CSE 252A Computer Vision I Fall 2018 - Assignment 3

Instructor: David Kriegman

Assignment Published On: Wednesday, November 7, 2018

Due On: Tuesday, November 20, 2018 11:59 pm

Instructions

- Review the academic integrity and collaboration policies on the course website.
- This assignment must be completed individually.
- This assignment contains theoretical and programming exercises. If you plan to submit hand written answers for theoretical exercises, please be sure your writing is readable and merge those in order with the final pdf you create out of this notebook. You could fill the answers within the notebook iteself by creating a markdown cell. Please do not mention your explanatory answers in code comments.
- Programming aspects of this assignment must be completed using Python in this notebook.
- If you want to modify the skeleton code, you can do so. This has been provided just to provide you with a framework for the solution.
- You may use python packages for basic linear algebra (you can use numpy or scipy for basic operations), 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 teaching assistants for clarification.
- You must submit this notebook exported as a pdf. You must also submit this notebook as .ipynb file.
- You must submit both files (.pdf and .ipynb) on Gradescope. You must mark each problem on Gradescope in the pdf.
- **Late policy** - 10% per day late penalty after due date up to 3 days.

Problem 1: Epipolar Geometry [3 pts]

Consider two cameras whose image planes are the z=1 plane, and whose focal points are at (-20, 0, 0) and (20, 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 (-20, 0, 1) in world coordinates, while if (u, v) = (0, 0) this is the point (20, 0, 1).<img src = "fig/fig1.png" width="500"> a) Suppose the points (x, y) = (12, 12) is matched to the point (u, v) = (1, 12). What is the 3D location of this point?

b) Consider points that lie on the line x + z = 0, 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.

Solution P1

(a)

For the two given points $(-20, 0, 1)$$(-20, 0, 1)$ and $(20, 0, 1)$$(20, 0, 1)$ are corresponding to the points in the image (0,0) and (0,0) respectively. Therefore, a point $p1 : (x, y) = (12, 12)$$p1 : (x, y) = (12, 12)$ of image 1 is $(-8, 12, 1)$$(-8, 12, 1)$ and another point $p2 : (u, v)$$p2 : (u, v)$ of image 2 is $(21, 12, 1)$$(21, 12, 1)$ in world coordinates.

Besides, because of the correspondence of these two points, the real point in 3D location is the intersection of two lines $\vec{f_1}\vec{p_1}$$\vec{f_1}\vec{p_1}$ and $\vec{f_2}\vec{p_2}$$\vec{f_2}\vec{p_2}$.

$$\begin{cases} l_1 : (-20, 0, 0) + k_1[(-8, 12, 1) - (-20, 0, 0)] \\ l_2 : (20, 0, 0) + k_2[(21, 12, 1) - (20, 0, 0)] \end{cases}$$

$$\begin{cases} l_1 : (-20, 0, 0) + k_1[(-8, 12, 1) - (-20, 0, 0)] \\ l_2 : (20, 0, 0) + k_2[(21, 12, 1) - (20, 0, 0)] \end{cases}$$

Therefore, two lines will be intersected when $k_1 = k_2 = \frac{40}{11}$$k_1 = k_2 = \frac{40}{11}$

Therefore, the point in 3D location is $[\frac{260}{11}, \frac{480}{11}, \frac{40}{11}]$$[\frac{260}{11}, \frac{480}{11}, \frac{40}{11}]$.

(b)

First of all, we can draw the diagram as following. We need to calcualte the disparity d namely $P_2P_1'$$P_2P_1'$.

According to similrity rules, it has,

$$\frac{O_2A}{O_2B} = \frac{P_1'P_2}{P'P} = \frac{AP_2}{BP}$$
$$\frac{f}{Z} = \frac{d}{40} = \frac{-u}{20-x}$$

$$\frac{O_2A}{O_2B} = \frac{P_1'P_2}{P'P} = \frac{AP_2}{BP}$$
$$\frac{f}{Z} = \frac{d}{40} = \frac{-u}{20-x}$$

Using the first and third term, we can derive the denotation of x x wrt u u. Besides, the point is on the line $X + Z = 0, Y = 0$ $X + Z = 0, Y = 0$. Therefore, $Z = -X$ $Z = -X$.

$$\frac{f}{Z} = \frac{-u}{20-X}$$
$$\frac{f}{-X} = \frac{-u}{20-X}$$
$$u = \frac{20}{X} - 1$$
$$X = \frac{20}{u+1}$$

$$\frac{f}{Z} = \frac{-u}{20-X}$$
$$\frac{f}{-X} = \frac{-u}{20-X}$$
$$u = \frac{20}{X} - 1$$
$$X = \frac{20}{u+1}$$

Substituting u u into the first and second of the equation,

$$\frac{1}{-X} = \frac{d}{40}$$
$$d = -\frac{40}{X}$$
$$d = -\frac{40}{X}$$
$$d = -\frac{40}{\frac{20}{u+1}}$$
$$d = -2(u+1) = -2u - 2$$

$$\frac{1}{-X} = \frac{d}{40}$$

$$d = -\frac{40}{X}$$

$$d = -\frac{40}{X}$$

$$d = -\frac{40}{\frac{20}{u+1}}$$

$$d = -2(u+1) = -2u - 2$$

Therefore, the disparity $d$$d$ is $-2u - 2$$-2u - 2$.

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 $I_A I_A$ and $I_B I_B$ from identical calibrated cameras separated by a rigid transformation

$$ {}^B_A\mathrm{T} = \begin{bmatrix} R & t \\ 0^T & 1 \end{bmatrix} {}^B_A\mathrm{T} = \begin{bmatrix} R & t \\ 0^T & 1 \end{bmatrix} $$

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 $H_A e_A = H_B e_B = [1, 0, 0]^T H_A e_A = H_B e_B = [1, 0, 0]^T$. Consider the following special cases:

a) Pure horizontal translation $t = [tx, 0, 0]^T t = [tx, 0, 0]^T$, R = I

b) Pure translation orthogonal to the optical axis $t = [tx, ty, 0]^T t = [tx, ty, 0]^T$, R = I

c) Pure translation along the optical axis $t = [0, 0, tz]^T t = [0, 0, tz]^T$, R = I

d) Pure rotation $t = [0, 0, 0]^T t = [0, 0, 0]^T$, 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

- The epipoles e_A e_A and e_B e_B
- The equation of the epipolar line $l_B l_B$ in $I_B I_B$ corresponding to the point $[x_A, y_A, 1]^T [x_A, y_A, 1]^T$ in $I_A I_A$ (if one exists)
- A plausible solution to the rectifying transforms H_A H_A and H_B 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

Solution 2

case a)

(1) Since in this case $t = [tx, 0, 0]^T t = [tx, 0, 0]^T$ and R = I, therefore E matrix will be

$$E = [tx]R = t = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -tx \\ 0 & tx & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -tx \\ 0 & tx & 0 \end{bmatrix}$$

$$E = [tx]R = t = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -tx \\ 0 & tx & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -tx \\ 0 & tx & 0 \end{bmatrix}$$

Then epipoles e1 and e2 are (According to $Ee' = 0$$Ee' = 0$ and $E^T e = 0$$E^T e = 0$)

$$\begin{cases} Ee_B = 0 \\ E^T e_A = 0 \end{cases}$$

$$\begin{cases} Ee_B = 0 \\ E^T e_A = 0 \end{cases}$$

Therefore,

$$\begin{cases} e_A = [1, 0, 0]^T \\ e_B = [1, 0, 0]^T \end{cases}$$

$$\begin{cases} e_A = [1, 0, 0]^T \\ e_B = [1, 0, 0]^T \end{cases}$$

(2) According to the equation $b = E^T p$$b = E^T p$, we can find the epipolar line in $I_B$$I_B$ corresponding to the point $[x_A, y_A, 1]^T$$[x_A, y_A, 1]^T$ in $I_A$$I_A$, which is

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & tx \\ 0 & -tx & 0 \end{bmatrix} \times \begin{bmatrix} x_A \\ y_A \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ t_x \\ -t_x y_A \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & tx \\ 0 & -tx & 0 \end{bmatrix} \times \begin{bmatrix} x_A \\ y_A \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ t_x \\ -t_x y_A \end{bmatrix}$$

Therefore, the line $l_B$$l_B$ is

$$-t_x y - t_x y_A = 0 => y = y_A$$

$$-t_x y - t_x y_A = 0 => y = y_A$$

(3) In order to complete the rectifying, we need to meet the equation $H_A e_A = H_B e_B = [1, 0, 0]^T$ $H_A e_A = H_B e_B = [1, 0, 0]^T$ Therefore,

$$H_A = H_B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$H_A = H_B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

case b)

(1) Since in this case $t = [tx, ty, 0]^T t = [tx, ty, 0]^T$ and R = I, therefore E matrix will be

$$E = [tx]R = t = \begin{bmatrix} 0 & 0 & ty \\ 0 & 0 & -tx \\ -ty & tx & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & ty \\ 0 & 0 & -tx \\ -ty & tx & 0 \end{bmatrix}$$

$$E = [tx]R = t = \begin{bmatrix} 0 & 0 & ty \\ 0 & 0 & -tx \\ -ty & tx & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & ty \\ 0 & 0 & -tx \\ -ty & tx & 0 \end{bmatrix}$$

Then epipoles e1 and e2 are (According to $Ee' = 0Ee' = 0$ and $E^T e = 0E^T e = 0$)

$$\begin{cases} e_A = [\frac{tx}{\sqrt{tx^2 + ty^2}}, \frac{ty}{\sqrt{tx^2 + ty^2}}, 0]^T \\ e_B = [\frac{tx}{\sqrt{tx^2 + ty^2}}, \frac{ty}{\sqrt{tx^2 + ty^2}}, 0]^T \end{cases}$$

$$\begin{cases} e_A = [\frac{tx}{\sqrt{tx^2 + ty^2}}, \frac{ty}{\sqrt{tx^2 + ty^2}}, 0]^T \\ e_B = [\frac{tx}{\sqrt{tx^2 + ty^2}}, \frac{ty}{\sqrt{tx^2 + ty^2}}, 0]^T \end{cases}$$

(2) Similarly as above, we can find the epipolar line in $I_B I_B$ corresponding to the point $[x_A, y_A, 1]^T [x_A, y_A, 1]^T$ in $I_A I_A$, which is

$$\begin{bmatrix} 0 & 0 & -ty \\ 0 & 0 & tx \\ ty & -tx & 0 \end{bmatrix} \times \begin{bmatrix} x_A \\ y_A \\ 1 \end{bmatrix} = \begin{bmatrix} -ty \\ tx \\ tyx_A - txy_A \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & -ty \\ 0 & 0 & tx \\ ty & -tx & 0 \end{bmatrix} \times \begin{bmatrix} x_A \\ y_A \\ 1 \end{bmatrix} = \begin{bmatrix} -ty \\ tx \\ tyx_A - txy_A \end{bmatrix}$$

Therefore, the line $l_B l_B$ is

$$-tyx + txy + tyx_A - txy_A = 0$$

$$-tyx + txy + tyx_A - txy_A = 0$$

(3) In order to complete the rectifying, we need to meet the equation $H_A e_A = H_B e_B = [1,0,0]^T$ $H_A e_A = H_B e_B = [1,0,0]^T$ Therefore,

$$H_A = H_B = \begin{bmatrix} \frac{tx}{\sqrt{tx^2+ty^2}} & \frac{ty}{\sqrt{tx^2+ty^2}} & 0 \\ -\frac{ty}{\sqrt{tx^2+ty^2}} & \frac{tx}{\sqrt{tx^2+ty^2}} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$H_A = H_B = \begin{bmatrix} \frac{tx}{\sqrt{tx^2+ty^2}} & \frac{ty}{\sqrt{tx^2+ty^2}} & 0 \\ -\frac{ty}{\sqrt{tx^2+ty^2}} & \frac{tx}{\sqrt{tx^2+ty^2}} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

case c)

(1) Since in this case $t = [0,0,tz]^T$ $t = [0,0,tz]^T$ and R = I, therefore E matrix will be

$$E = [tx]R = t = \begin{bmatrix} 0 & -tz & 0 \\ tz & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -tz & 0 \\ tz & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$E = [tx]R = t = \begin{bmatrix} 0 & -tz & 0 \\ tz & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -tz & 0 \\ tz & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Then epipoles e1 and e2 are (According to $Ee' = 0$ $Ee' = 0$ and $E^T e = 0$ $E^T e = 0$)

$$\begin{cases} e_A = [0,0,1]^T \\ e_B = [0,0,1]^T \end{cases}$$

$$\begin{cases} e_A = [0,0,1]^T \\ e_B = [0,0,1]^T \end{cases}$$

(2) Similarly as above, we can find the epipolar line in $I_B I_B$ corresponding to the point $[x_A, y_A, 1]^T$ $[x_A, y_A, 1]^T$ in $I_A I_A$, which is

$$\begin{bmatrix} 0 & tz & 0 \\ -tz & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} x_A \\ y_A \\ 1 \end{bmatrix} = \begin{bmatrix} tzy_A \\ -tzx_A \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & tz & 0 \\ -tz & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} x_A \\ y_A \\ 1 \end{bmatrix} = \begin{bmatrix} tzy_A \\ -tzx_A \\ 0 \end{bmatrix}$$

Therefore, the line $l_B l_B$ is

$$tzy_A x - tzX_A y = 0 => y_A x - X_A y = 0$$

$$tzy_A x - tzX_A y = 0 => y_A x - X_A y = 0$$

(3) In order to complete the rectifying, we need to meet the equation $H_A e_A = H_B e_B = [1, 0, 0]^T$ $H_A e_A = H_B e_B = [1, 0, 0]^T$ Therefore,

$$H_A = H_B = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

$$H_A = H_B = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

case d)

(1) Since in this case $t = [0, 0, 0]^T$ $t = [0, 0, 0]^T$ and R = an arbitrary rotation matrix, therefore E matrix will be

$$E = [tx]R = t = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \times R = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$E = [tx]R = t = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \times R = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Then epipoles e1 and e2 are arbitrary unit vectors.

(2) Similarly as above, we can find the epipolar line in $I_B I_B$ corresponding to the point $[x_A, y_A, 1]^T [x_A, y_A, 1]^T$ in $I_A I_A$, which is

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} x_A \\ y_A \\ 1 \end{bmatrix} = [0, 0, 0]^T$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} x_A \\ y_A \\ 1 \end{bmatrix} = [0, 0, 0]^T$$

However, it is not a line. Therefore,there's no such line $l_B l_B$.

(3) There's no such $H_A H_A$ and $H_B H_B$ exist. Therefore, there's no plausible solution to the rectifying transforms.

Problem 3: Filtering [3 pts]

a) Consider smoothing an image with a 3x3 box filter and then computing the derivative in the x direction. What is a single convolution kernel that will implement this operation?

b) Give an example of a separable filter and compare the number of arithmetic operations it takes to convolve using that filter on an $n \times n n \times n$ image before and after separation.

Solution P3

(a)

Suppose we use the following smoothing kennel. Then the kernel will be

$$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

$$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

Further, we need to compute the derivative in x direction. In order to use convolution, the kernel will be

$$\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

If we want to combine the two process into single one, we can multiply the vector with the mean matrix, which is

$$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

where * is the convolve operation.

(b)

Suppose we denote seperable filter as matrix in $A \times A$$A \times A$ of dimension. Before "seperation", the arithmetic operations equals $nnA^2$$nnA^2$,namely $O(A^2n^2)$$O(A^2n^2)$. because there are nn pixels in the given image. And for each window, we need to do $A^2$$A^2$ times operations. Whereas after "seperation", there're two filters whose dimensions are $A \times 1$$A \times 1$ and $1 \times A$$1 \times A$. Therefore, the operations should be $nnA + nnB = nn(A + B)$$nnA + nnB = nn(A + B)$, namely $O(An^2)$$O(An^2)$.

Problem 4: Sparse Stereo Matching [22 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 set 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.

Corner Detection [5 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/fa18/cse252A-a/lec11.pdf (http://cseweb.ucsd.edu/classes/fa18/cse252A-a/lec11.pdf)](http://cseweb.ucsd.edu/classes/fa18/cse252A-a/lec11.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 in Figure 2. You may find scipy.ndimage.filters.gaussian_filter easy to use for smoothing. In this problem, try different parameters and then comment on results.

1. windowSize = 3, 5, 9, 17
2. smoothSTD = 0.5, 1, 2, 4 <img src = "fig/dinoCorner1.png" alt="dino Corner 1" width="400"/> <img src = "fig/dinoCorner2.png" alt="dino Corner 2" width="400"/>

In [6]:

```python
import numpy as np
from scipy.misc import imread
from scipy.signal import fftconvolve
import matplotlib.pyplot as plt
from scipy.ndimage.filters import gaussian_filter
```

In [7]:

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

In [11]:

```python
def window(I,centerx,centery,halfsize):
    xmin = centerx - halfsize
    xmax = centerx + halfsize + 1
    ymin = centery - halfsize
    ymax = centery + halfsize + 1


    (width,height) = I.shape
    if xmin < 0:
        xmin = 0
    if xmax > width:
        xmax = width
    if ymin < 0:
        ymin = 0
    if ymax > height:
        ymax = height
    return I[xmin:xmax,ymin:ymax]

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).

    """
    #smooth image
    image_smoothed = gaussian_filter(image,smoothSTD)
    (width,height) = image_smoothed.shape

    #compute gradient
    Ix, Iy = np.gradient(image_smoothed)
    Ix2 = fftconvolve(Ix*Ix,np.ones([windowSize,windowSize]))
    Iy2 = fftconvolve(Iy*Iy,np.ones([windowSize,windowSize]))
    Ixy = fftconvolve(Ix*Iy,np.ones([windowSize,windowSize]))

    lamb = np.zeros((width,height))
    for i in range(width):
        for j in range(height):
            C = np.array([[Ix2[i,j],Ixy[i,j]],[Ixy[i,j],Iy2[i,j]]])
            (V,D) = np.linalg.eig(C)
            lamb[i,j] = min(V)

    #nms
    i_red = int((windowSize-1)/2)
    corners = np.zeros((nCorners, 2))
    corner_and_value = []
    for i in range(i_red, image.shape[0]-i_red):
        for j in range(i_red, image.shape[1]-i_red):
            c_sub = window(lamb,i,j,i_red)
            if (lamb[i,j]==c_sub.max()):
                corner_and_value.append((i,j,lamb[i,j]))
    corner_and_value= sorted(corner_and_value, key=lambda x : x[2],reverse=True)

    top_n = corner_and_value[:nCorners]
    for index,item in enumerate(top_n):
        corners[index][0] = item[1]
        corners[index][1] = item[0]
    return corners
```

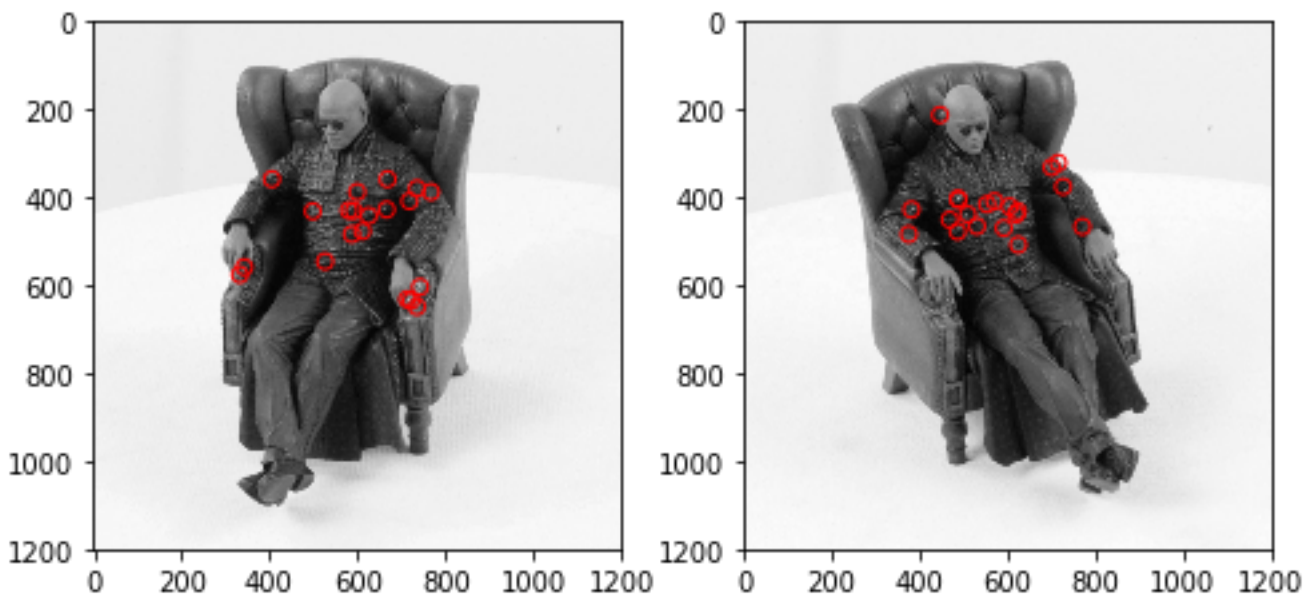
In [12]:

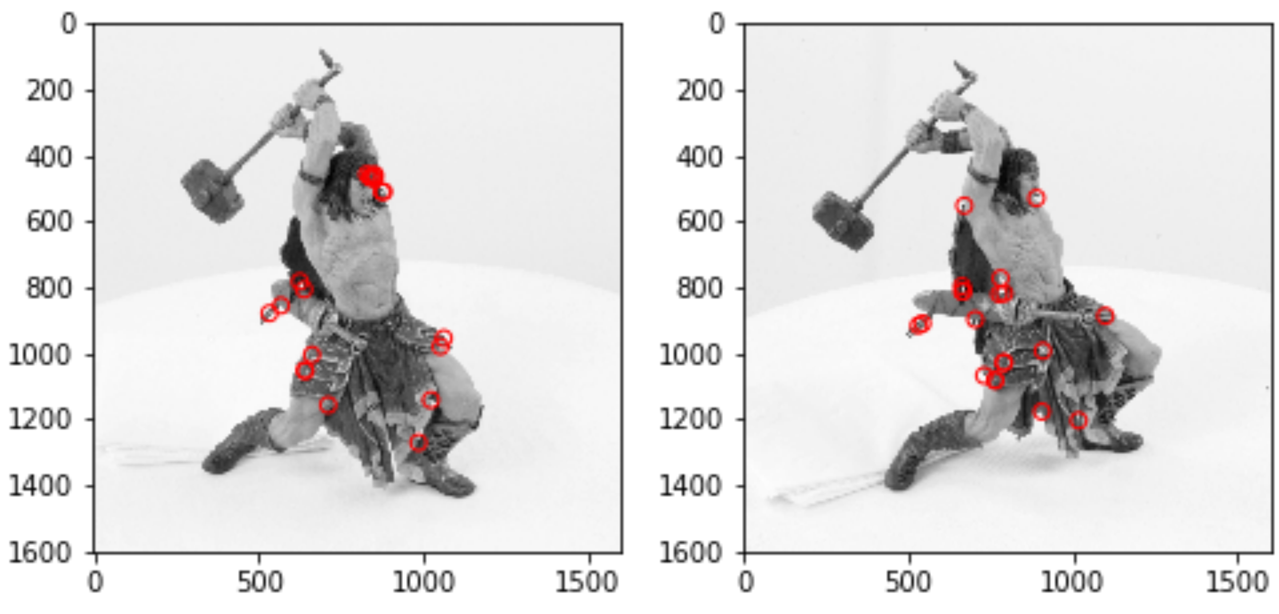
```
#case1
nCorners = 20
smoothSTD = 0.5
windowSize = 3
imgs_mat_0 = []
crns_mat_0 = []
imgs_war_0 = []
crns_war_0 = []
for i in range(2):
    img_mat = imread('p4/matrix/matrix' + str(i) + '.png')
    imgs_mat_0.append(rgb2gray(img_mat))
    # downsize your image in case corner_detect runs slow in test
    #imgs_mat.append(rgb2gray(img_mat)[::2, ::2])
    crns_mat_0.append(corner_detect(imgs_mat_0[i], nCorners, smoothSTD, windowSi
ze))

    img_war = imread('p4/warrior/warrior' + str(i) + '.png')
    imgs_war_0.append(rgb2gray(img_war))
    # downsize your image in case corner_detect runs slow in test
    #imgs_war.append(rgb2gray(img_war)[::2, ::2])
    crns_war_0.append(corner_detect(imgs_war_0[i], nCorners, smoothSTD, windowSi
ze))

#case2
nCorners = 20
smoothSTD = 1
windowSize = 5
imgs_mat_1 = []
crns_mat_1 = []
imgs_war_1 = []
crns_war_1 = []
for i in range(2):
    img_mat = imread('p4/matrix/matrix' + str(i) + '.png')
    imgs_mat_1.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_1.append(corner_detect(imgs_mat_1[i], nCorners, smoothSTD, windowSi
ze))

    img_war = imread('p4/warrior/warrior' + str(i) + '.png')
    imgs_war_1.append(rgb2gray(img_war))
    # downsize your image in case corner_detect runs slow in test
    #imgs_war.append(rgb2gray(img_war)[::2, ::2])
    crns_war_1.append(corner_detect(imgs_war_1[i], nCorners, smoothSTD, windowSi
ze))


#case3
nCorners = 20
smoothSTD = 2
windowSize = 9
imgs_mat_2 = []
crns_mat_2 = []
imgs_war_2 = []
crns_war_2 = []
for i in range(2):
    img_mat = imread('p4/matrix/matrix' + str(i) + '.png')
    imgs_mat_2.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_2.append(corner_detect(imgs_mat_2[i], nCorners, smoothSTD, windowSi
ze))

    img_war = imread('p4/warrior/warrior' + str(i) + '.png')
    imgs_war_2.append(rgb2gray(img_war))
    # downsize your image in case corner_detect runs slow in test
    #imgs_war.append(rgb2gray(img_war)[::2, ::2])
    crns_war_2.append(corner_detect(imgs_war_2[i], nCorners, smoothSTD, windowSi
ze))


#case4
nCorners = 20
smoothSTD = 4
windowSize = 17
imgs_mat_3 = []
crns_mat_3 = []
imgs_war_3 = []
crns_war_3 = []
for i in range(2):
    img_mat = imread('p4/matrix/matrix' + str(i) + '.png')
    imgs_mat_3.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_3.append(corner_detect(imgs_mat_3[i], nCorners, smoothSTD, windowSi
ze))

    img_war = imread('p4/warrior/warrior' + str(i) + '.png')
    imgs_war_3.append(rgb2gray(img_war))
    # downsize your image in case corner_detect runs slow in test
    #imgs_war.append(rgb2gray(img_war)[::2, ::2])
    crns_war_3.append(corner_detect(imgs_war_3[i], nCorners, smoothSTD, windowSi
ze))
```

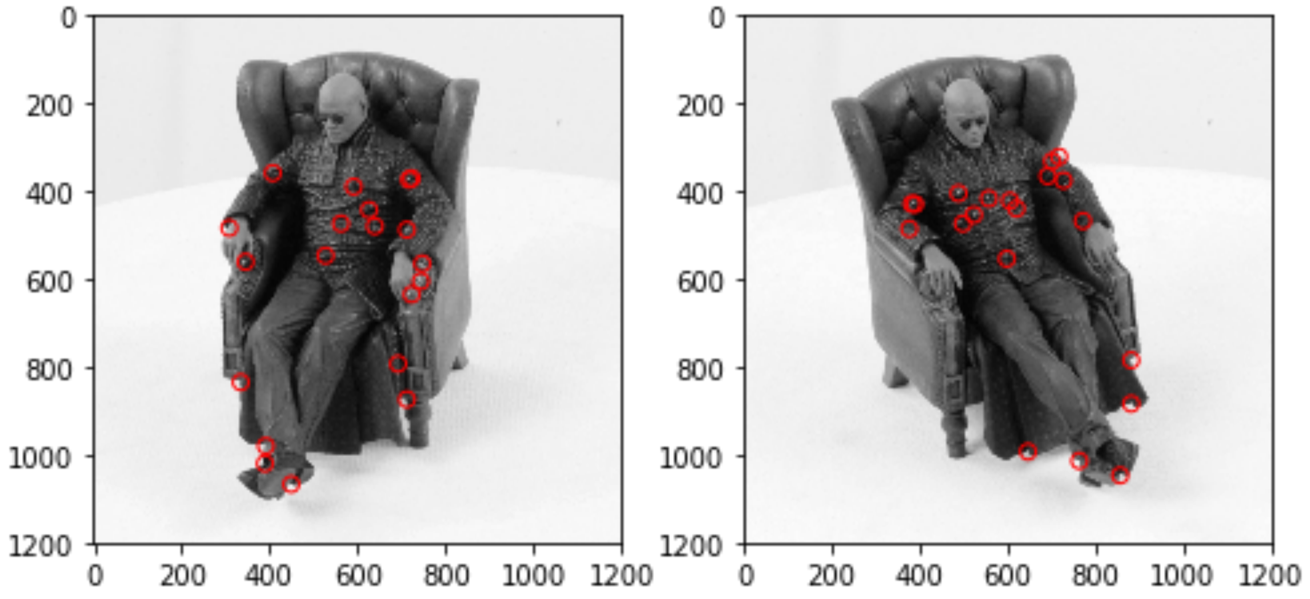
```
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:10: Dep
recationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  # Remove the CWD from sys.path while we load stuff.
/anaconda3/lib/python3.7/site-packages/mkl_fft/_numpy_fft.py:1044: F
utureWarning: Using a non-tuple sequence for multidimensional indexi
ng is deprecated; use `arr[tuple(seq)]` instead of `arr[seq]`. In th
e future this will be interpreted as an array index, `arr[np.array(s
eq)]`, which will result either in an error or a different result.
  output = mkl_fft.rfftn_numpy(a, s, axes)
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:16: Dep
recationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  app.launch_new_instance()
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:31: Dep
recationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:37: Dep
recationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:52: Dep
recationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:58: Dep
recationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:74: Dep
recationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:80: Dep
recationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
```

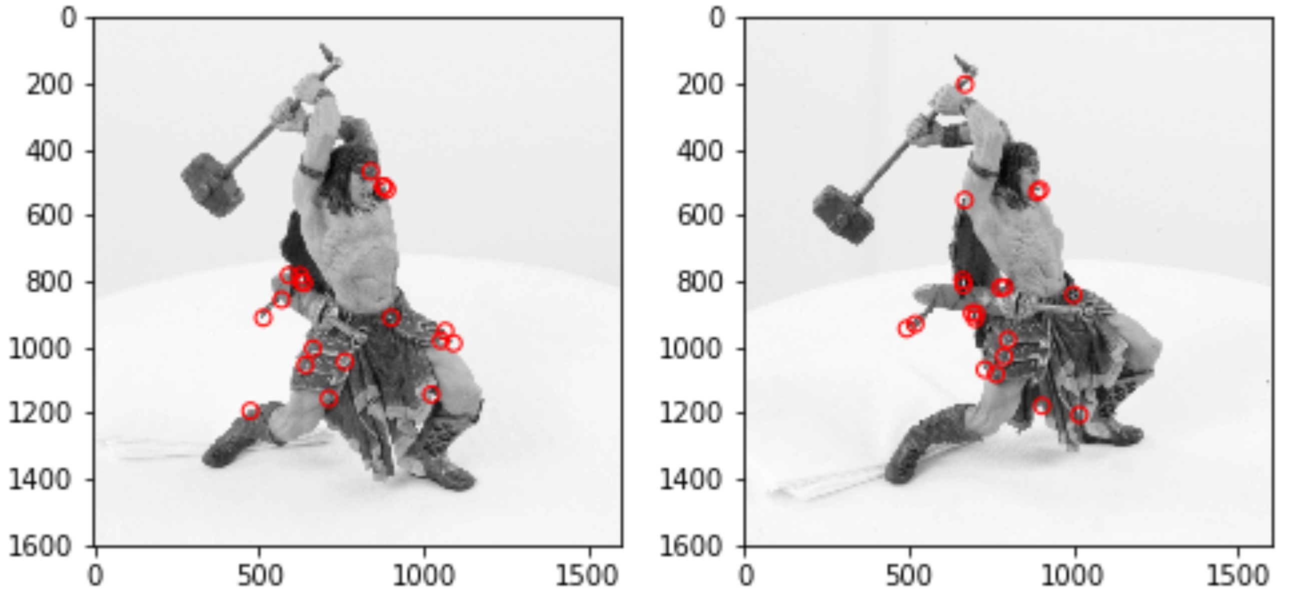
In [13]:

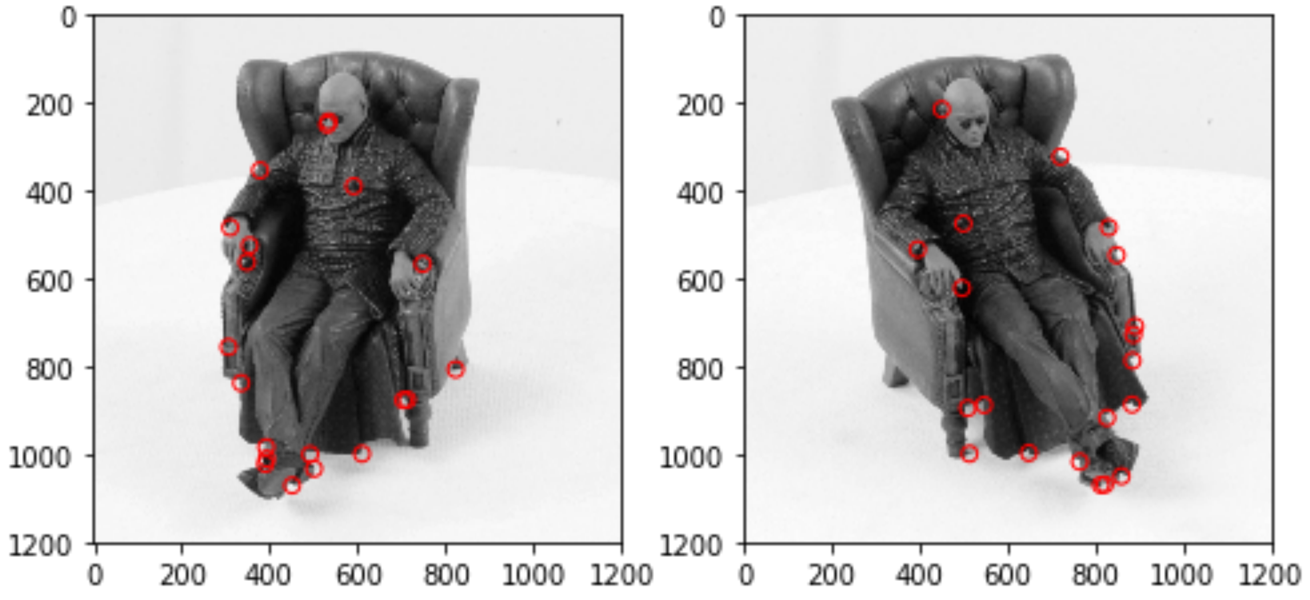
```python
def show_corners_result(imgs, corners):
    fig = plt.figure(figsize=(8, 8))
    ax1 = fig.add_subplot(221)
    ax1.imshow(imgs[0], cmap='gray')
    ax1.scatter(corners[0][:, 0], corners[0][:, 1], s=35, edgecolors='r', faceco
lors='none')

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

show_corners_result(imgs_mat_0, crns_mat_0)
show_corners_result(imgs_war_0, crns_war_0)
show_corners_result(imgs_mat_1, crns_mat_1)
show_corners_result(imgs_war_1, crns_war_1)
show_corners_result(imgs_mat_2, crns_mat_2)
show_corners_result(imgs_war_2, crns_war_2)
show_corners_result(imgs_mat_3, crns_mat_3)
show_corners_result(imgs_war_3, crns_war_3)
```

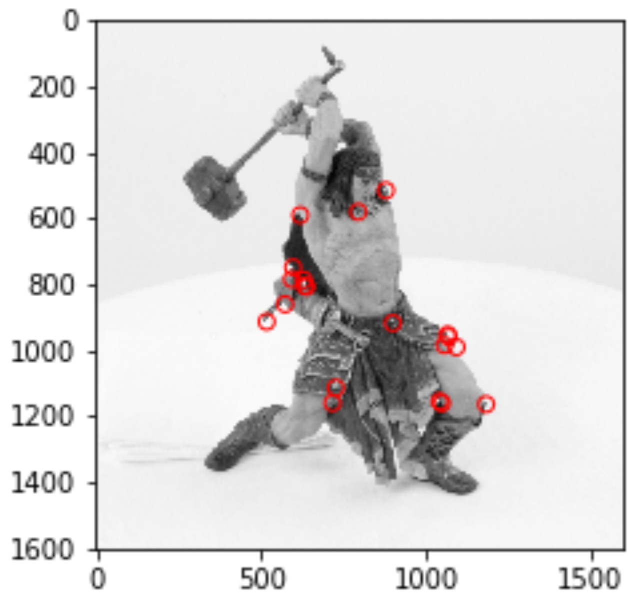


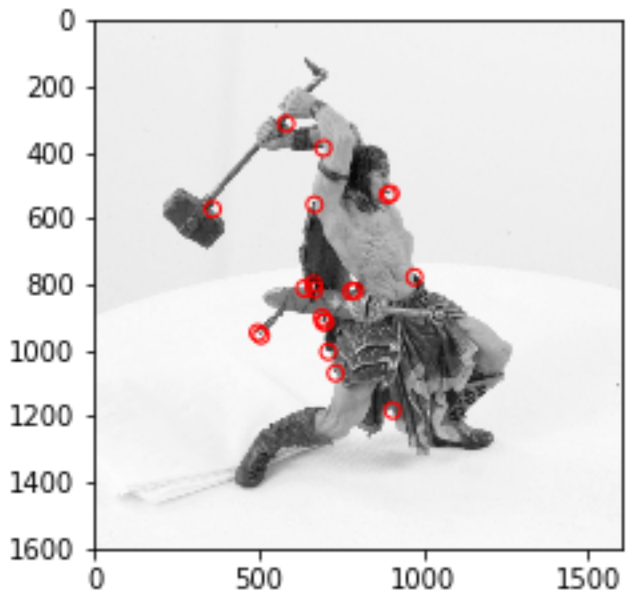


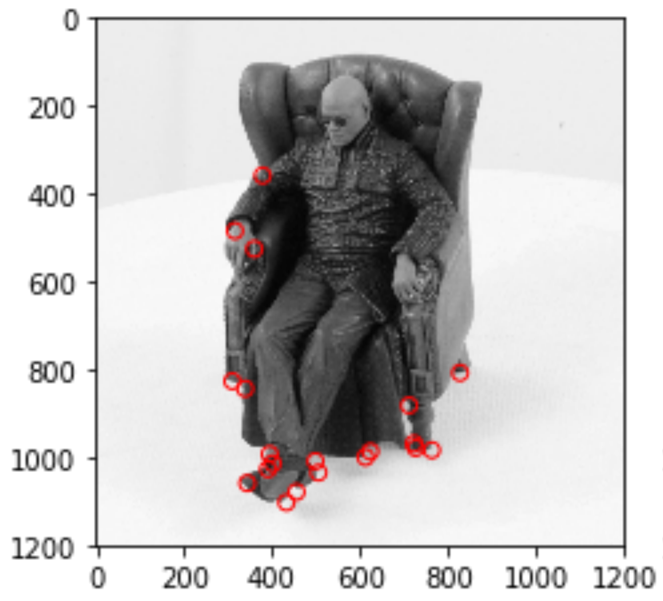

0
200
400
600
800
1000
1200

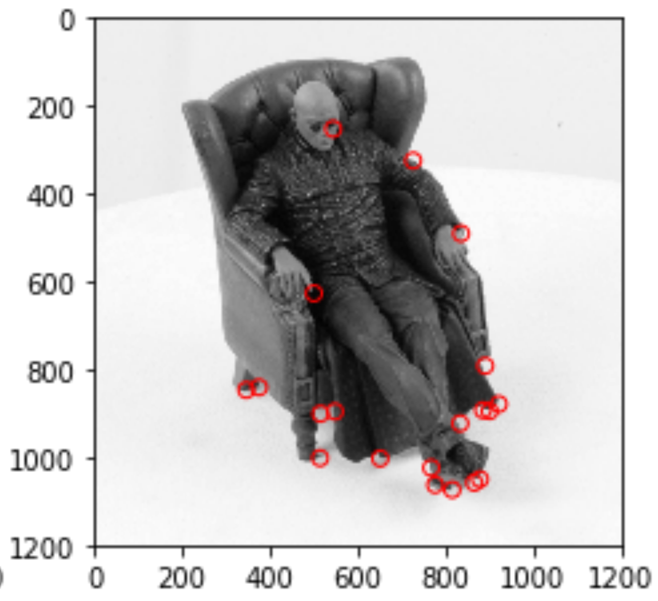

0
200
400
600
800
1000
1200
1400
1600
500
1500

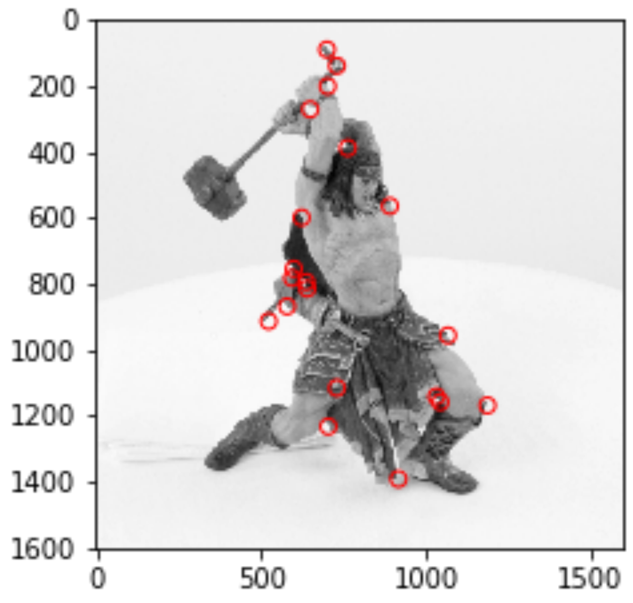

0
200
400
600
800
1000
1200

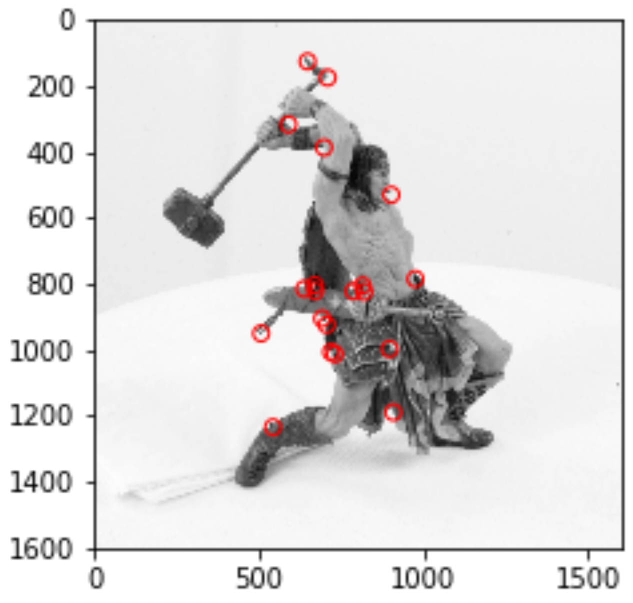












Comment

WindowSize: When the windowSize is small,the corners will be detected colosely but actually they are the same corner.

SmoothSTD: The gaussian filter is used to smooth the image, so applying it will blur the image and overlook some details of the image. When the smoothSTD is small, we may detect many corners that are not clear.

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

Write a function nccmatch that implements the NCC matching algorithm for two input windows. NCC = $\sum{i,j}\tilde{W_1} (i,j)\cdot \tilde{W2} (i,j)wherewhere\tilde{W} = \frac{W - \overline{W}}{\sqrt{\sum{k,l}(W(k,l) - \overline{W})^2}}isamean − shiftedandnormalizedversionofthewindowand isamean − shiftedandnormalizedversionofthewindowand\overline{W}$ is the mean pixel value in the window W.

In [14]:

```
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.

    """
    (width,height) = img1.shape
    matching_score = 0
    w1 = window(img1,c1[1],c1[0],R)
    w2 = window(img2,c2[1],c2[0],R)
    mean1 = w1.mean()
    mean2 = w2.mean()
    w1 = (w1-mean1)/np.linalg.norm(w1-mean1)
    w2 = (w2-mean2)/np.linalg.norm(w2-mean2)

    if (w1.shape!=w2.shape):
        matching_score = -1000000
    else:
        matching_score = (w1*w2).ravel().sum()
    return matching_score
```

In [15]:

```
# 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
```

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 naiveCorrespondanceMatching.m and call it as below. Examine your results for 10, 20, and 30 detected corners in each image. Choose a number of detected corners to the maximize the number of correct matching pairs. naive_matching will call your NCC mathching code. <img src = "fig/dinoMatch.png" alt="dino match" width="800"/>

In [16]:

```
def naive_matching(img1, img2, corners1, corners2, R, SSDth):
    """Compute SSD given two windows.

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

    Returns:
        SSD 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.

    """
    matching = []
    for cor1 in range(corners1.shape[0]):
        score_max = -1000
        location_max = None
        c1 = corners1[cor1]
        for cor2 in range(corners2.shape[0]):
            c2 = corners2[cor2]
            score = ncc_match(img1, img2, c1, c2, R)
            if (score>score_max) and (score>SSDth):
                score_max = score
                location_max = c2
        if (location_max is not None):
            matching.append((corners1[cor1],location_max))
    return matching
```

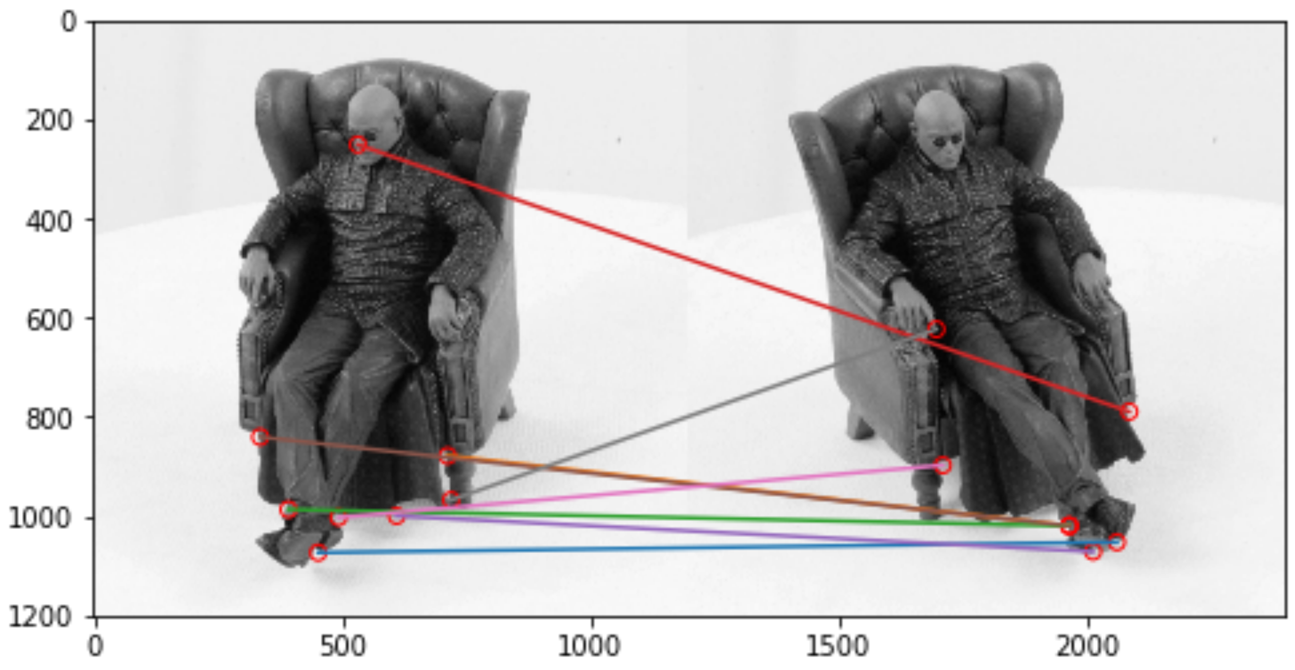
In [17]:

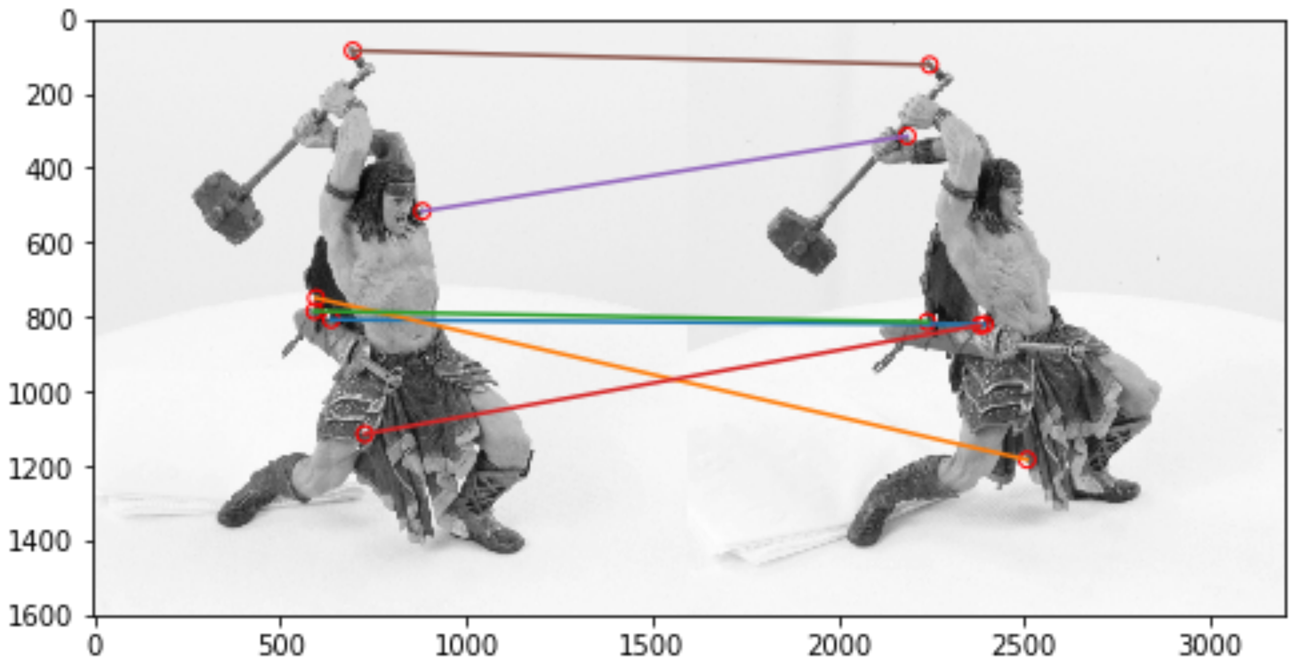
```
# detect corners on warrior and matrix sets
# adjust your corner detection parameters here
nCorners = 20
smoothSTD = 2
windowSize = 11

# read images and detect corners on images
imgs_mat = []
crns_mat = []
imgs_war = []
crns_war = []
for i in range(2):
    img_mat = 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 = 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))
```

```
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:13: Dep
recationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  del sys.path[0]
/anaconda3/lib/python3.7/site-packages/mkl_fft/_numpy_fft.py:1044: F
utureWarning: Using a non-tuple sequence for multidimensional indexi
ng is deprecated; use `arr[tuple(seq)]` instead of `arr[seq]`. In th
e future this will be interpreted as an array index, `arr[np.array(s
eq)]`, which will result either in an error or a different result.
  output = mkl_fft.rfftn_numpy(a, s, axes)
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:19: Dep
recationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
```

In [37]:

```
# match corners
R = 15
NCCth = 0.75
matching_mat = naive_matching(imgs_mat[0]/255, imgs_mat[1]/255, crns_mat[0].asty
pe(np.int),
                              crns_mat[1].astype(np.int), R, NCCth)
matching_war = naive_matching(imgs_war[0]/255, imgs_war[1]/255, crns_war[0].asty
pe(np.int),
                              crns_war[1].astype(np.int),R, NCCth)
# 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 di
fferent 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', facecolo
rs='none')
        plt.plot([p1[0], p2[0] + img1.shape[1]], [p1[1], p2[1]])
    plt.savefig('dino_matching.png')
    plt.show()

show_matching_result(imgs_mat[0], imgs_mat[1], matching_mat)
show_matching_result(imgs_war[0], imgs_war[1], matching_war)
```





Epipolar Geometry [4 pts]

Using the fundamental_matrix function, and the corresponding points provided in cor1.npy and cor2.npy, calculate the fundamental matrix.

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. Shown your result for matrix and warrior as the figure below. <img src = "fig/dinoEpi1.png" alt="dino epipolar" width="400"/> <img src = "fig/dinoEpi2.png" alt="dino epipolar" width="400"/>

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

In [38]:

```
import numpy as np
from scipy.misc import imread
import matplotlib.pyplot as plt
from scipy.io import loadmat
```

```python
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]
    """

    n = x1.shape[1]
    if x2.shape[1] != n:
        raise ValueError("Number of points don't match.")

    # build matrix for equations
    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] ]

    # compute linear least square solution
    U,S,V = np.linalg.svd(A)
    F = V[-1].reshape(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]


def fundamental_matrix(x1,x2):
    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]
def compute_epipole(F):
    '''
    This function computes the epipoles for a given fundamental matrix and corne
r point correspondences
    input:
    F--> Fundamental matrix
    output:
    e1--> corresponding epipole in image 1
    e2--> epipole in image2
    '''
    # Fe' = 0 and F.T e = 0
    (val2,vec2) = np.linalg.eig(F)
    (val1,vec1) = np.linalg.eig(F.T)

    min_val = val2[0]
    min_index = 0
    for index in range(1,len(val2)):
        if(val2[index]<min_val):
            min_val = val2[index]
            min_index = index
    e2 = vec2[:,min_index]

    min_val = val1[0]
    min_index = 0
    for index in range(1,len(val1)):
        if(val1[index]<min_val):
            min_val = val1[index]
            min_index = index
    e1 = vec1[:,min_index]
    return e1,e2
```

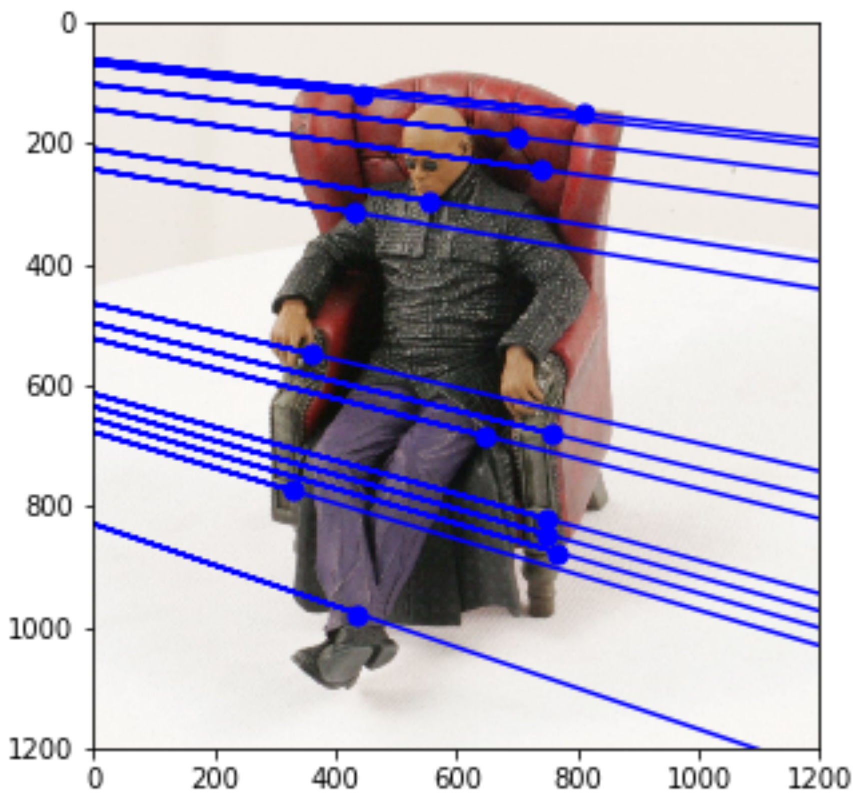
In [39]:

```
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)
    """
    F = fundamental_matrix(cor1,cor2)
    e1,e2 = compute_epipole(F)
    ex1 = e1[0]/e1[2]
    ey1 = e1[1]/e1[2]

    ex2 = e2[0]/e2[2]
    ey2 = e2[1]/e2[2]

    #image1
    fig = plt.figure(figsize=(5,5))
    plt.imshow(img1)
    for index in range(cor1.shape[1]):
        x = cor1[0,index]/cor1[2,index]
        y = cor1[1,index]/cor1[2,index]
        plt.scatter(x,y,color='b',s=50,edgecolors='b')
        plt.plot([0,x,ex1,img1.shape[0]-1],[ey1-(ey1-y)/(ex1-x)*ex1,y,ey1,(ey1-y
)/(ex1-x)*(img1.shape[0]-1-ex1)+ey1],
                 color='b')
    plt.xlim((0, img1.shape[1]))
    plt.ylim((img1.shape[0],0))
    plt.show()

    #image2
    plt.figure()
    fig = plt.figure(figsize=(5, 5))
    plt.imshow(img2)
    for index in range(cor2.shape[1]):
        x = cor2[0,index]/cor2[2,index]
        y = cor2[1,index]/cor2[2,index]
        plt.scatter(x,y,color='b',s=50,edgecolors='b')
        plt.plot([0,x,ex2,img2.shape[0]-1],[ey2-(ey2-y)/(ex2-x)*ex2,y,ey2,(ey2-y
)/(ex2-x)*(img2.shape[0]-1-ex2)+ey2],
                 color='b')
    plt.xlim((0, img2.shape[1]))
    plt.ylim((img2.shape[0],0))
    plt.show()
```

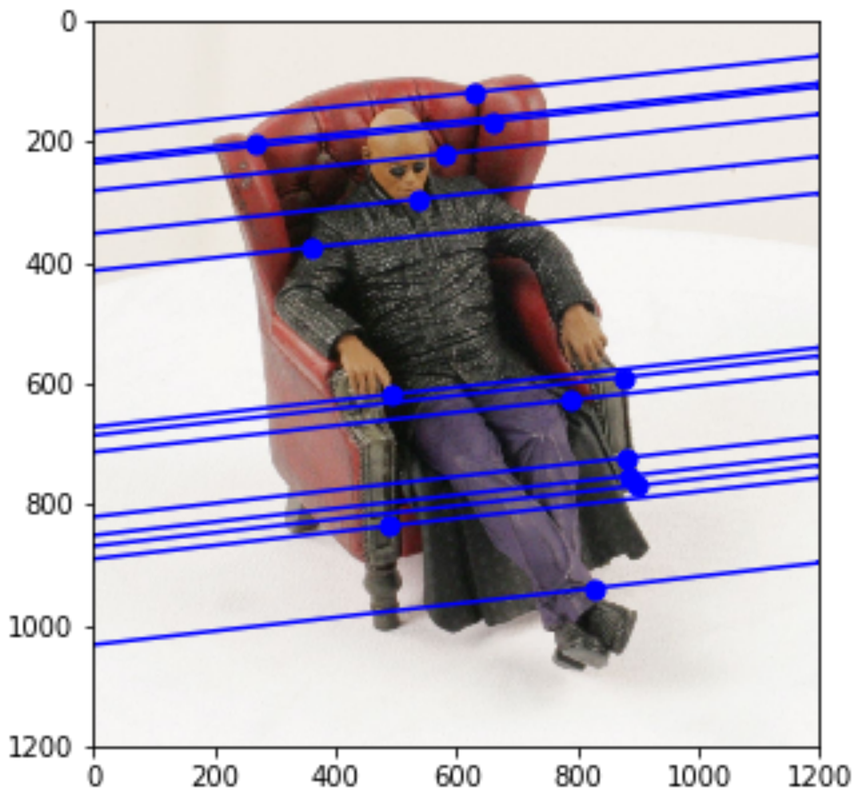
In [40]:

```
# replace images and corners with those of matrix and warrior

#matrix
I1 = imread("./p4/matrix/matrix0.png")
I2 = imread("./p4/matrix/matrix1.png")

cor1 = np.load("./p4/matrix/cor1.npy")
cor2 = np.load("./p4/matrix/cor2.npy")

plot_epipolar_lines(I1,I2,cor1,cor2)
#warrior
I1_w = imread("./p4/warrior/warrior0.png")
I2_w = imread("./p4/warrior/warrior1.png")

cor1_w = np.load("./p4/warrior/cor1.npy")
cor2_w = np.load("./p4/warrior/cor2.npy")

plot_epipolar_lines(I1_w,I2_w,cor1_w,cor2_w)
```

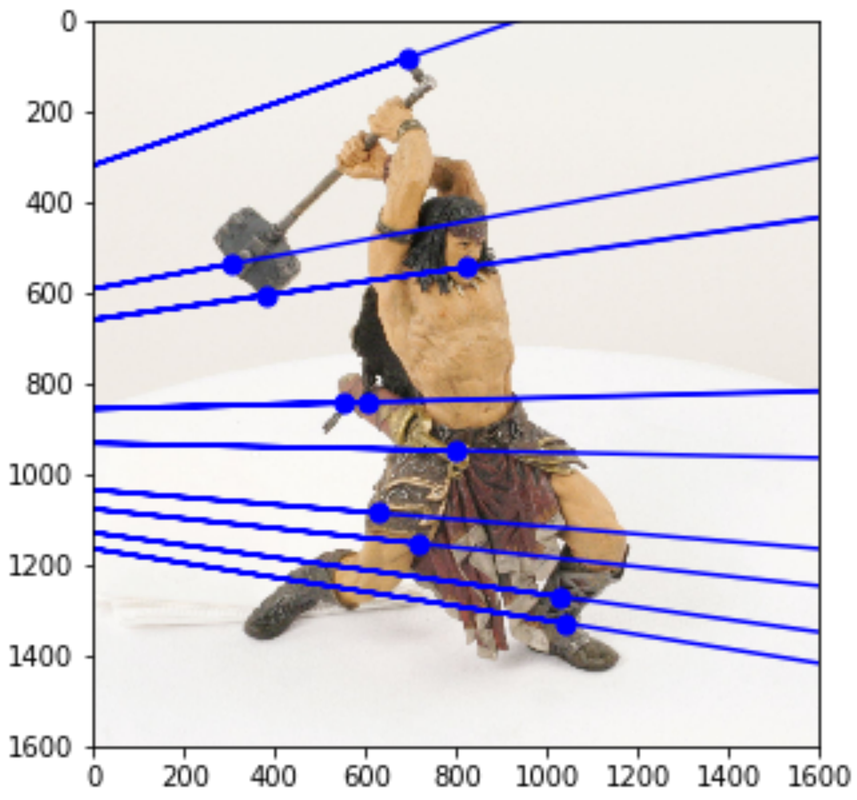
```
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:4: Depr
ecationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  after removing the cwd from sys.path.
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:5: Depr
ecationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  """
```



```
<Figure size 432x288 with 0 Axes>
```



```
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12: Dep
recationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  if sys.path[0] == '':
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:13: Dep
recationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  del sys.path[0]
```



```
<Figure size 432x288 with 0 Axes>
```

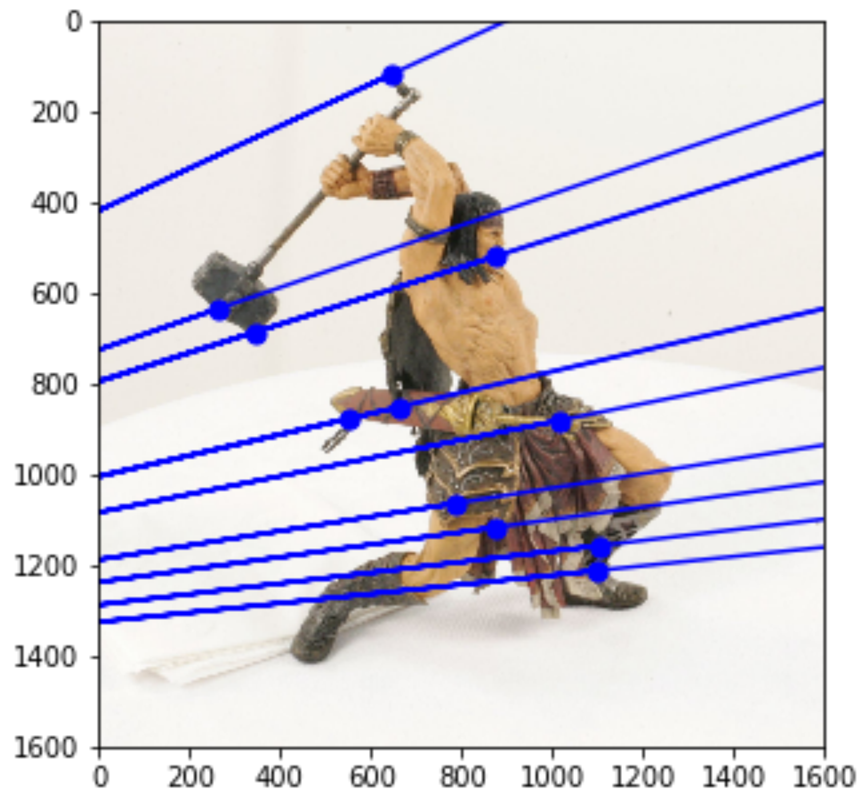


Image Rectification [3 pts]

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 $\mathrm{E} = [\boldsymbol{T_x}]\boldsymbol{R} = [\boldsymbol{T_x}]$. Also if you observe the epipolar lines $\boldsymbol{l}$ and $\boldsymbol{l}'$ 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 $\boldsymbol{l_i}$ and $\boldsymbol{l_i'}$ for each of the correspondances. The intersection of these lines will give us the respective epipoles $\boldsymbol{e}$ and $\boldsymbol{e}'$. 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.

Image Rectification Setup

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:

sample rectification

In [49]:

```
import cv2
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
    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 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
    outpu:
    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)
    e1 = e1/e1[2]
    e2 = e2/e2[2]
    H1,H2 = compute_matching_homographies(e2, F.T, im2, points1.T, points2.T)

    rectified_im1 = np.zeros(im1.shape)
    inverseH1 = np.linalg.inv(H1)
    for i in range(im1.shape[1]):
        for j in range(im1.shape[0]):
            newPosition = np.matmul(inverseH1,np.array([i,j,1]).T)
            x,y = int(newPosition[0]/newPosition[2]),int(newPosition[1]/newPosit
ion[2])
            if x>=0 and x<im1.shape[1] and y>=0 and y<im1.shape[0]:
                rectified_im1[j,i,:] = im1[y,x,:]


    rectified_im2 = np.zeros(im2.shape)
    inverseH2 = np.linalg.inv(H2)
    for i in range(im2.shape[1]):
        for j in range(im2.shape[0]):
            newPosition = np.matmul(inverseH2,np.array([i,j,1]).T)
            y,x = int(newPosition[0]/newPosition[2]),int(newPosition[1]/newPosit
ion[2])
            if x>=0 and x<im2.shape[0] and y>=0 and y<im2.shape[1]:
                rectified_im2[j,i,:] = im2[x,y,:]
    new_cor1 = np.matmul(H1,points1)
    new_cor2 = np.matmul(H2,points2)
    #print("e1:{},e2:{}".format(e1,e2))
    #print("H1:{},H2:{}".format(H1,H2))
    rectified_im1 = (rectified_im1-rectified_im1.min())/(rectified_im1.max()-rec
tified_im1.min())
    rectified_im2 = (rectified_im2-rectified_im2.min())/(rectified_im2.max()-rec
tified_im2.min())

    """In the following, we need to compensate the black bround into white."""

    for i in range(rectified_im1.shape[1]):
        for j in range(rectified_im1.shape[0]):
            if np.all(rectified_im1[j,i,:] == 0):
                rectified_im1[j,i,:] = [1,1,1]

    for i in range(rectified_im2.shape[1]):
        for j in range(rectified_im2.shape[0]):
            if np.all(rectified_im2[j,i,:] == 0):
                rectified_im2[j,i,:] = [1,1,1]
    return rectified_im1,rectified_im2,new_cor1,new_cor2
```

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.

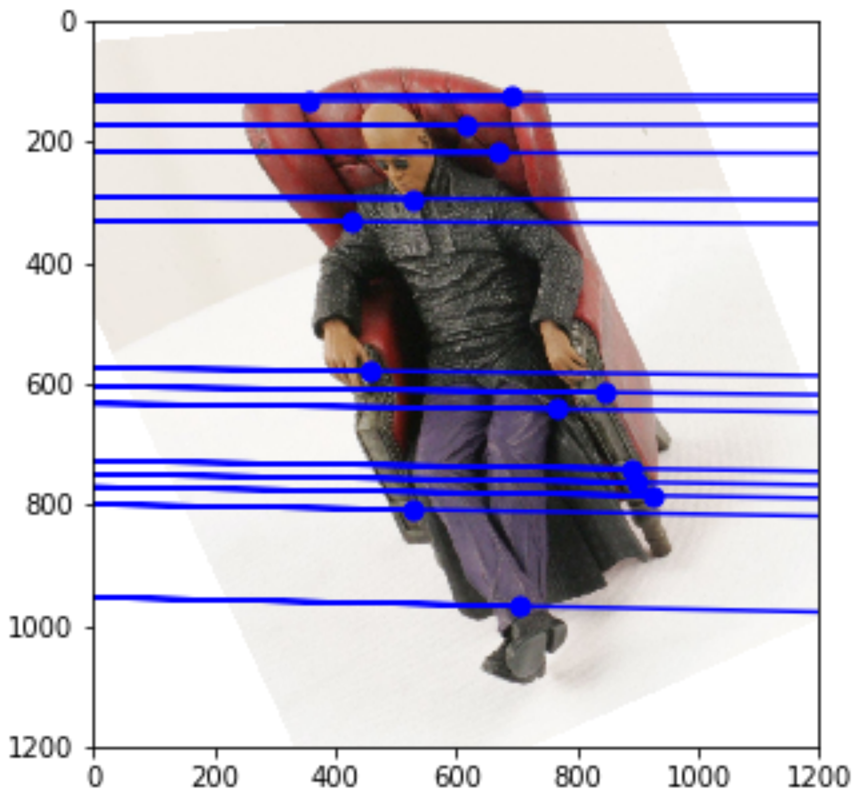
In [67]:

```
def display_correspondence(img1, img2, corrs):
    """Plot matching result on image pair given images and correspondences"""
    fig = plt.figure(figsize=(8, 8))
    plt.imshow(np.hstack((img1, img2))) # two dino images are of different sizes
, resize one before use
    for p1, p2 in corrs:
        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', facecolo
rs='none')
        plt.plot([p1[0], p2[0] + img1.shape[1]], [p1[1], p2[1]])
    plt.show()

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 corresp
ondences.
        R: SSD matching window radius.
        NCCth: NCC matching threshold.
    Returns:
        Matching result to be used in display_correspondence function
    """
    epi_lines = np.zeros((1,3))
    for crn_index in range(corners1.shape[0]):
        cor_homo = np.array([corners1[crn_index,0],corners1[crn_index,1],1])
        epi_line = np.matmul(F,cor_homo.T)
        epi_lines = np.vstack((epi_lines,epi_line))
        #A * x + B * y + C = 0
    lines = np.zeros((corners1.shape[0],1))
    for index in range(1,epi_lines.shape[0]):
        lines[index-1] = -epi_lines[index,2]/epi_lines[index,1]

    matching = []
    img_1gray = rgb2gray(img1)
    img_2gray = rgb2gray(img2)

    for index in range(len(lines)):
        y = int(lines[index,0])
        c1 = corners1[index,:].astype(np.int64)
        c2_match = None
        max_score = -1000
        for x in range(R+1,img_2gray.shape[1]-R):
            c2 = [x,y]
            score = ncc_match(img_1gray,img_2gray, c1, c2, R)
            if score>max_score and score > NCCth:
                max_score = score
                c2_match = c2
        if (c2_match is not None) and (max_score>NCCth):
            #print("max_score:",max_score)
            matching.append((c1,c2_match))
    return matching
```

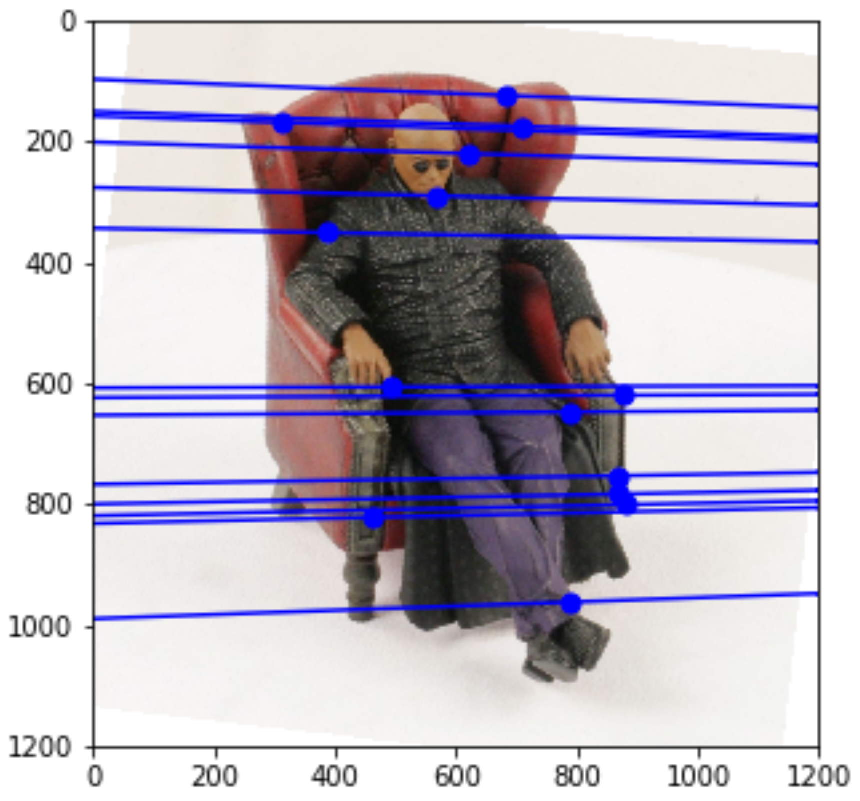
In [63]:

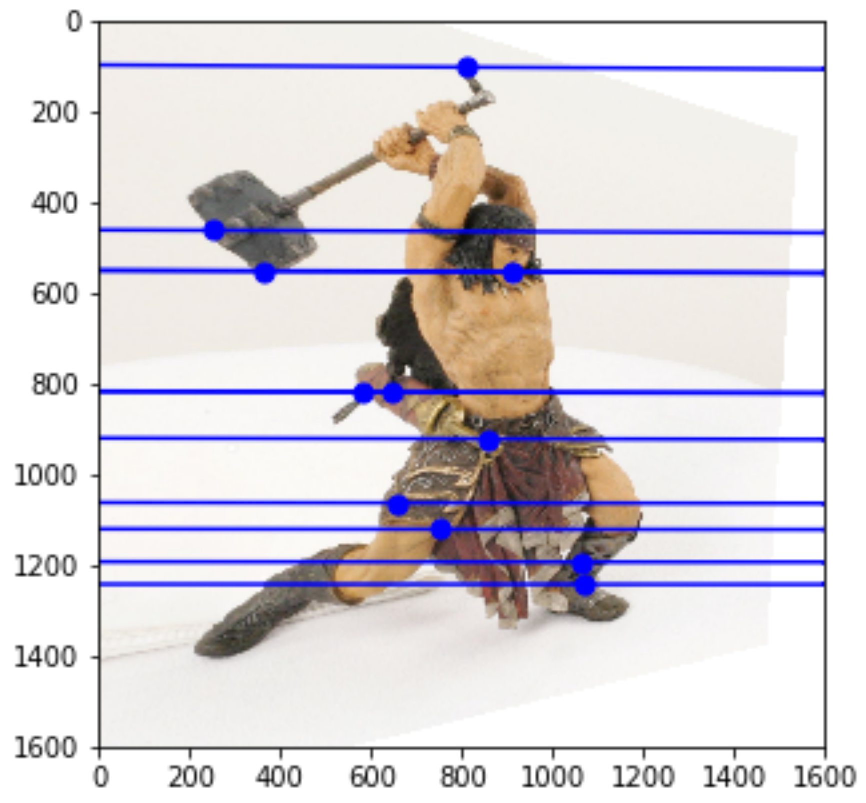
```
I1=imread("./p4/matrix/matrix0.png")
I2=imread("./p4/matrix/matrix1.png")
cor1 = np.load("./p4/matrix/cor1.npy")
cor2 = np.load("./p4/matrix/cor2.npy")
I3=imread("./p4/warrior/warrior0.png")
I4=imread("./p4/warrior/warrior1.png")
cor3 = np.load("./p4/warrior/cor1.npy")
cor4 = np.load("./p4/warrior/cor2.npy")
rectified_im1,rectified_im2,new_cor1,new_cor2 = image_rectification(I1,I2,cor1,c
or2)
rectified_im3,rectified_im4,new_cor3,new_cor4 = image_rectification(I3,I4,cor3,c
or4)
plot_epipolar_lines(rectified_im1,rectified_im2,new_cor1, new_cor2)
plot_epipolar_lines(rectified_im3,rectified_im4,new_cor3, new_cor4)
```

```
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:1: Depr
ecationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  """Entry point for launching an IPython kernel.
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:2: Depr
ecationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.

/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:5: Depr
ecationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  """
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:6: Depr
ecationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.

/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:62: Fut
ureWarning: `rcond` parameter will change to the default of machine
precision times ``max(M, N)`` where M and N are the input matrix dim
ensions.
To use the future default and silence this warning we advise to pass
`rcond=None`, to keep using the old, explicitly pass `rcond=-1`.
```


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

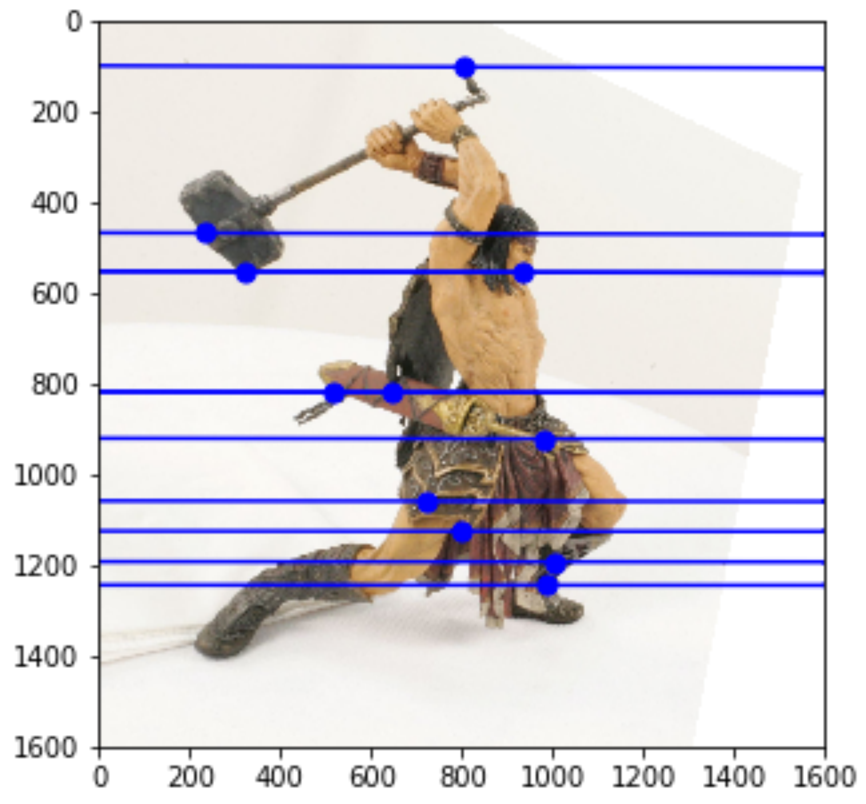
```
<Figure size 432x288 with 0 Axes>
```


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



0
200
400
600
800
1000
1200
1400
1600
0
200
400
600
800
1000
1200
1400
1600

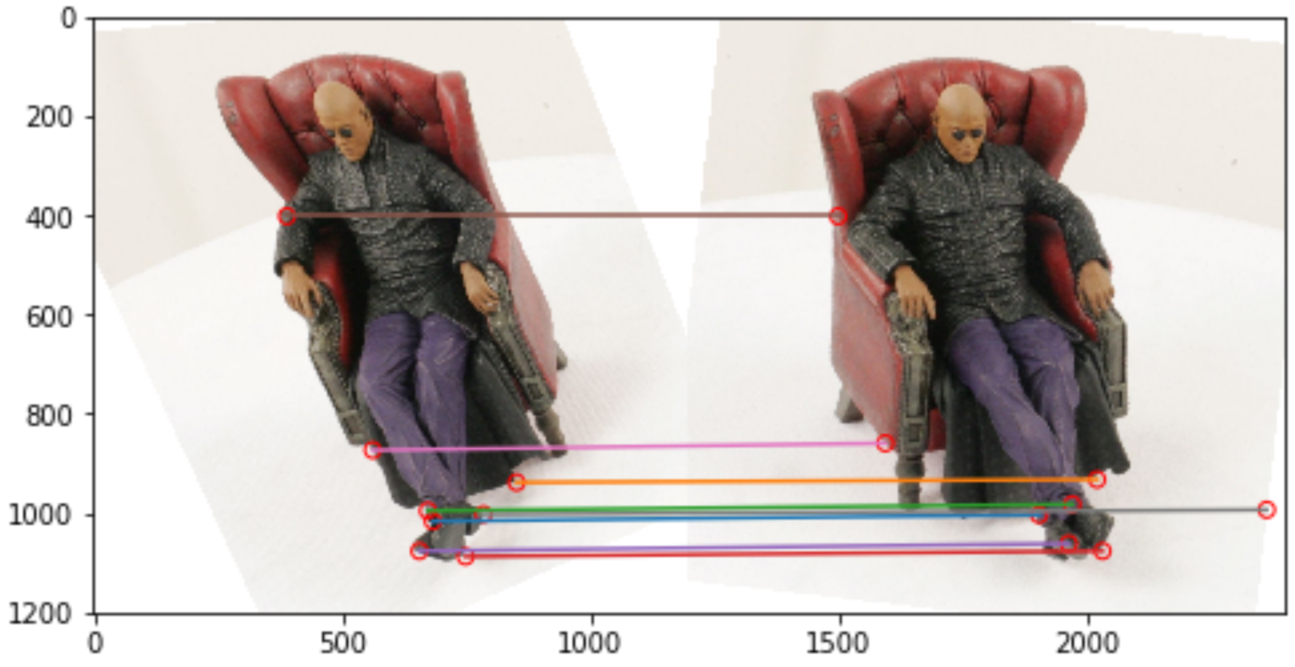
```
<Figure size 432x288 with 0 Axes>
```

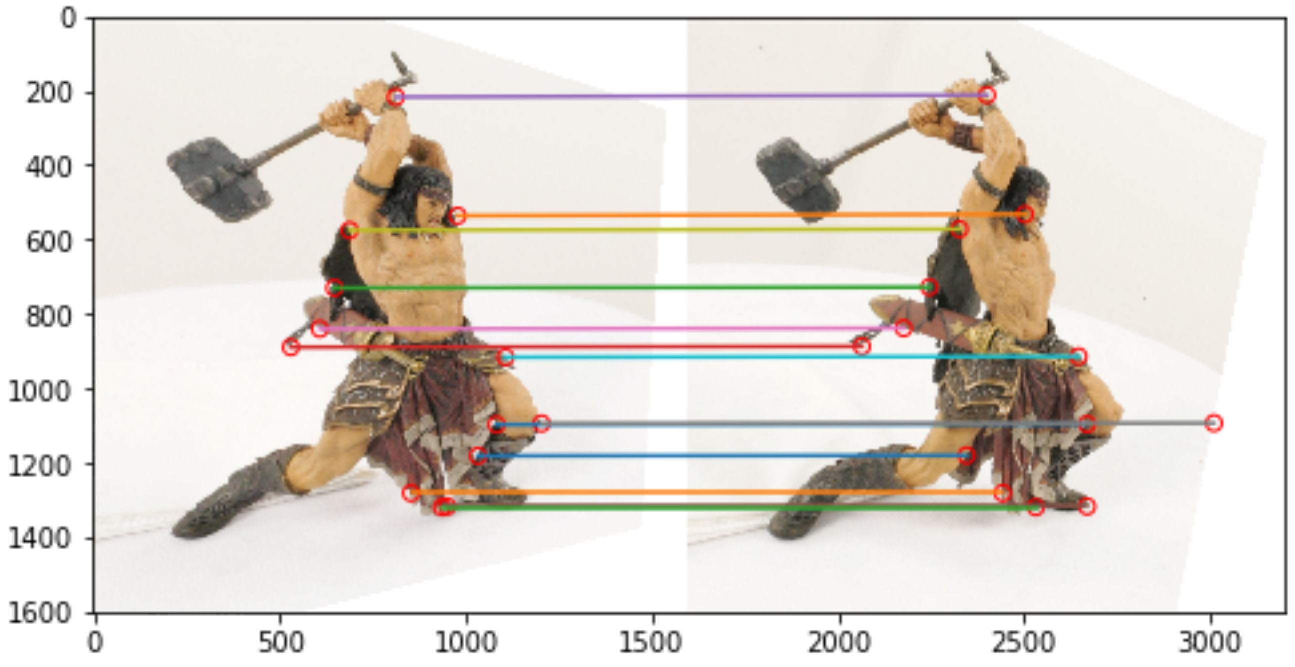


0
200
400
600
800
1000
1200
1400
1600
0
200
400
600
800
1000
1200
1400
1600

In [68]:

```
import warnings
warnings.filterwarnings('ignore')
F_new = fundamental_matrix(new_cor1, new_cor2)
nCorners = 35
R = 15
# detect corners using corner detector here, store in corners1
smoothSTD = 1
windowSize = 15
NCCth = 0.8
corners1 = corner_detect(rgb2gray(rectified_im1), nCorners, smoothSTD, windowSiz
e)
corrs = correspondence_matching_epipole(rectified_im1, rectified_im2, corners1,
F_new, R, NCCth)
display_correspondence(rectified_im1, rectified_im2, corrs)

F_new2=fundamental_matrix(new_cor3, new_cor4)
corners2 = corner_detect(rgb2gray(rectified_im3), nCorners, smoothSTD, windowSiz
e)
corrs = correspondence_matching_epipole(rectified_im3, rectified_im4, corners2,
F_new2, R, NCCth)
display_correspondence(rectified_im3, rectified_im4, corrs)
```


0
200
400
600
800
1000
1200
0
500
1000
1500
2000


0
200
400
600
800
1000
1200
1400
1600
0
500
1000
1500
2000
2500
3000