0])
plt.show()

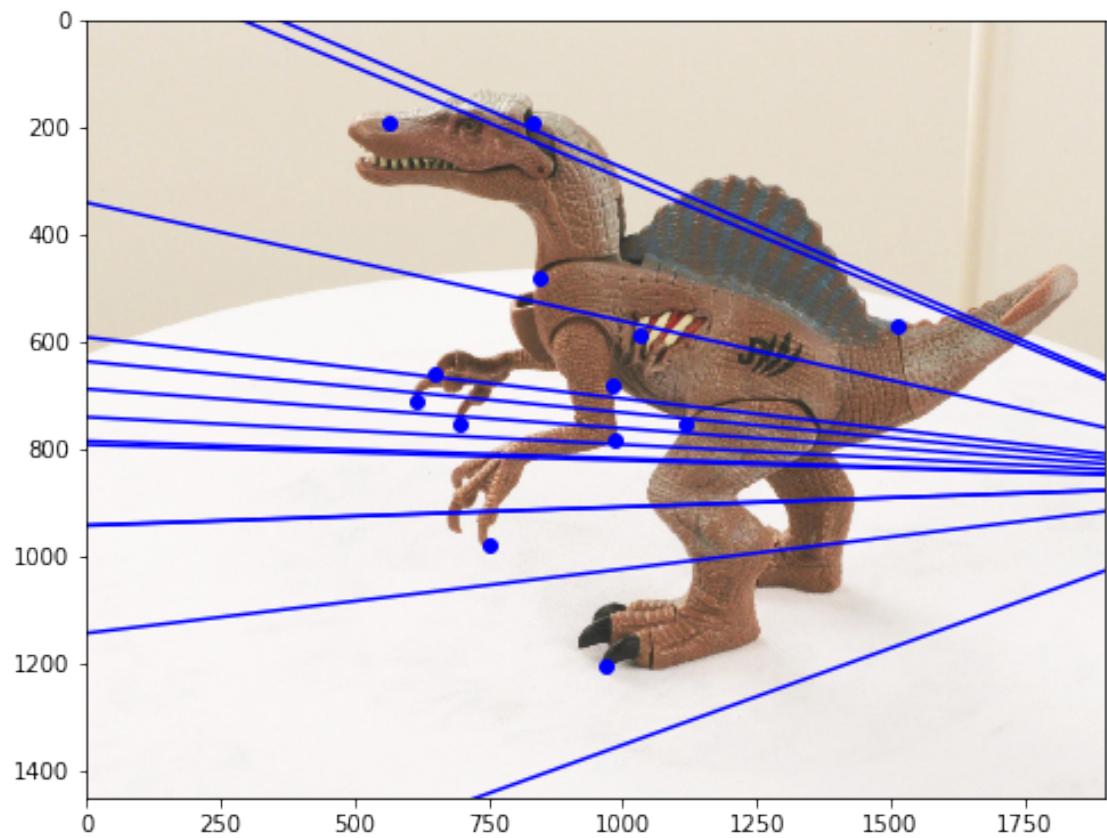
fig2 = plt.figure(figsize = (8, 8))
plt.imshow(img2, cmap = 'gray')
for i in range(cor2.shape[1]):
    plt.scatter(cor2[0][i], cor2[1][i], s=35, edgecolors='b', facecolors='b')
for i in range(cor1.shape[1]):
    l2 = np.dot(F, cor1[:, i])
    start = l2[2]/(-l2[1])
    end = (img2.shape[1]*l2[0] + l2[2])/(-l2[1])
    plt.plot([0, img2.shape[1]], [start, end], color = 'b')
    plt.axis([0, img2.shape[1], img2.shape[0], 0])
plt.show()

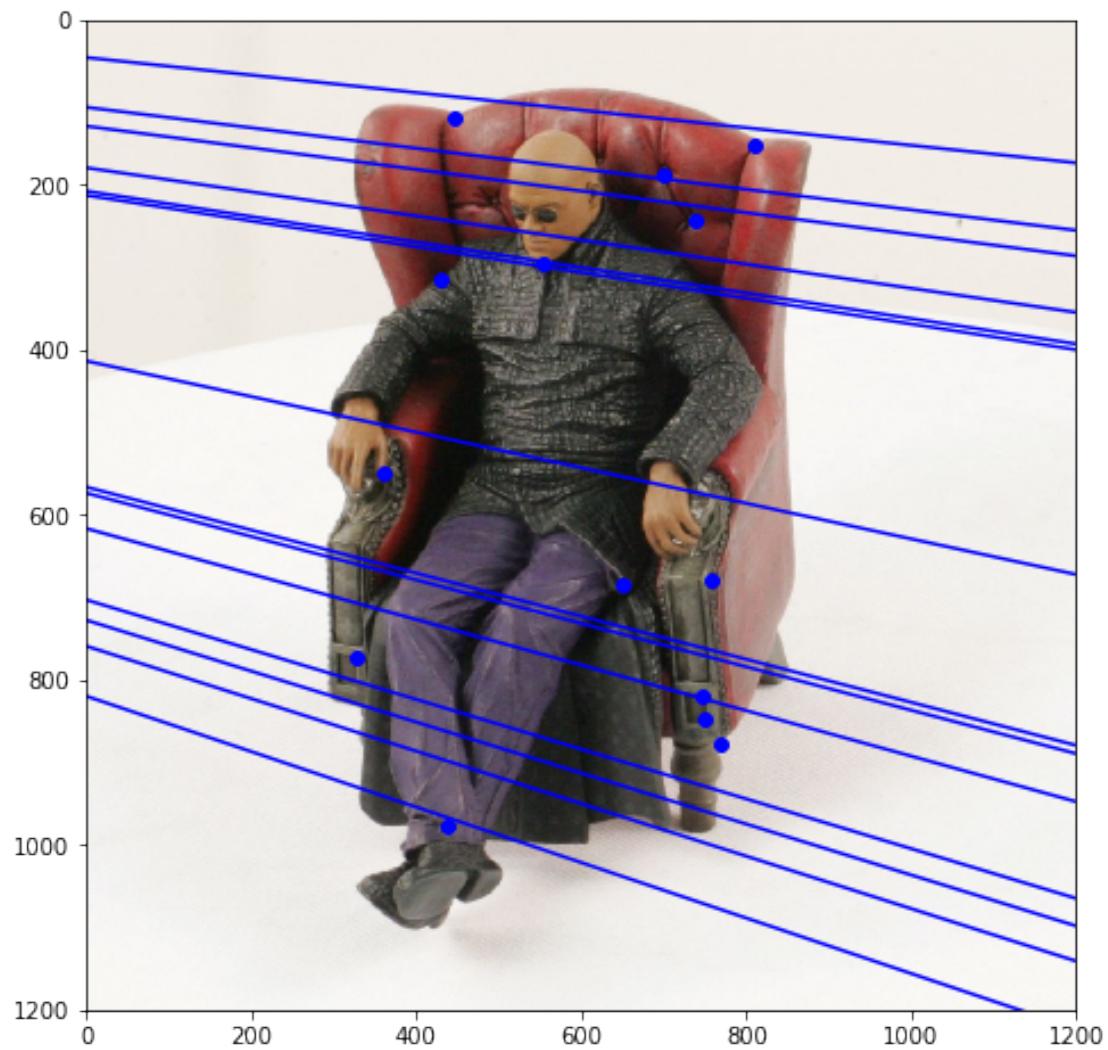
```

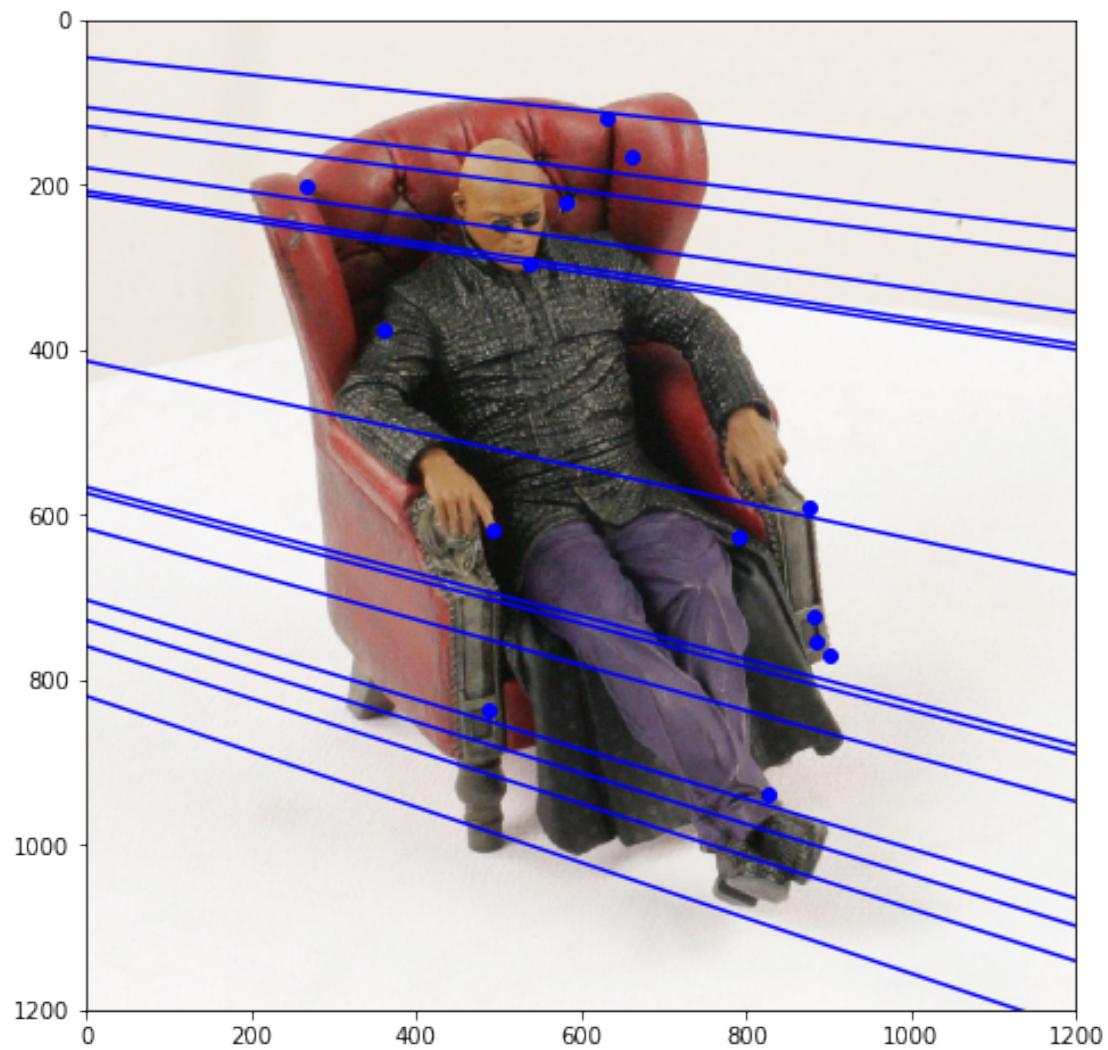
```
[32]: # replace images and corners with those of matrix and warrior
imgids = ["dino", "matrix", "warrior"]
for imgid in imgids:
    I1 = imageio.imread("./p4/" + imgid + "/" + imgid + "0.png")
    I2 = imageio.imread("./p4/" + imgid + "/" + imgid + "1.png")

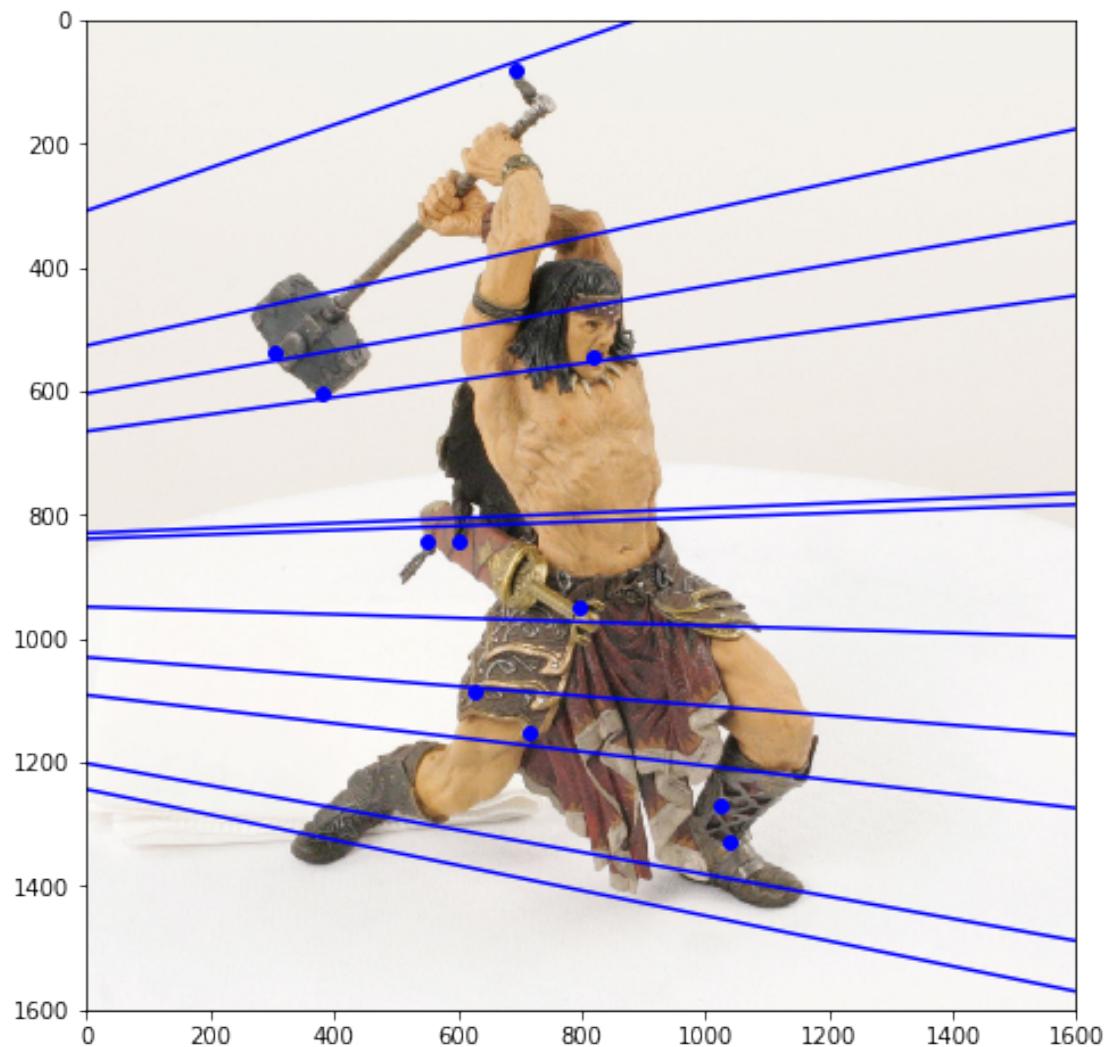
    cor1 = np.load("./p4/" + imgid + "/cor1.npy")
    cor2 = np.load("./p4/" + imgid + "/cor2.npy")
    plot_epipolar_lines(I1,I2,cor1,cor2)
```

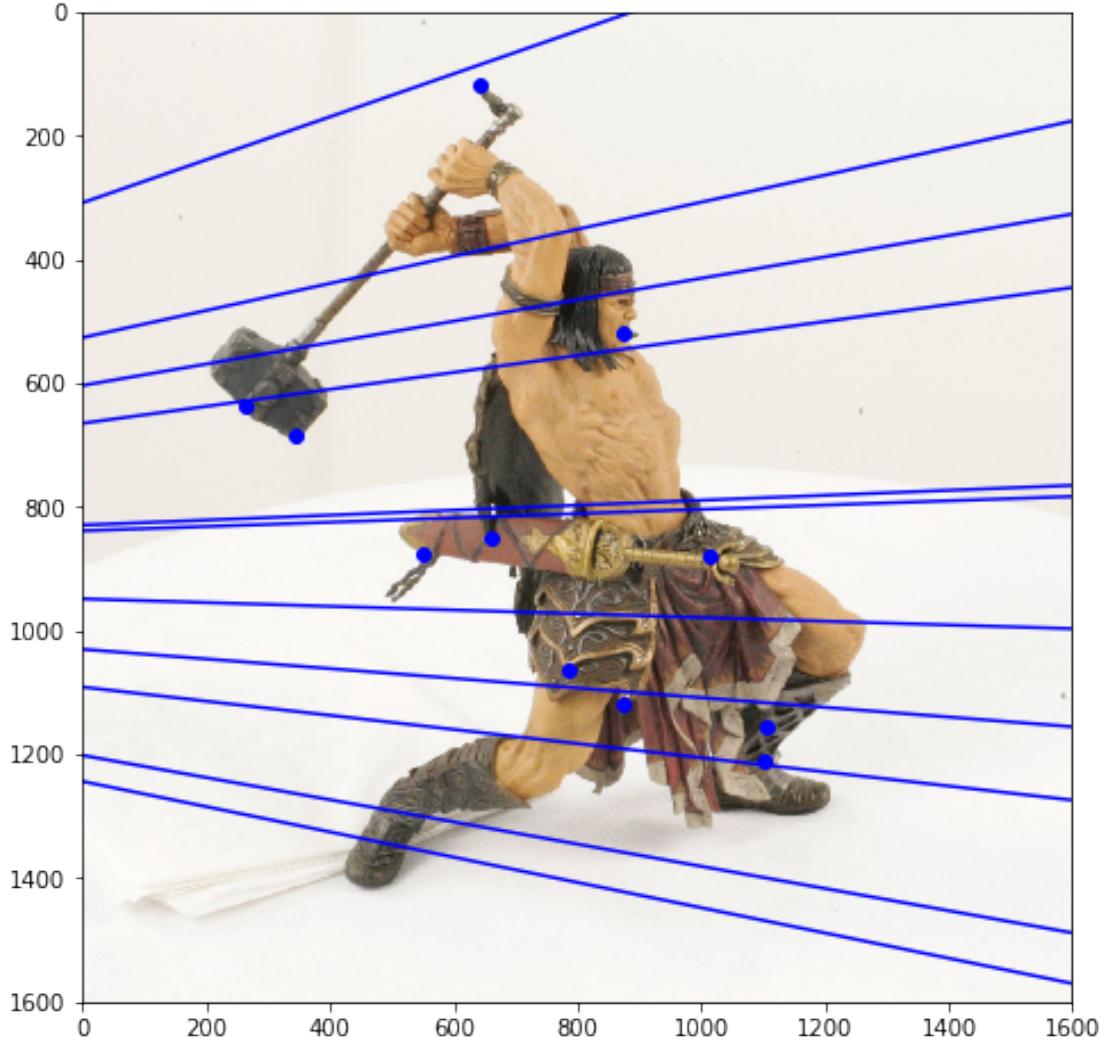








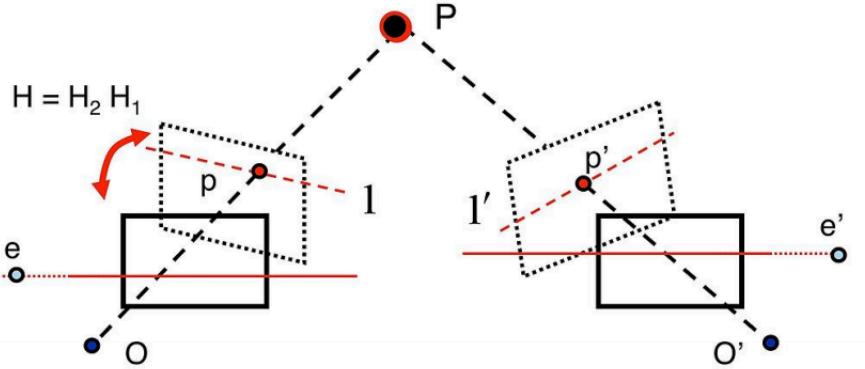




### 2.0.5 Image Rectification [5 pts]

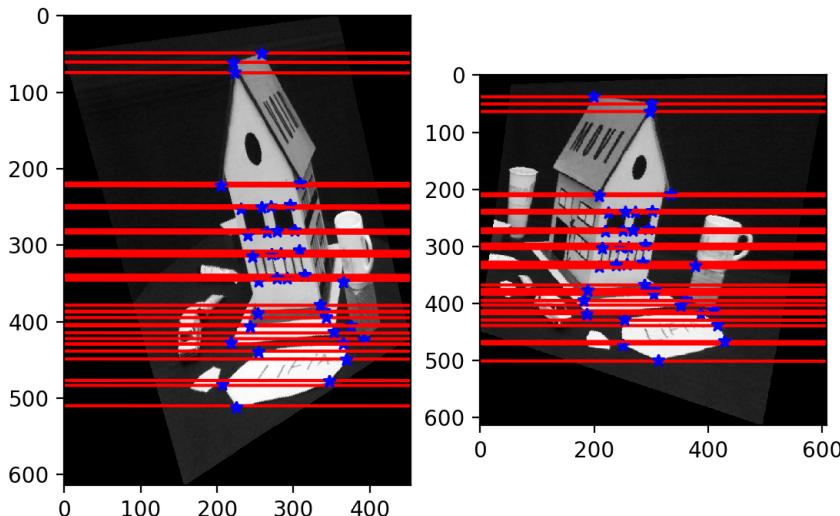
An interesting case for epipolar geometry occurs when two images are parallel to each other. In this case, there is no rotation component involved between the two images and the essential matrix is  $E = [T_x]R = [T_x]$ . Also if you observe the epipolar lines  $l$  and  $l'$  for parallel images, they 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 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_epipoles` 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. A sample output will look as below:



```
[33]: 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

def to_homog(points):
    return np.vstack((points, np.ones(np.size(points, 1)))))

def from_homog(points_homog):
    m = np.divide(points_homog, points_homog[-1])
    return np.delete(m, (-1), axis = 0)

def rectify_img(H, img):
    target_image = np.ones(img.shape)

    for x in range(0, img.shape[1]):
        for y in range(0, img.shape[0]):
            # convert to homogeneous coordinate
            xy_homo = to_homog(np.array([[x, y]]).T)
            # apply homography
            mapped_x, mapped_y = from_homog(np.dot(H, xy_homo)).T[0]
            if 0<=int(mapped_y)<img.shape[0] and \
                0<=int(mapped_x)<img.shape[1]:
                target_image[y, x, :] = img[int(mapped_y), int(mapped_x), :]
    return target_image

def rectify_points(H, points):
    for i in range(points.shape[1]):
        points[:, i] = np.dot(H, points[:, i])
        points[:, i] = points[:, i]/points[:, i][-1]
    return points

def image_rectification(im1, im2, points1, points2):
    '''this function provides the rectified images along with the new
    corner points as outputs for a given pair of 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
'''

F = fundamental_matrix(points1, points2)
e1, e2 = compute_epipole(F)
H1, H2 = compute_matching_homographies(e2, F.T, im2, points1.T, points2.T)
rectified_im1 = rectify_img(np.linalg.inv(H1), im1/np.max(im1))
rectified_im2 = rectify_img(np.linalg.inv(H2), im2/np.max(im2))
new_cor1 = rectify_points(H1, points1)
new_cor2 = rectify_points(H2, points2)

return rectified_im1, rectified_im2, new_cor1, new_cor2

```

[34]: # replace images and corners with those of matrix and warrior

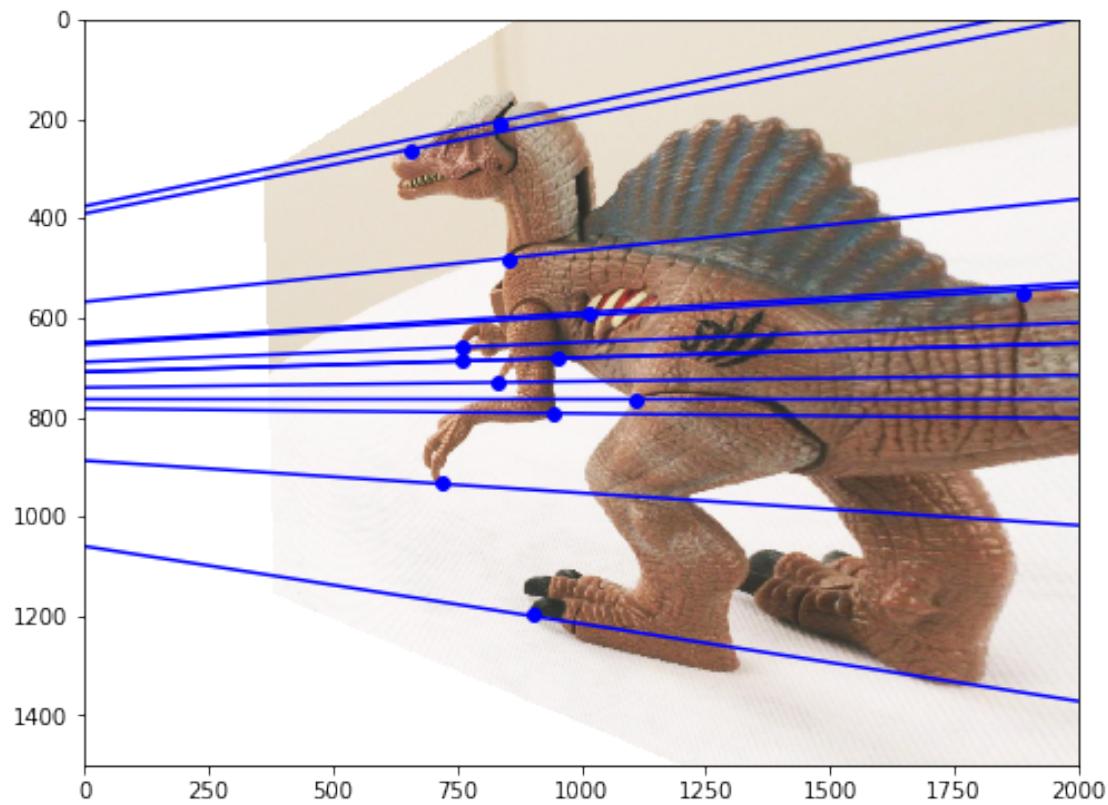
```

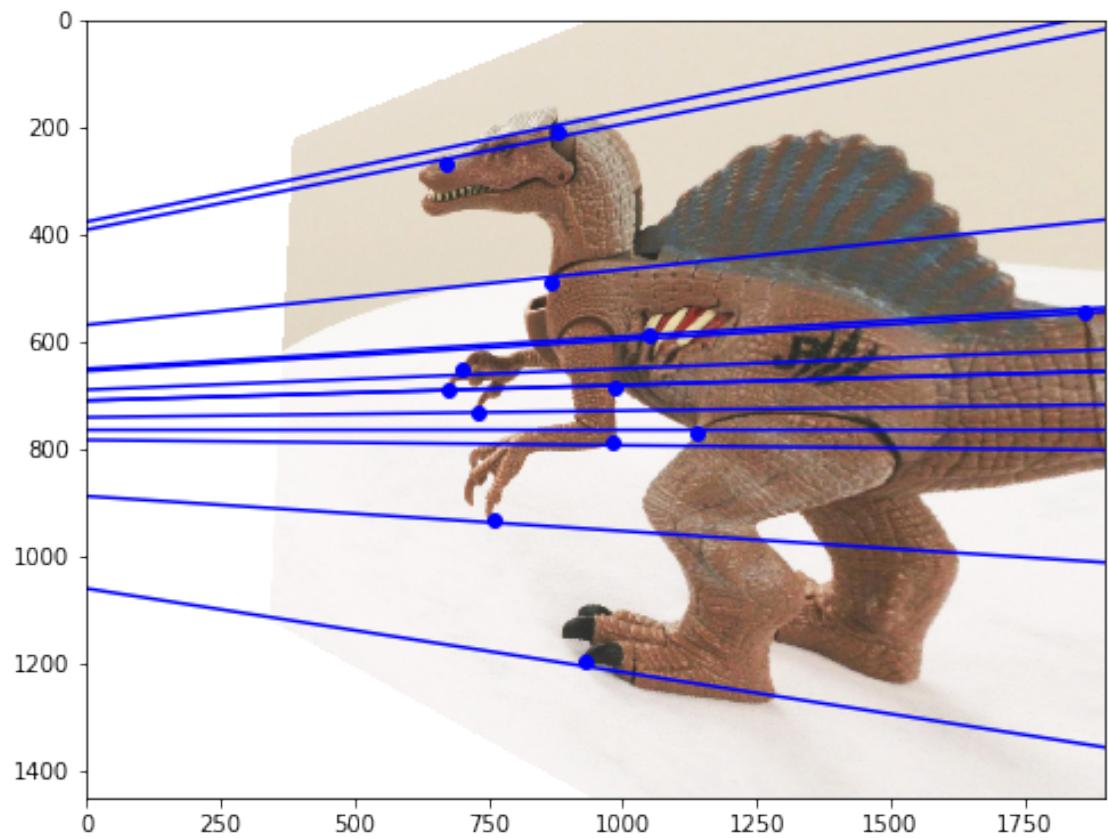
imgids = ["dino", "matrix", "warrior"]
for imgid in imgids:
    I1 = imageio.imread("./p4/" + imgid + "/" + imgid + "0.png")
    I2 = imageio.imread("./p4/" + imgid + "/" + imgid + "1.png")

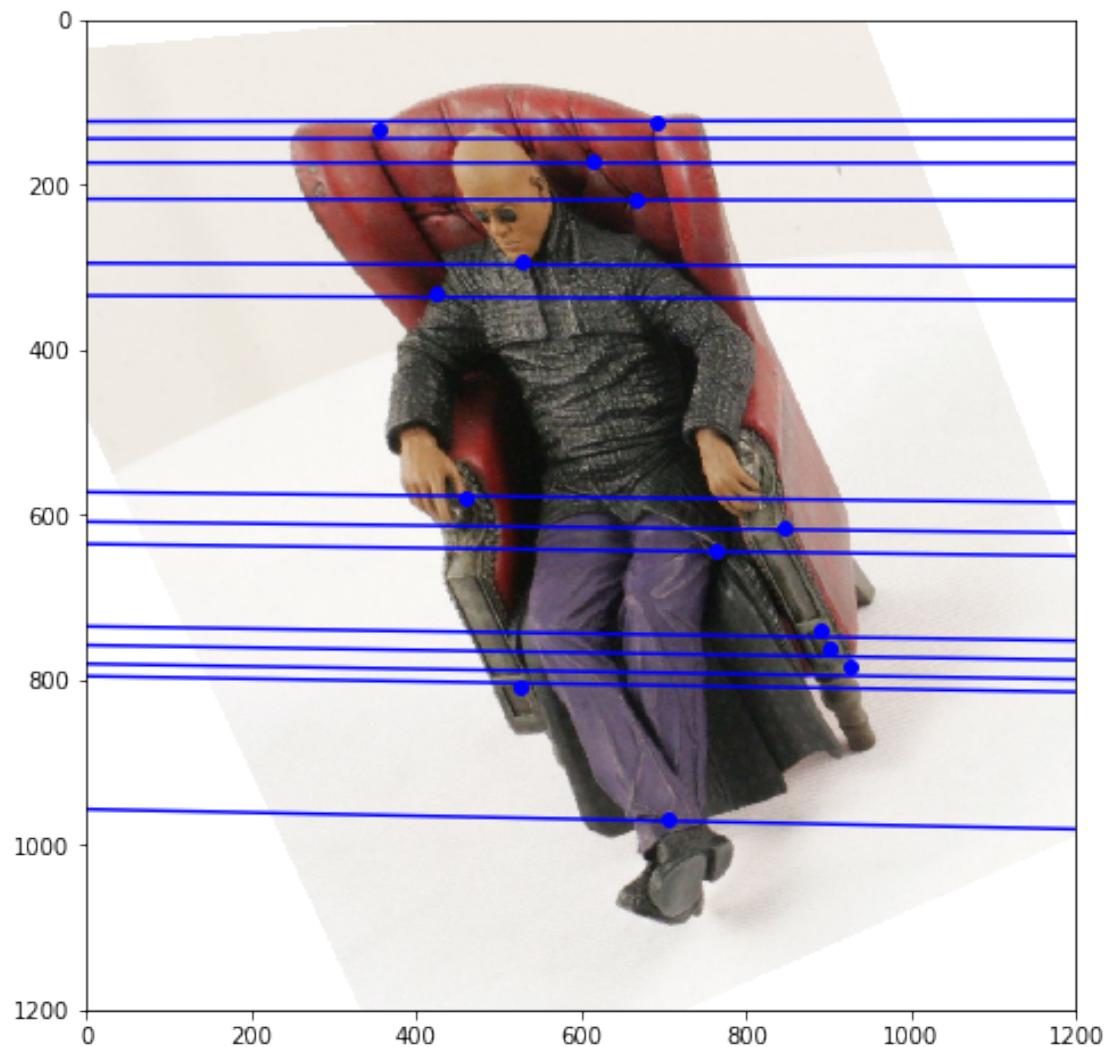
    cor1 = np.load("./p4/" + imgid + "/cor1.npy")
    cor2 = np.load("./p4/" + imgid + "/cor2.npy")
    I1, I2, cor1, cor2 = image_rectification(I1, I2, cor1, cor2)
    plot_epipolar_lines(I1, I2, cor1, cor2)

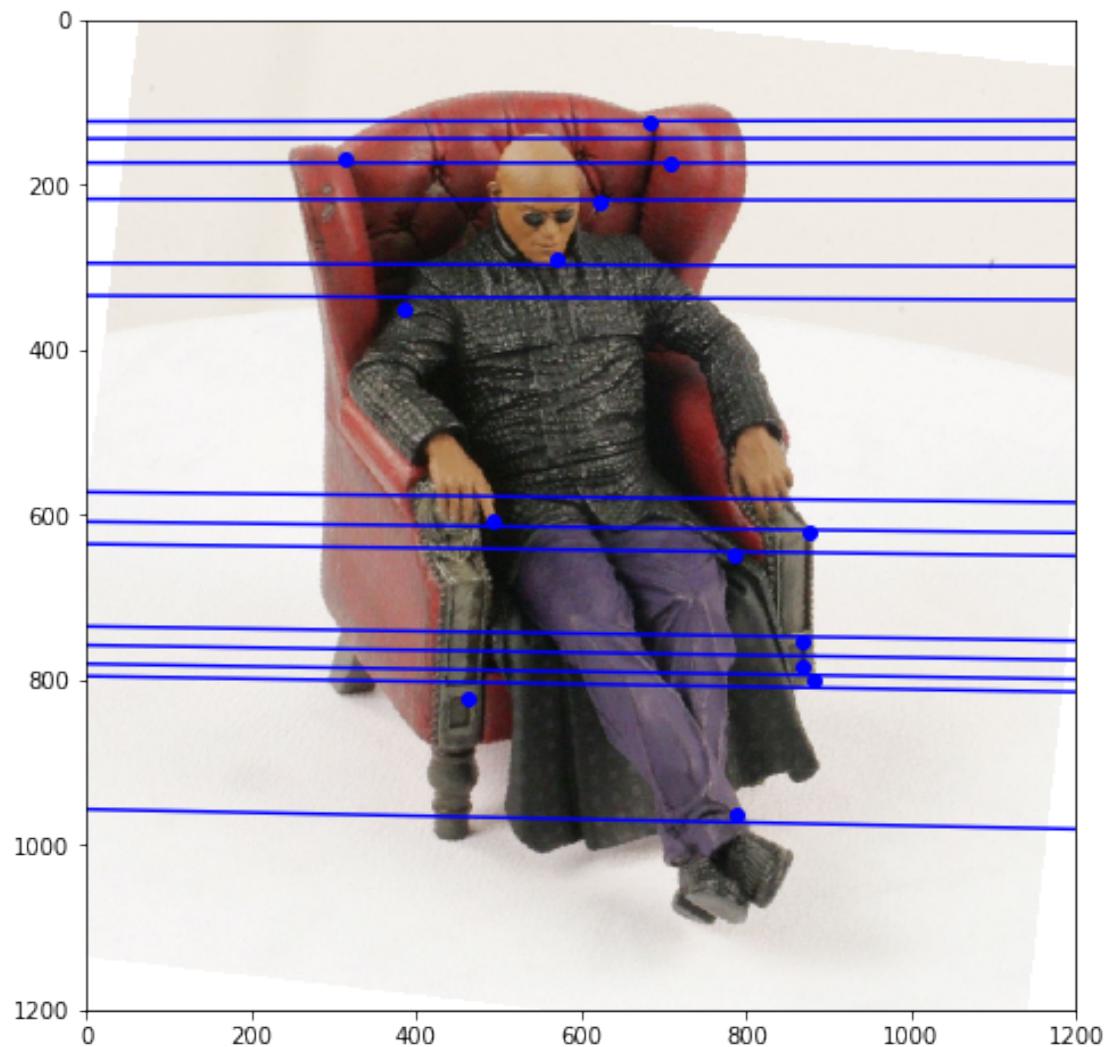
```

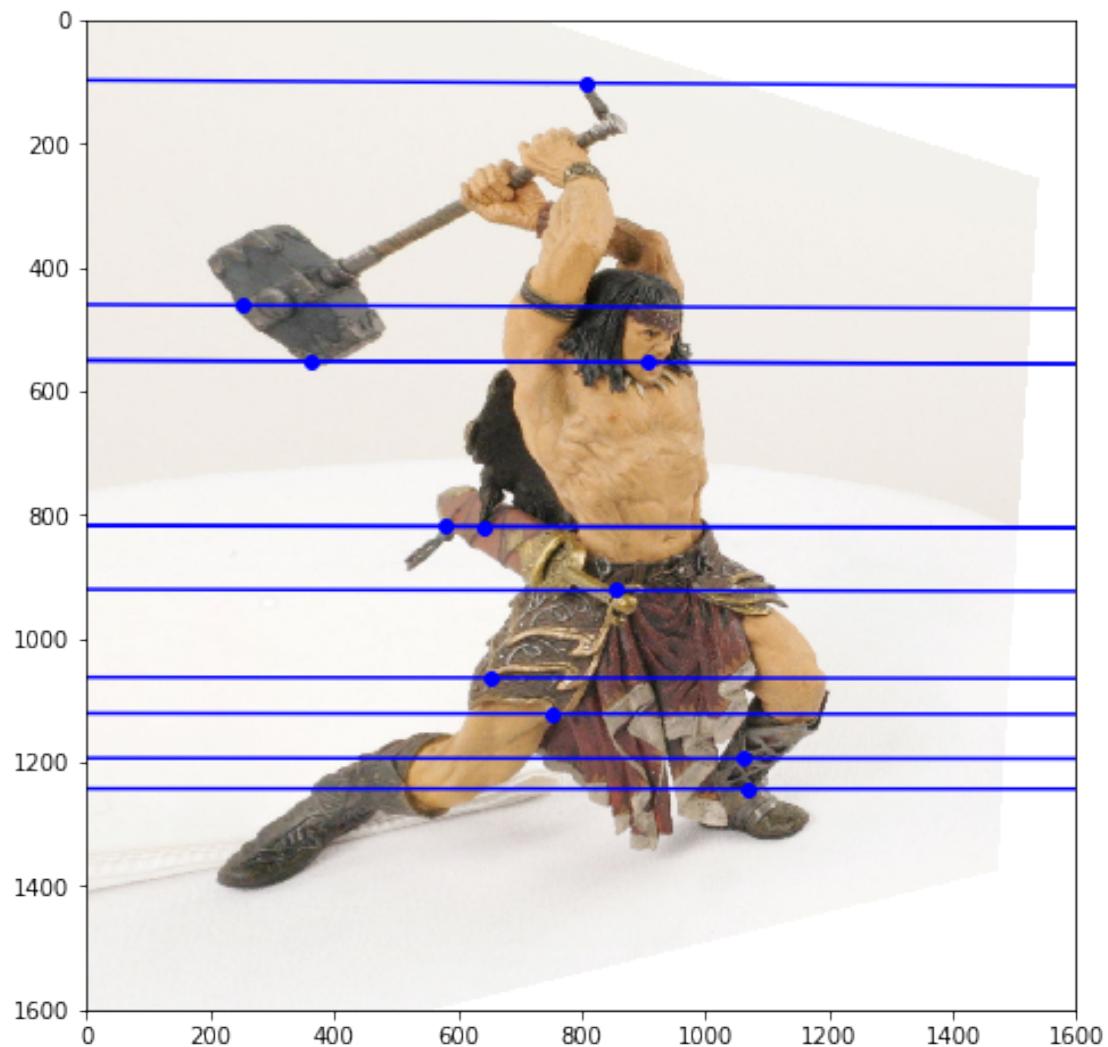
/Users/apple/anaconda3/lib/python3.7/site-packages/ipykernel\_launcher.py:61:  
FutureWarning: `rcond` parameter will change to the default of machine precision  
times ``max(M, N)`` where M and N are the input matrix dimensions.  
To use the future default and silence this warning we advise to pass  
`rcond=None`, to keep using the old, explicitly pass `rcond=-1`.

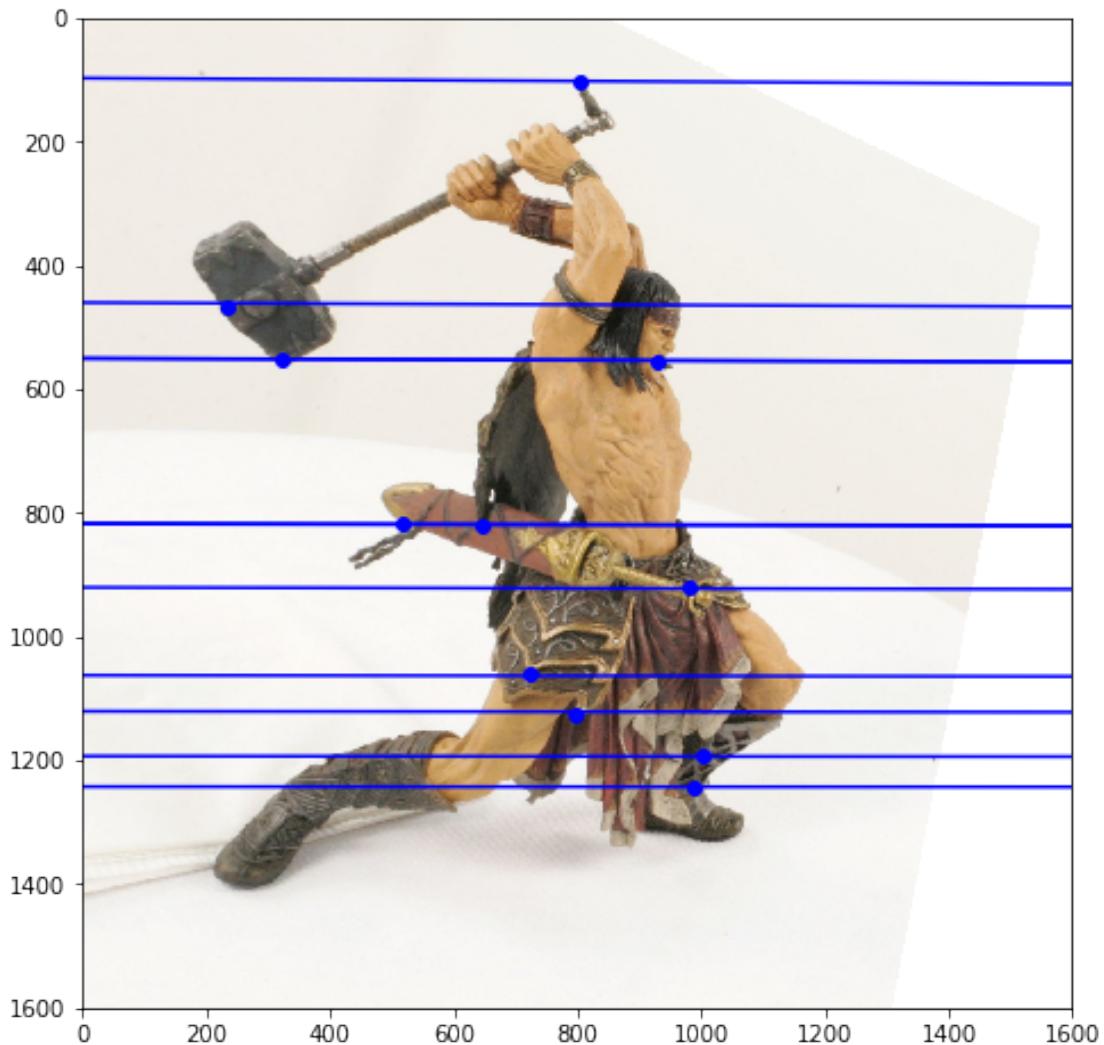












## 2.0.6 Matching Using epipolar geometry[4 pts]

We will now use the epipolar geometry constraint on the rectified images and updated corner points to build a better matching algorithm. First, detect 10 corners in Image1. Then, for each corner, do a linesearch along the corresponding parallel epipolar line in Image2. Evaluate the NCC score for each point along this line and return the best match (or no match if all scores are below the NCCth). R is the radius (size) of the NCC patch in the code below. You do not have to run this in both directions. Show your result as in the naive matching part. Execute this for the warrior and matrix images (**Total two outputs images**).

```
[35]: def display_correspondence(img1, img2, corrs, name):
    """Plot matching result on image pair given images and correspondences
```

*Args:*

*img1: Image 1.*

```



```

```

def correspondence_matching_epipole(img1, img2, corners1, F, R, NCCth):
    """Find corner correspondence along epipolar line.

```

*Args:*

*img1: Image 1.*  
*img2: Image 2.*  
*corners1: Detected corners in image 1.*  
*F: Fundamental matrix calculated using given*  
*ground truth corner correspondences.*  
*R: NCC matching window radius.*  
*NCCth: NCC matching threshold.*

*Returns:*

*Matching result to be used in display\_correspondence function*

```

"""

matching = []
corners1 = to_homog(corners1.T)
for i in range(0, corners1.shape[1]):
    a = np.dot(F.T, corners1[:, i])
    score_max = -100
    match_pair = None
    # we will use line equation Ax+By+C = 0
    for x in range(R, img2.shape[1]-R):
        y = int((a[0]*x + a[2])/(-a[1]))
        if y < R or y > img2.shape[0]-R:
            continue
        corn1 = corners1[:2, i].astype(np.int32)
        corn2 = np.array([x, y]).astype(np.int32)
        # calculate ncc match similarity

```

```

        score = ncc_match(img1, img2, corn1, corn2, R)
        if score > score_max:
            score_max = score
            match_pair = (corn1, corn2)

    if score_max > NCCth:
        matching.append(match_pair)

return matching

```

[36]:

```

I1 = imageio.imread("./p4/matrix/matrix0.png")
I2 = imageio.imread("./p4/matrix/matrix1.png")
cor1 = np.load("./p4/matrix/cor1.npy")
cor2 = np.load("./p4/matrix/cor2.npy")
I3 = imageio.imread("./p4/warrior/warrior0.png")
I4 = imageio.imread("./p4/warrior/warrior1.png")
cor3 = np.load("./p4/warrior/cor1.npy")
cor4 = np.load("./p4/warrior/cor2.npy")

```

[37]:

```

# For matrix
rectified_im1, rectified_im2, new_cor1, new_cor2 = image_rectification(I1, I2, cor1, cor2)
F_new = fundamental_matrix(new_cor1, new_cor2)

```

/Users/apple/anaconda3/lib/python3.7/site-packages/ipykernel\_launcher.py:61:  
FutureWarning: `rcond` parameter will change to the default of machine precision  
times ``max(M, N)`` where M and N are the input matrix dimensions.  
To use the future default and silence this warning we advise to pass  
`rcond=None`, to keep using the old, explicitly pass `rcond=-1`.

[38]:

```

I1 = imageio.imread("./p4/matrix/matrix0.png")
I2 = imageio.imread("./p4/matrix/matrix1.png")
cor1 = np.load("./p4/matrix/cor1.npy")
cor2 = np.load("./p4/matrix/cor2.npy")

rectified_im1, rectified_im2, new_cor1, new_cor2 = image_rectification(I1, I2, cor1, cor2)
F_new = fundamental_matrix(new_cor1, new_cor2)

nCorners = 10
#decide the NCC matching window radius
R = 10
NCCTh = 0.6
# detect corners using corner detector here, store in corners1
corners1 = corner_detect(rgb2gray(rectified_im1), nCorners, smoothSTD,
                           windowSize)

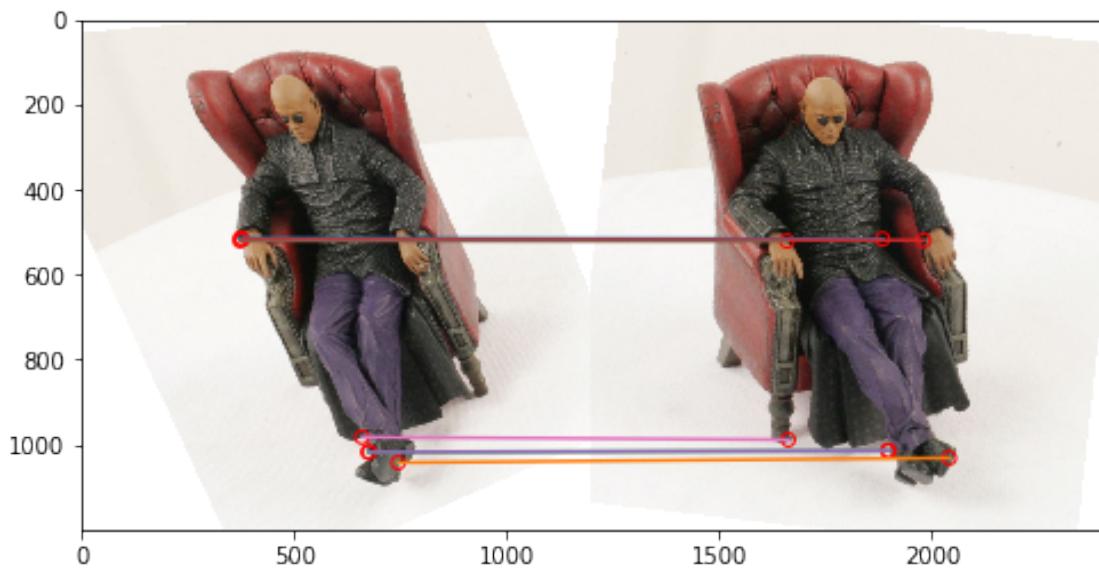
```

/Users/apple/anaconda3/lib/python3.7/site-packages/ipykernel\_launcher.py:61:

FutureWarning: `rcond` parameter will change to the default of machine precision times ``max(M, N)`` where M and N are the input matrix dimensions.  
To use the future default and silence this warning we advise to pass `rcond=None`, to keep using the old, explicitly pass `rcond=-1`.

```
[39]: corrs = correspondence_matching_epipole(
    rectified_im1, rectified_im2, corners1, F_new, R, NCCth)
display_correspondence(rectified_im1, rectified_im2, corrs, name="matrix")
```

/Users/apple/anaconda3/lib/python3.7/site-packages/ipykernel\_launcher.py:22:  
RuntimeWarning: invalid value encountered in true\_divide



```
[40]: # For warrior
rectified_im3, rectified_im4, new_cor3, new_cor4 =
    image_rectification(I3, I4, cor3, cor4)
F_new2=fundamental_matrix(new_cor3, new_cor4)
```

/Users/apple/anaconda3/lib/python3.7/site-packages/ipykernel\_launcher.py:61:  
FutureWarning: `rcond` parameter will change to the default of machine precision times ``max(M, N)`` where M and N are the input matrix dimensions.  
To use the future default and silence this warning we advise to pass `rcond=None`, to keep using the old, explicitly pass `rcond=-1`.

```
[41]: # You may wish to change your NCCth and R for warrior here.
corners2 = corner_detect(rgb2gray(rectified_im3), nCorners, smoothSTD,
    windowSize)
```

```
[42]: corrs = correspondence_matching_epipole(rectified_im3, rectified_im4, corners2,  
      →F_new2, R, NCCth)  
display_correspondence(rectified_im3, rectified_im4, corrs ,name="warrior")
```

/Users/apple/anaconda3/lib/python3.7/site-packages/ipykernel\_launcher.py:22:  
RuntimeWarning: invalid value encountered in true\_divide

