Assignment3

November 20, 2018

1 CSE 252A Computer Vision I Fall 2018 - Assignment 3

1.0.1 Instructor: David Kriegman

1.0.2 Assignment Published On: Wednesday, November 7, 2018

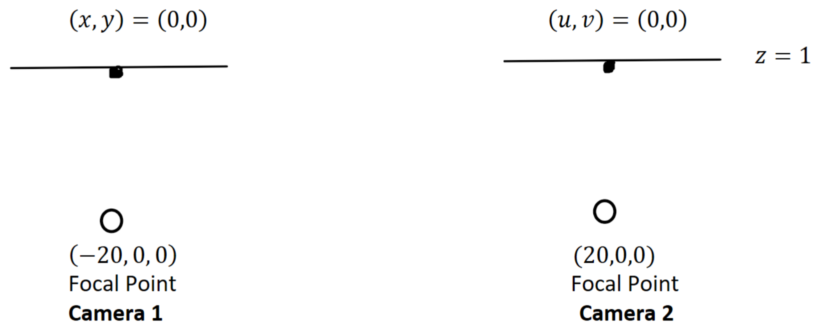
1.0.3 Due On: Tuesday, November 20, 2018 11:59 pm

1.1 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.

1.2 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). 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.

1.3 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

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

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]ˆT\$. 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

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 * The epipoles e_A and e_B * 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) * 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.

1.4 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?

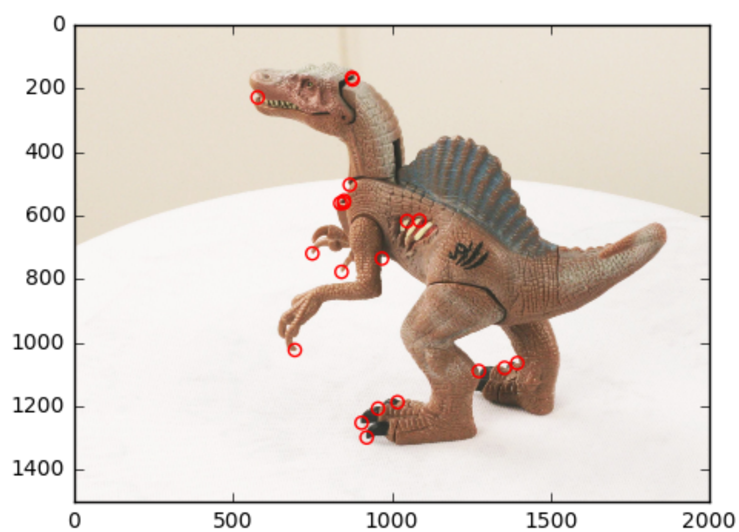
b) Give an example of a separable filter and compare the number of arithmetic operations it takes to convolve using that filter on an $n \times n$ image before and after separation.

1.5 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.

1.5.1 4.1 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. 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

Homework#3 Due Nov/20/2018

problem#1

(a) ignore z in camera_P_object

$$\begin{cases} \frac{{}^{c_1}P_{obj}}{{}_{c_1}z} = \frac{(x,y)}{1} \longrightarrow \frac{({}_{c_1}x_{obj}, {}_{c_1}y_{obj})}{{}_{c_1}z} = \frac{(12,12)}{1} \\ \frac{{}^{c_2}P_{obj}}{{}_{c_2}z} = \frac{(u,v)}{1} \longrightarrow \frac{({}_{c_2}u_{obj}, {}_{c_2}v_{obj})}{{}_{c_2}z} = \frac{(1,12)}{1} \end{cases}$$

$\because$

$${}^{w}P_{obj} = {}^{w}P_{c_1} + {}^{c_1}P_{obj} = {}^{w}P_{c_2} + {}^{c_2}P_{obj}$$

$$\Rightarrow {}^{c_1}P_{obj} = {}^{w}P_{obj} - {}^{w}P_{c_1} = {}^{w}P_{obj} - (-20,0)$$

$$\Rightarrow {}^{c_2}P_{obj} = {}^{w}P_{obj} - {}^{w}P_{c_1} = {}^{w}P_{obj} - (20,0)$$

Also ${}_{c_1}z = {}_{c_2}z$

$\therefore$

$$\frac{({}_{w}x_{obj}+20, {}_{w}y_{obj})}{z} = \frac{(12,12)}{1}$$

$$\frac{({}_{w}x_{obj}-20, {}_{w}y_{obj})}{z} = \frac{(1,12)}{1}$$

$$\begin{cases} {}_{w}x_{obj}+20 = 12z \\ {}_{w}x_{obj}-20 = z \end{cases} \longrightarrow \begin{matrix} {}_{w}x_{obj}+20 = 12z \\ -12\,{}_{w}x_{obj}+240 = -12z \end{matrix}$$

$$\Rightarrow \begin{cases} {}_{w}x_{obj} = 260/11 \\ z = 40/11 \\ {}_{w}y_{obj} = 12z = 480/11 \end{cases}$$

(b) $x+z=0 \quad y=0 \quad z>1$

① Consider: in camera 2 setting

$$_{c_2}P_{obj} = {}_{c_2}P_w + {}_wP_{obj} = -{}_wP_{c_2} + {}_wP_{obj}$$

$$= -(20,0,0) + (-z,0,z)$$

$$= (-20-z, 0, z) \quad z>1$$

on camera 2 image

$$((-20-z)/z, 0, 1) = (u, v, 1)$$

② Consider in Camer 1 setting

$$_{c_1}P_{obj} = {}_{c_1}P_w + {}_wP_{obj} = -{}_wP_{c_1} + {}_wP_{obj}$$

$$= -(-20,0,0) + (-z, 0, z)$$

$$= (-z+20, 0, z)$$

on camera 1 image

$$((20-z)/z, 0, 1) = (x, y, 1)$$

There fore

$$\begin{cases} x = 20/z - 1 \\ u = -20/z - 1 \end{cases} \Rightarrow x+u = -2$$

Also $d = x-u$

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

problem #2

$${}^{B}_{A}T = \begin{bmatrix} R & t \\ 0^T & 1 \end{bmatrix} \qquad H_A e_A = H_B e_B = [1, 0, 0]^T$$

(a) $t = [t_x, 0, 0]^T \qquad R = I$

① $$E = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix} = [t]R \qquad B \xrightarrow{E} A$$

$$E e_A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix} \begin{bmatrix} e_{1A} \\ e_{2A} \\ e_{3A} \end{bmatrix} = \vec{0} \qquad E^T e_B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & t_x \\ 0 & -t_x & 0 \end{bmatrix} \begin{bmatrix} e_{1B} \\ e_{2B} \\ e_{3B} \end{bmatrix} = \vec{0}$$

$$\begin{cases} e_{1A} = e_{1A} \\ -t_x e_{3A} = 0 \\ t_x e_{2A} = 0 \end{cases} \Rightarrow \vec{e_A} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad \begin{cases} e_{1B} = e_{1B} \\ t_x e_{3B} = 0 \\ -t_x e_{2B} = 0 \end{cases} \Rightarrow \vec{e_B} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

② given point $[x_A, y_A, 1]$ in A

$$P_B^T E P_A = 0 \longrightarrow (E P_A)^T \cdot P_B = 0$$

where $$(E P_A)^T = \left\{ \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -t_x \\ 0 & t_x & 0 \end{bmatrix} \cdot \begin{bmatrix} x_A \\ y_A \\ 1 \end{bmatrix} \right\}^T = \begin{bmatrix} 0 \\ -t_x \\ t_x y_A \end{bmatrix}^T$$

$$\Rightarrow \begin{bmatrix} 0 & -t_x & t_x y_A \end{bmatrix} \begin{bmatrix} x_B \\ y_B \\ 1 \end{bmatrix} = 0 \qquad \therefore t_x y_B = t_x y_A$$

$$y_B = y_A$$

③ $\begin{bmatrix}1\\0\\0\end{bmatrix} = H_A e_A = \begin{bmatrix}a^T\\b^T\\c^T\end{bmatrix} e_A \rightarrow \begin{bmatrix}1\\0\\0\end{bmatrix}$

i> $\underbrace{[a_1, a_2, a_3]}_{a^T} \cdot \begin{bmatrix}1\\0\\0\end{bmatrix} = 1 \Rightarrow a^T = [1\ 0\ 0]$

ii> $\underbrace{[b_1\ b_2\ b_3]}_{b^T} \begin{bmatrix}1\\0\\0\end{bmatrix} = 0$

$\Rightarrow b^T = \frac{1}{\sqrt{e_{1A}^2 + e_{2A}^2}} [-e_{2A}, e_{1A}, 0] = [0, 1, 0]$

iii> $\underbrace{[c_1\ c_2\ c_3]}_{c^T} \begin{bmatrix}1\\0\\0\end{bmatrix} = 0$

$\Rightarrow c^T = \vec{a} \times \vec{b} = [0, 0, 1]$

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

rectification is possible.

1b) $t = [t_x, t_y, 0]^T$; $R = I$

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

$E e_A = \begin{bmatrix} 0 & 0 & t_y \\ 0 & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix} \begin{bmatrix} e_{1A} \\ e_{2A} \\ e_{3A} \end{bmatrix} = 0 \qquad E^T e_B = \begin{bmatrix} 0 & 0 & -t_y \\ 0 & 0 & +t_x \\ t_y & -t_x & 0 \end{bmatrix} \begin{bmatrix} e_{1B} \\ e_{2B} \\ e_{3B} \end{bmatrix} = 0$

$\begin{cases} t_y e_{3A} = 0 \\ -t_x e_{3A} = 0 \\ -t_y e_{1A} + t_x e_{2A} = 0 \end{cases} \Rightarrow \begin{cases} e_{1A} = \frac{t_x}{t_y} e_{2A} \\ e_{2A} = e_{2A} \\ e_{3A} = 0 \end{cases} \Rightarrow \vec{e_A} = \frac{1}{\sqrt{t_x^2 + t_y^2}} \begin{bmatrix} t_x \\ t_y \\ 0 \end{bmatrix}$

$\begin{cases} -t_y e_{3B} = 0 \\ t_x e_{3B} = 0 \\ t_y e_{1B} - t_x e_{2B} = 0 \end{cases} \Rightarrow \begin{cases} e_{1B} = t_x/t_y \, e_{2B} \\ e_{2B} = e_{2B} \\ e_{3B} = 0 \end{cases} \Rightarrow \vec{e_B} = \frac{1}{\sqrt{t_x^2 + t_y^2}} \begin{bmatrix} t_x \\ t_y \\ 0 \end{bmatrix}$

② $P_B^T E P_A \rightarrow (E P_A)^T P_B = 0$

$(E P_A)^T = \left\{ \begin{bmatrix} 0 & 0 & t_y \\ 0 & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix} \begin{bmatrix} x_A \\ y_A \\ 1 \end{bmatrix} \right\}^T = \begin{bmatrix} t_y \\ -t_x \\ -t_y x_A + t_x y_A \end{bmatrix}^T$

$(E P_A)^T P_B = 0$

$\Rightarrow t_y x_B - t_x y_B - t_y x_A + t_x y_A = 0$

③ $H_A e_A = \begin{bmatrix} a^T \\ b^T \\ c^T \end{bmatrix} \begin{bmatrix} t_x \\ t_y \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$

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

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

$$c^T = a \times b = [0, 0, 1]$$

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

//

$H_B = R H_A$

rectification is possible

1c) $t = [0, 0, t_z]$ $R = I$

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

$E\, e_A = \begin{bmatrix} 0 & -t_z & 0 \\ t_z & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} e_{1A} \\ e_{2A} \\ e_{3A} \end{bmatrix}$ $E^T e_B = \begin{bmatrix} 0 & t_z & 0 \\ -t_z & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} e_{1B} \\ e_{2B} \\ e_{3B} \end{bmatrix}$

$e_{1A} = 0$
$e_{2A} = 0$ $\Rightarrow \vec{e}_A = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$
$e_{3A} = e_{3A}$

$e_{1B} = 0$
$e_{2B} = 0$ $\Rightarrow \vec{e}_B = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$
$e_{3B} = 0$

② $(E P_A)^T P_B = \left\{ \begin{bmatrix} 0 & -t_z & 0 \\ t_z & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_A \\ y_A \\ 1 \end{bmatrix} \right\}^T \begin{bmatrix} x_B \\ y_B \\ 1 \end{bmatrix} = 0$

$\Rightarrow \begin{bmatrix} -t_z y_A & x_A t_z & 0 \end{bmatrix} \begin{bmatrix} x_B \\ y_B \\ 1 \end{bmatrix} = 0$

$-t_z y_A x_B + t_z x_A y_B = 0$

rectification is possible

③ $a^T = e_A$ $b^T = [0 \;\; 1 \;\; 0]$

$c^T = a \times b = [1 \;\; 0 \;\; 0]$

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

1(d) $t = [0, 0, 0]$ R = Arbitrary

① $E = [t]R = 0$

$E e_A = 0$ $E^T e_B = 0$

$\Rightarrow$ e_A and e_B can be any unit length vector

② $(E P_A)^T P_B = 0$ $\Rightarrow$ $0 \cdot P_B = 0$

no such line exists in image B

③ H_A and $H_B = R H_A$ Do not exist

Rectification is impossible, as H_A & H_B do not exist

problem #3
1(a)

$$I_{output} = \Phi_{derivative} * \Phi_{smooth} * I_{input}$$

$$= (\Phi_{derivative\,x} * \Phi_{smooth}) * I_{input}$$

where $\Phi_{smooth} = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$

$$\Phi_{derivative\,x} = \frac{1}{3}\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

It is like Dot product

$$\begin{bmatrix} -1/27 & -1/27 & 0 & -1/27 & -1/27 \\ 2/27 & 2/27 & 0 & -2/27 & -2/27 \\ 1/9 & 1/9 & 0 & -1/9 & -1/9 \\ 2/27 & 2/27 & 0 & -2/27 & -2/27 \\ 1/27 & 1/27 & 0 & -1/27 & -1/27 \end{bmatrix} = \Phi_{smooth} * \Phi_{derivative\,x}$$

(b) Considering most efficient way to do convolution in both before separation & After seperation

before seperation

$(n-5+1) \times (n-5+1)$ is the output image size

$\Rightarrow$ (7 additions + 1 multiplication) x 2 + (3 addition + 1 multiplication)

= 17 additions + 3 multiplication

$(n-4)(n-4) \times$ (17 additions + 3 multiplication)

$(n^2-8n+16) \times 17$ additions + $(n^2-8n+16) \times 3$ multiplication

After seperation

1) output image size After smoothing $(n-2)(n-2)$

(8 additions + 1 multiplication) $\times (n^2-4n+4)$

$= (n^2-4n+4) \times 8$ additions + $(n^2-4n+4) \times 1$ multiplication

2) output image size After derivative $(n-4)(n-4)$

(5 additions + 1 multiplication) $\times (n^2-8n+16)$

$(n^2-8n+16) \times 5$ additions + $(n^2-8n+16) \times 1$ multiplication

$\Rightarrow$ $(13n^2-72n+112)$ additions + $(2n^2-12n+20)$ multiplication

2) Before seperation

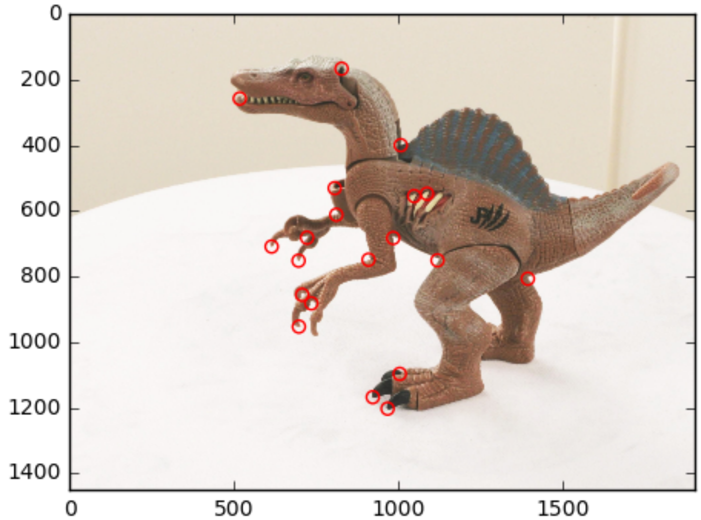
$(17n^2-136n+272)$ additions + $(3n^2-24n+48)$ multiplication

After seperation

$(13n^2-72n+112)$ additions + $(2n^2-12n+20)$ multiplication

Therefore for large n, After seperation is preferable

and for small n Before sepperation is preferable




```
In [156]: import numpy as np
          from scipy.misc import imread
          import matplotlib.pyplot as plt
          from scipy.ndimage.filters import gaussian_filter
          import cv2

In [157]: def rgb2gray(rgb):
              """ Convert rgb image to grayscale.
              """
              return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])

In [158]: def imconv(img, operator):
              conved_img = np.zeros(img.shape)
              r_center = int((operator.shape[0]-1)/2)
              c_center = int((operator.shape[1]-1)/2)
              num_row, num_col = img.shape

              for i in range(r_center, num_row-r_center):
                  for j in range(c_center, num_col-c_center):
                      conved_img[i][j] = (img[(i-r_center):(i+r_center+1),\
                                              (j-c_center):(j+c_center+1)] * operator).sum()
              return conved_img #[r_center : (num_row-r_center), c_center : (num_col-c_center)]


          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.
```

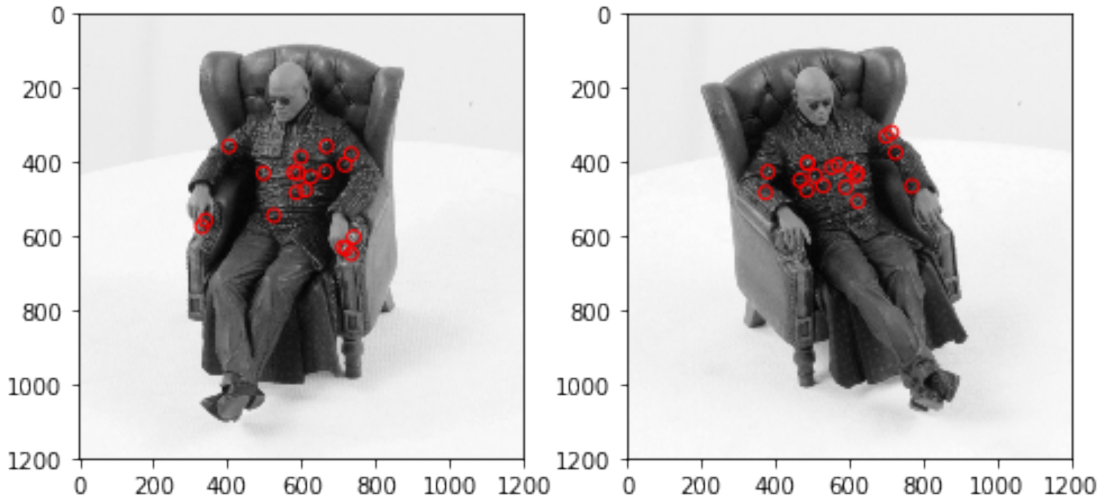
```
        smoothSTD: Standard deviation of the Gaussian smoothing kernel.
        windowSize: Window size for corner detector and non maximum suppression.

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

    """

    """
    Your code here:
    """
    smoothed_img = gaussian_filter(image,sigma=smoothSTD)

    sum_window = np.ones((windowSize, windowSize))
    img_dy, img_dx = np.gradient(smoothed_img) # y is row, x i column

    img_dxdx = img_dx * img_dx
    img_dydy = img_dy * img_dy
    img_dxdy = img_dx * img_dy
    cov_sum_dxdx = imconv(img_dxdx, sum_window)
    cov_sum_dxdy = imconv(img_dxdy, sum_window)
    cov_sum_dydy = imconv(img_dydy, sum_window)

    window_center = int((windowSize-1)/2)
    img_lmbd = np.zeros(image.shape)
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            covariance = np.array([[cov_sum_dxdx[i][j], cov_sum_dxdy[i][j]],\
                                   [cov_sum_dxdy[i][j], cov_sum_dydy[i][j]]])
            lmbds, egnvctrs = np.linalg.eig(covariance)
            img_lmbd[i][j] = min(lmbds)


    corner_coord_cndts = []
    corner_lmbd_cndts = []
    for i in range(window_center, image.shape[0]-window_center):
        for j in range(window_center, image.shape[1]-window_center):

            patch_lmbd = img_lmbd[i-window_center: i+window_center+1,\
                                  j-window_center: j+window_center+1]

            if patch_lmbd[window_center][window_center] == np.max(patch_lmbd):
                # it is local maximum
                corner_coord_cndts.append([j, i]) # y, x
                corner_lmbd_cndts.append(np.max(patch_lmbd))


    corner_lmbd_cndts = np.array(corner_lmbd_cndts)[:, None]
```

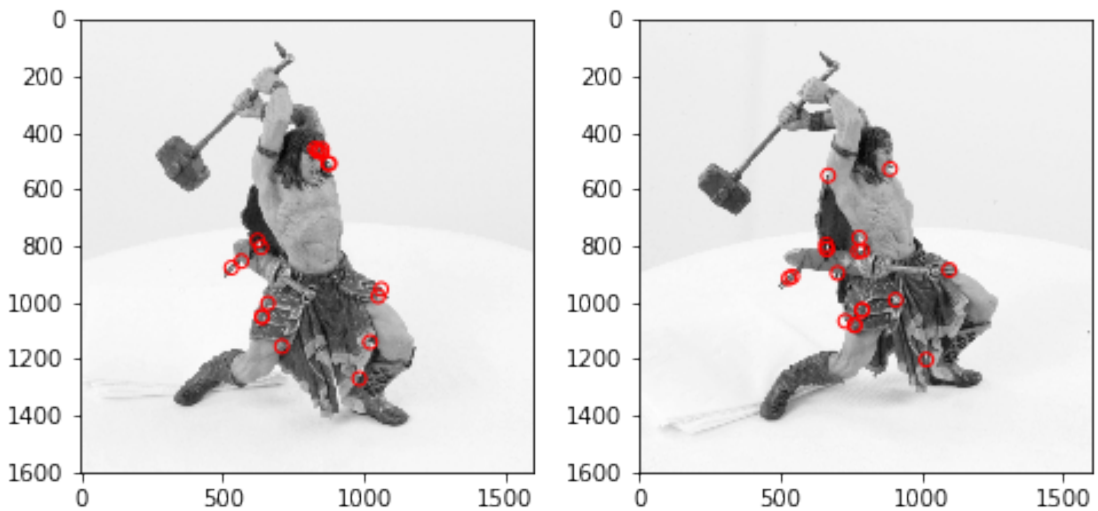
```
        corner_coord_cndts = np.array(corner_coord_cndts)

        unique_corner_lmbds = np.sort(corner_lmbd_cndts, axis=0)
        threshold = unique_corner_lmbds[-nCorners]
        idx_row, idx_col = np.where(corner_lmbd_cndts > threshold)
        corners = corner_coord_cndts[idx_row]
        return corners #corners N *3


    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', facecolors

        ax2 = fig.add_subplot(222)
        ax2.imshow(imgs[1], cmap='gray')
        ax2.scatter(corners[1][:, 0], corners[1][:, 1], s=35, edgecolors='r', facecolors
        plt.show()
```

```
In [159]: # detect corners on warrior and matrix sets
          # adjust your corner detection parameters here
          nCorners = 20
          windowSize = 3
          smoothSTD = 0.5

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

          show_corners_result(imgs_mat, crns_mat)
          show_corners_result(imgs_war, crns_war)
```

```
/usr/local/lib/python3.7/site-packages/ipykernel_launcher.py:13: DeprecationWarning: `imread` i
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
```

```
Use ``imageio.imread`` instead.
  del sys.path[0]
/usr/local/lib/python3.7/site-packages/ipykernel_launcher.py:19: DeprecationWarning: `imread` i
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
```

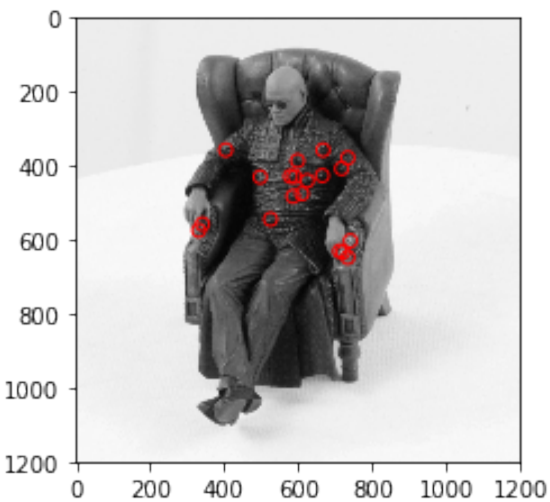


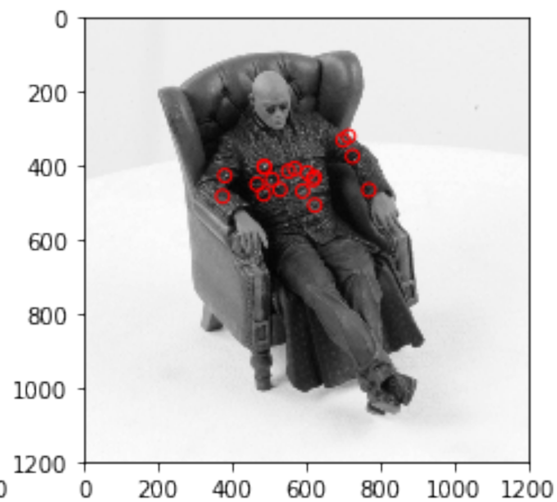


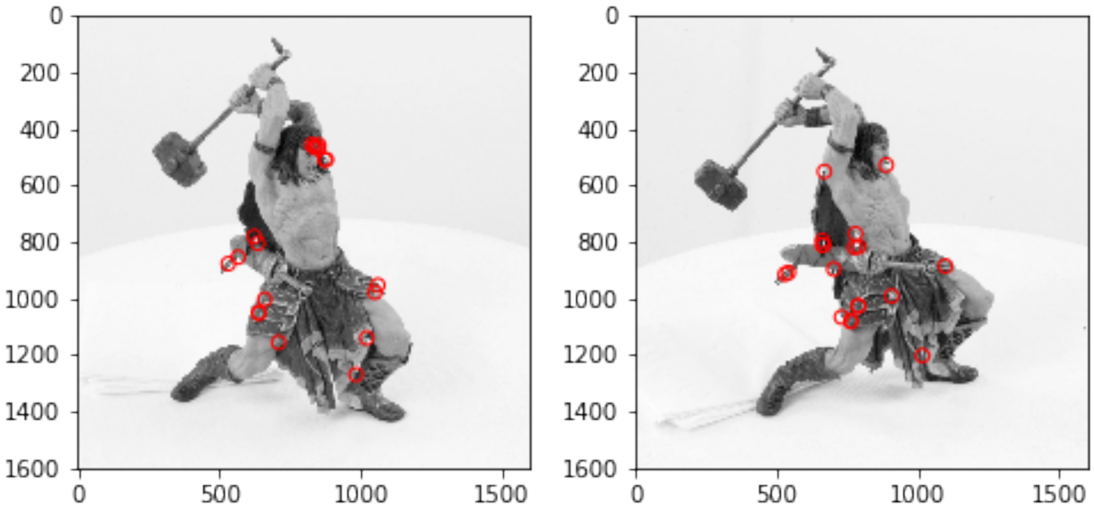

```
In [160]: nCorners = 20
          for windowSize in [3, 5, 9, 17]:
              for smoothSTD in [0.5, 1, 2, 4]:
                  print('windowSize is {0}, and smoothSTD is {1}'.format(windowSize, smoothSTD)
```

```python
# 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, windowSi

    img_war = imread('p4/warrior/warrior' + str(i) + '.png')
    imgs_war.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.append(corner_detect(imgs_war[i], nCorners, smoothSTD, windowSi

show_corners_result(imgs_mat, crns_mat)
show_corners_result(imgs_war, crns_war)
print('\n')
```

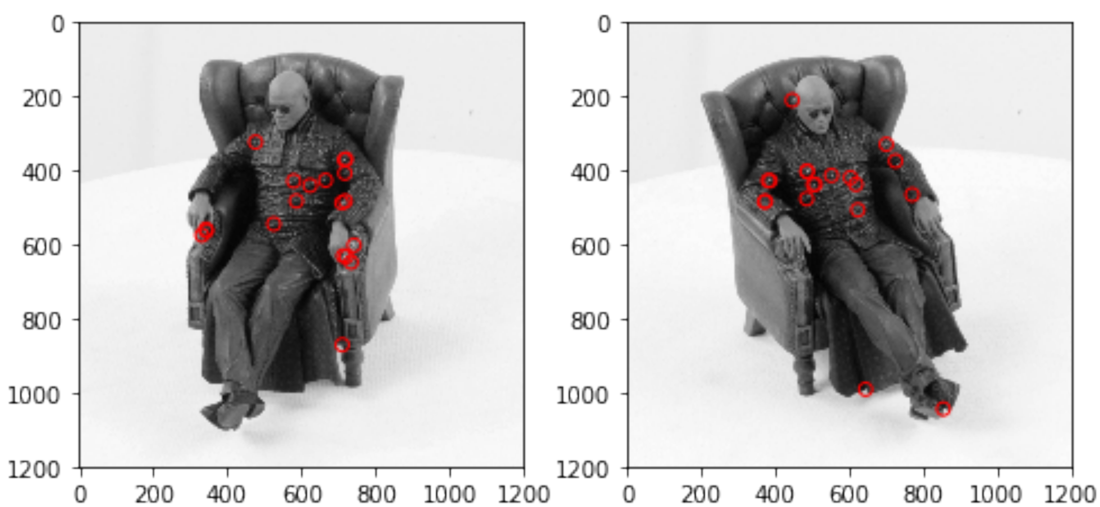
```
windowSize is 3, and smoothSTD is 0.5


/usr/local/lib/python3.7/site-packages/ipykernel_launcher.py:11: DeprecationWarning: `imread`
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  # This is added back by InteractiveShellApp.init_path()
/usr/local/lib/python3.7/site-packages/ipykernel_launcher.py:17: DeprecationWarning: `imread`
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
```

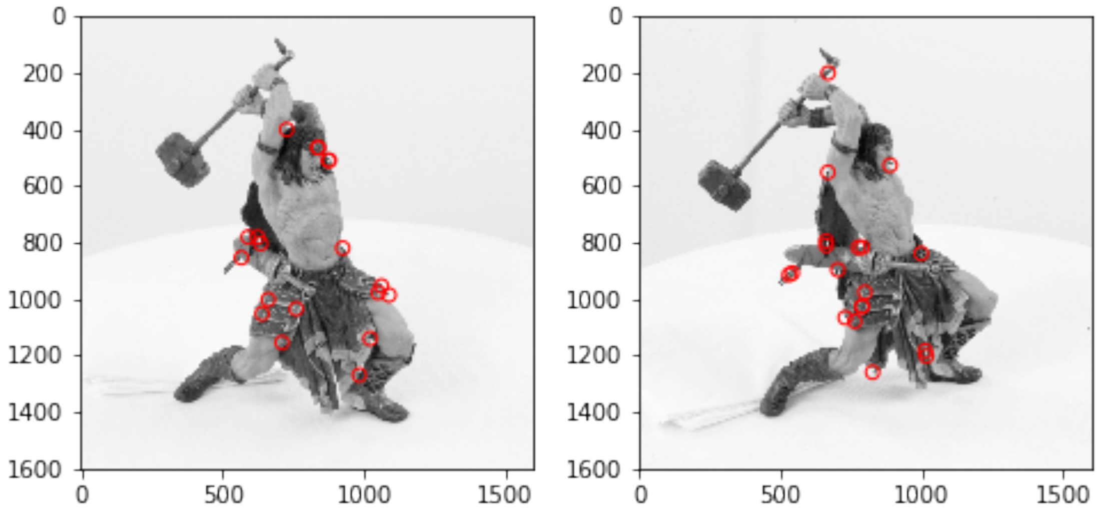




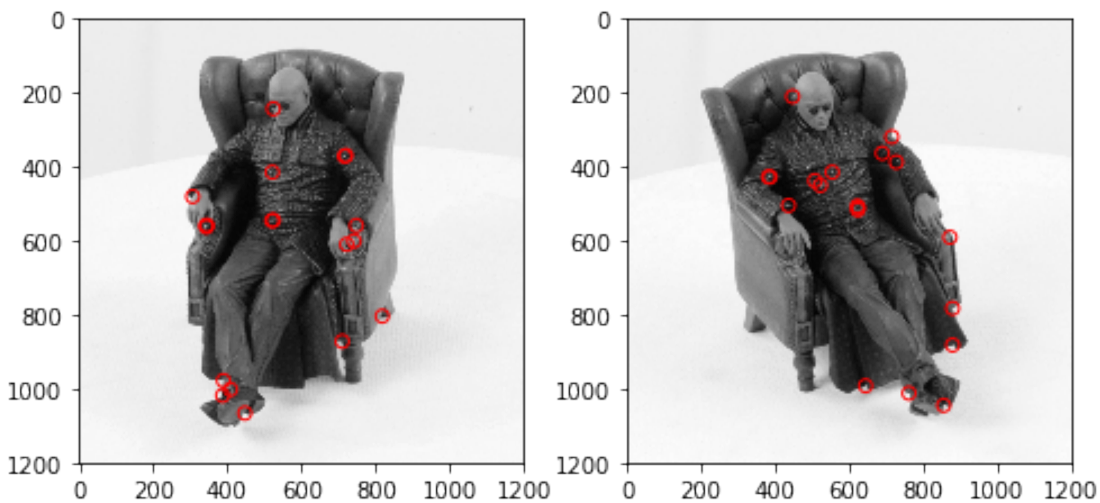


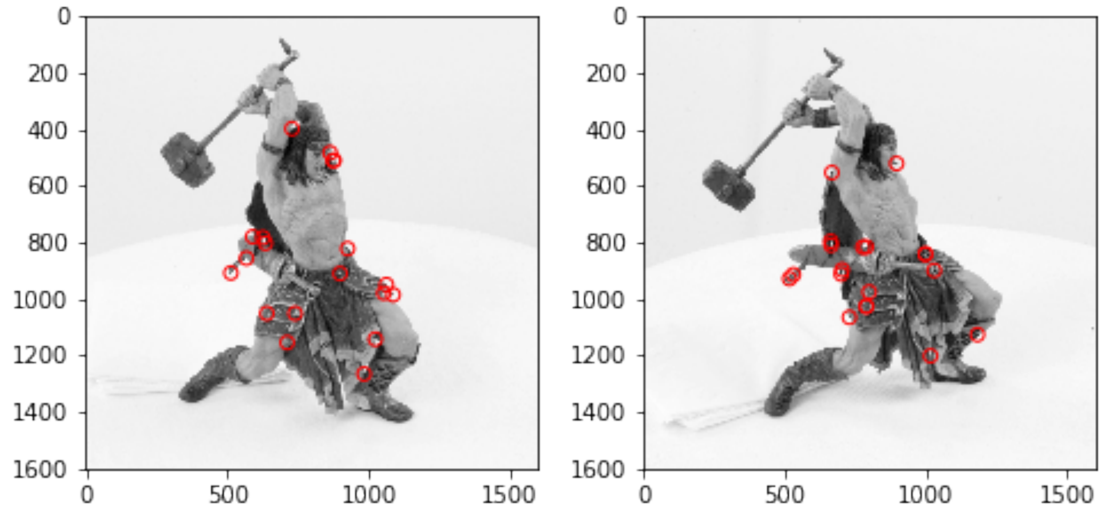
windowSize is 3, and smoothSTD is 1



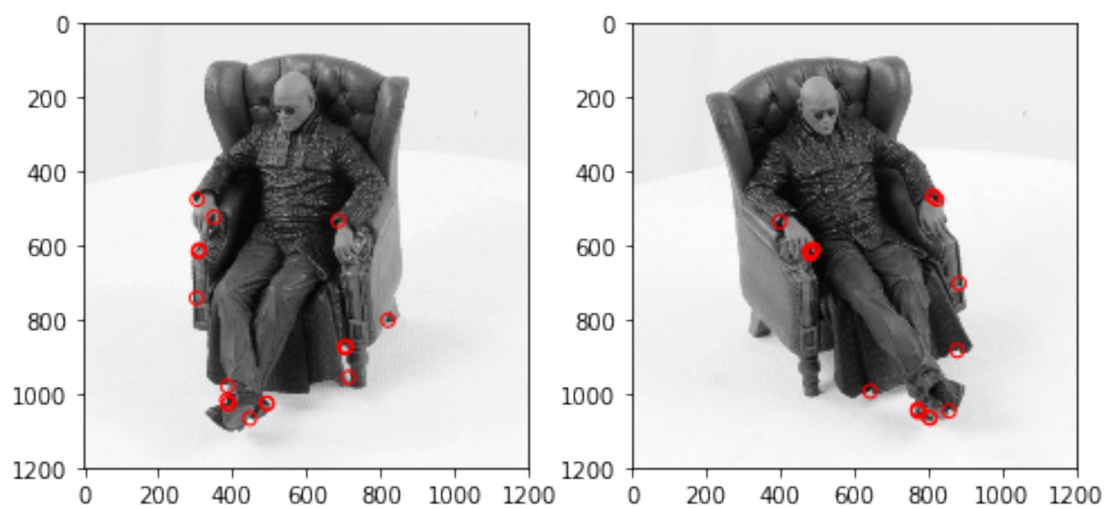


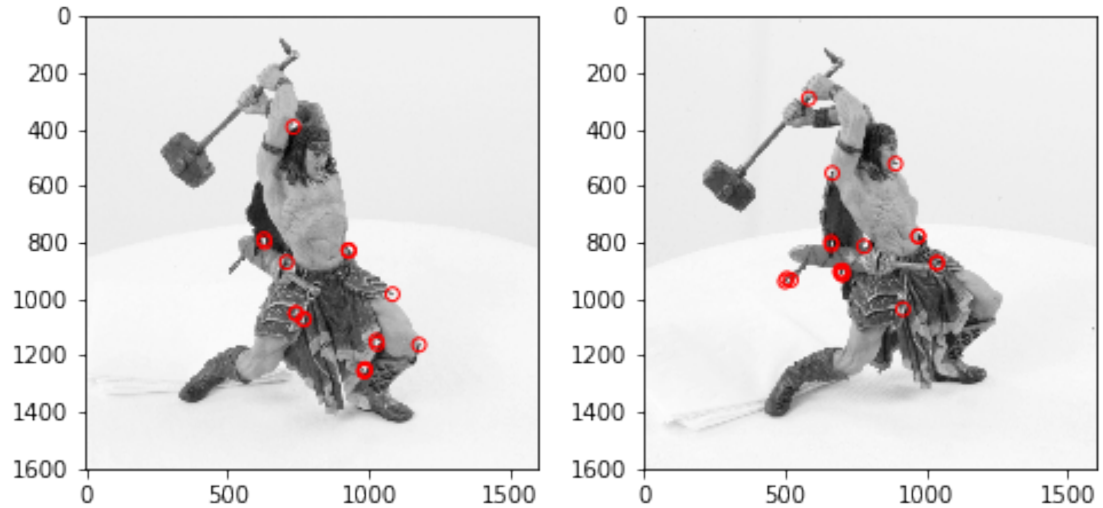
windowSize is 3, and smoothSTD is 2



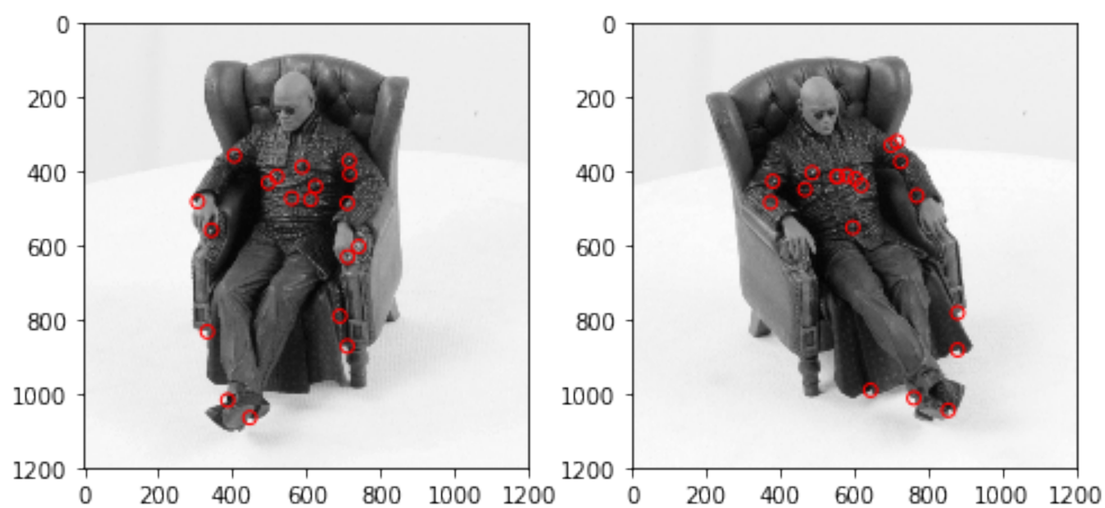


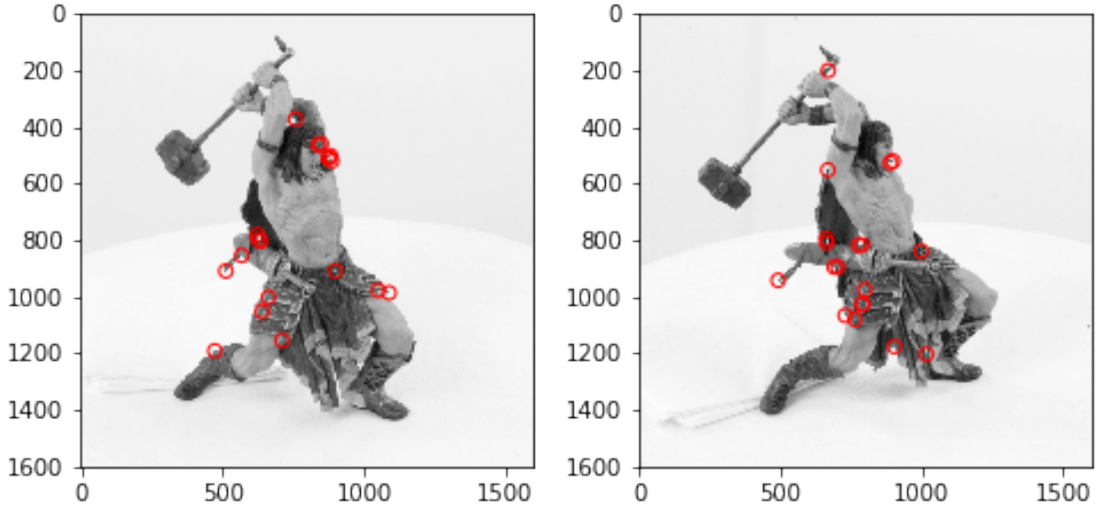
windowSize is 3, and smoothSTD is 4



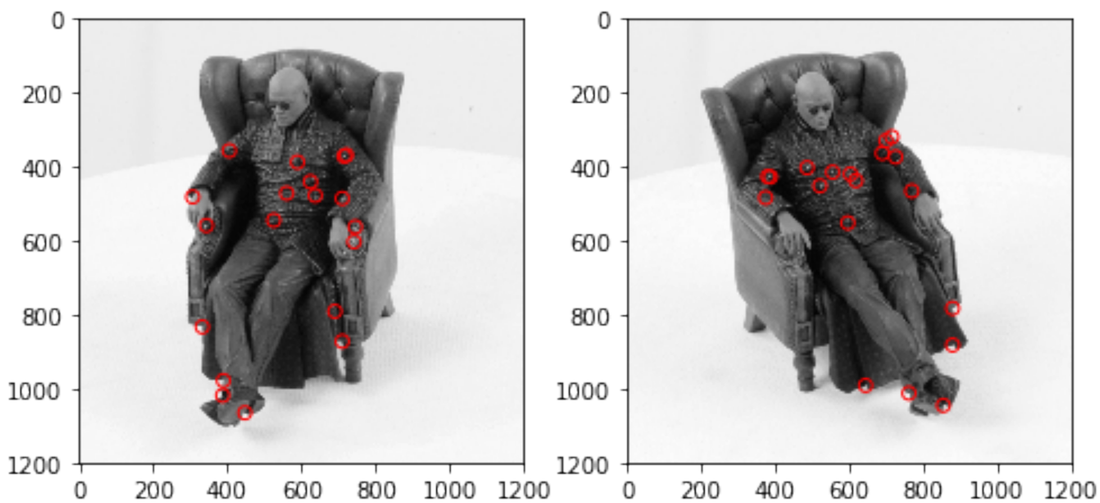


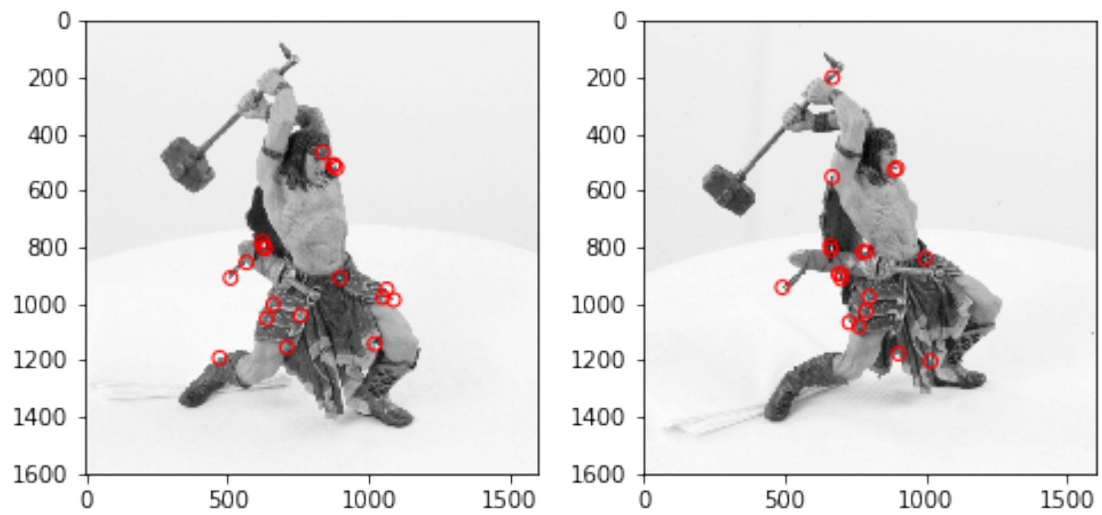
`windowSize is 5, and smoothSTD is 0.5`



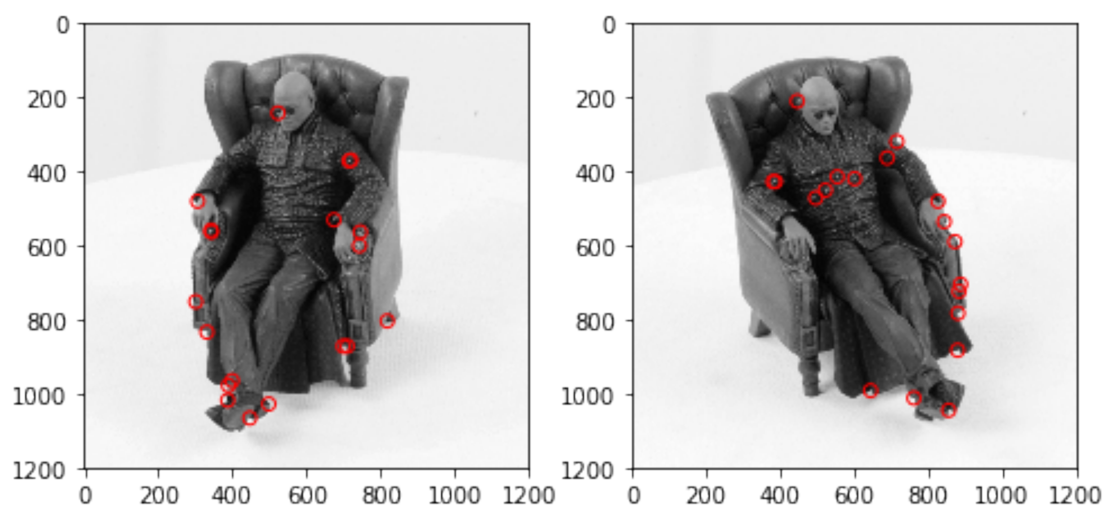


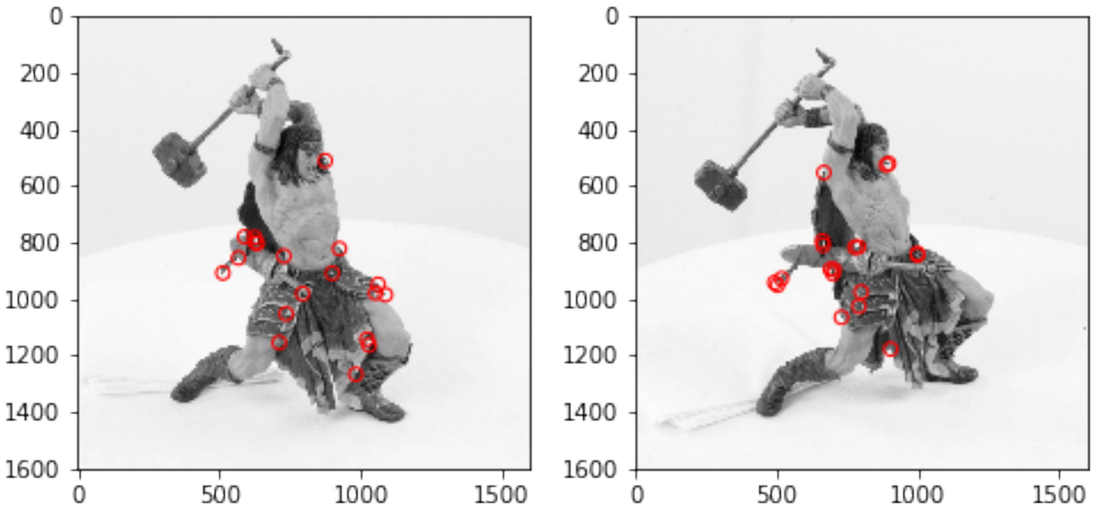
`windowSize is 5, and smoothSTD is 1`



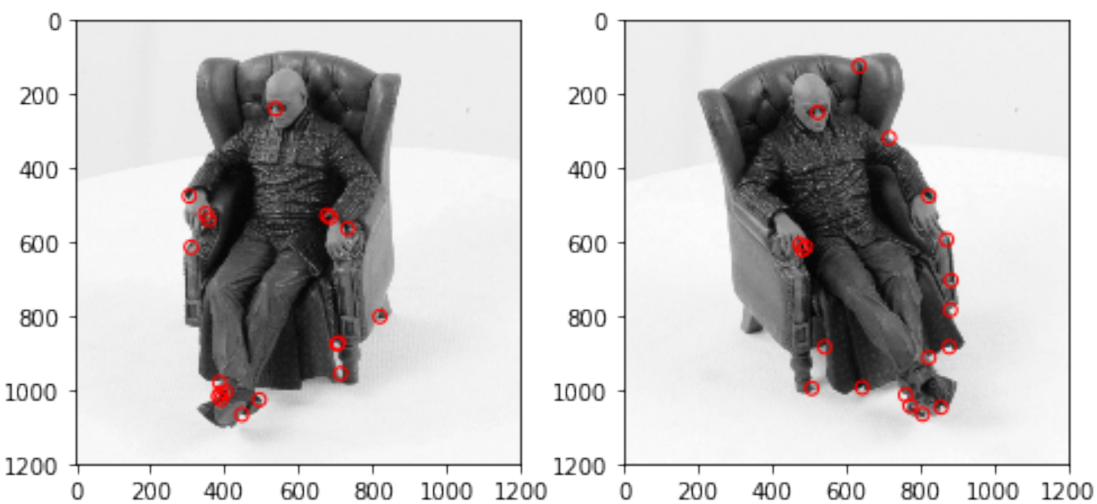


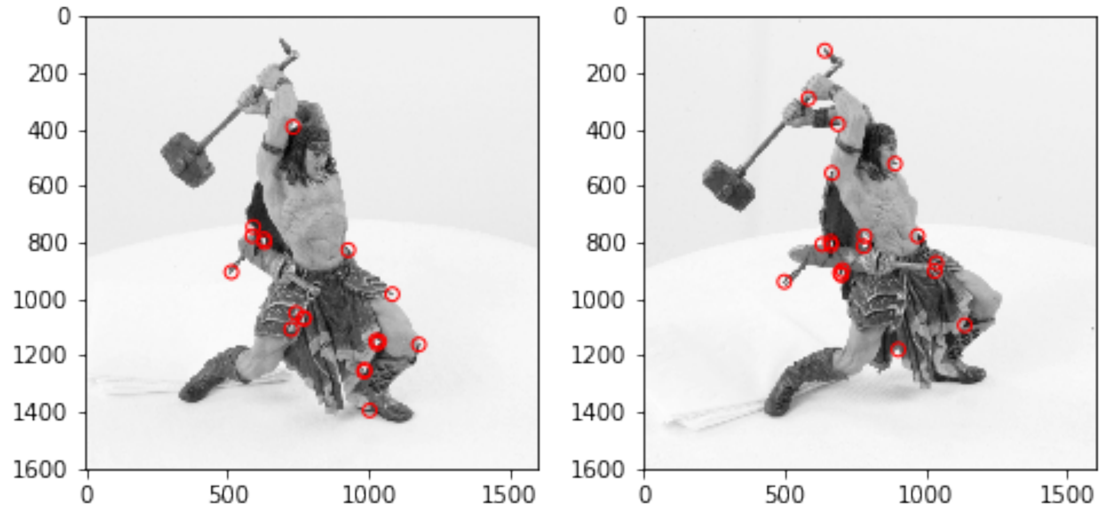
windowSize is 5, and smoothSTD is 2



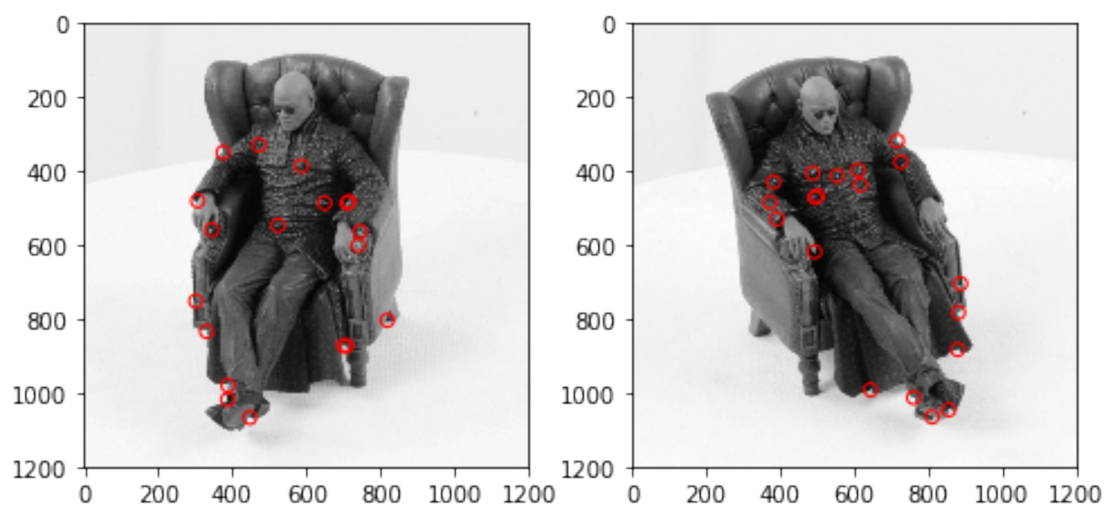


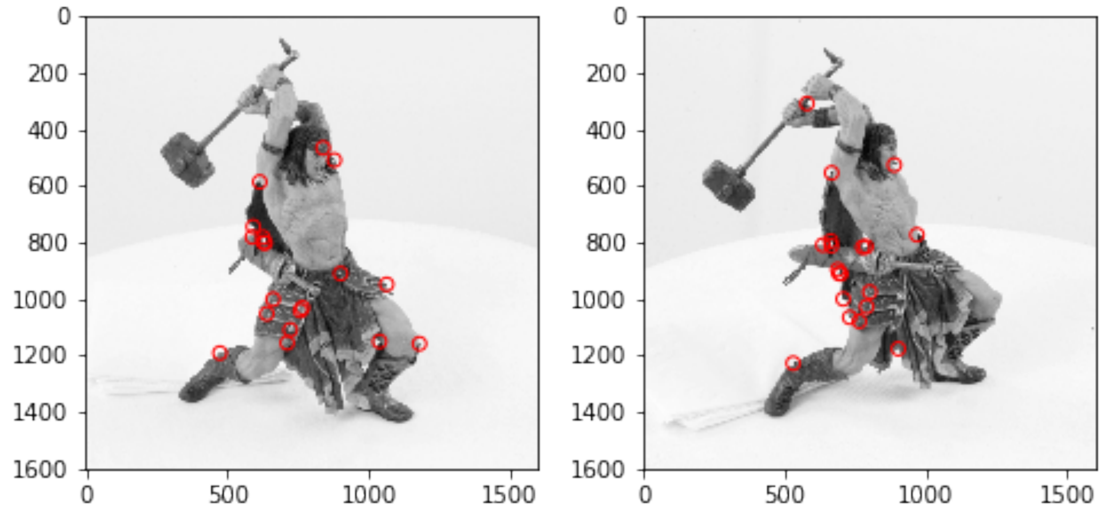
windowSize is 5, and smoothSTD is 4



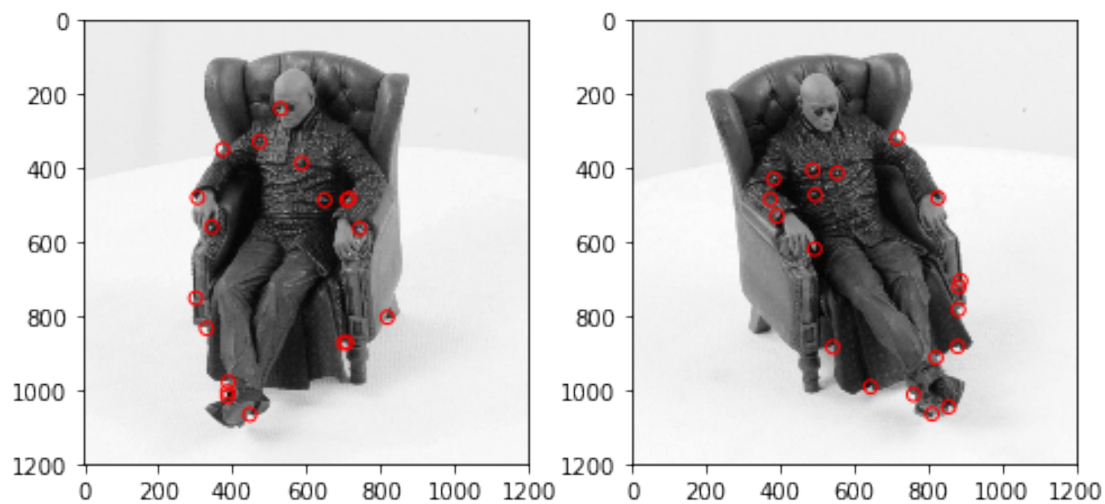


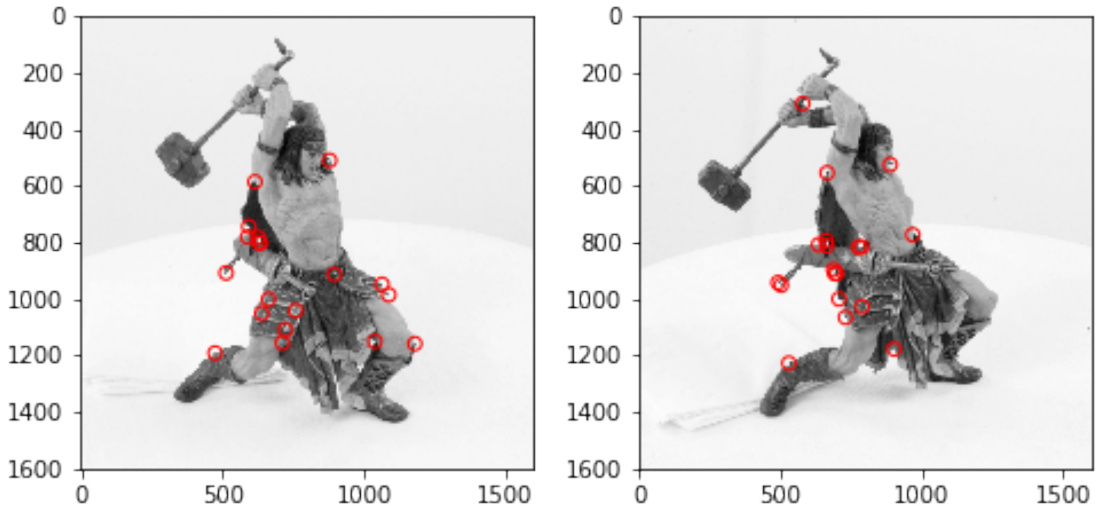
`windowSize is 9, and smoothSTD is 0.5`



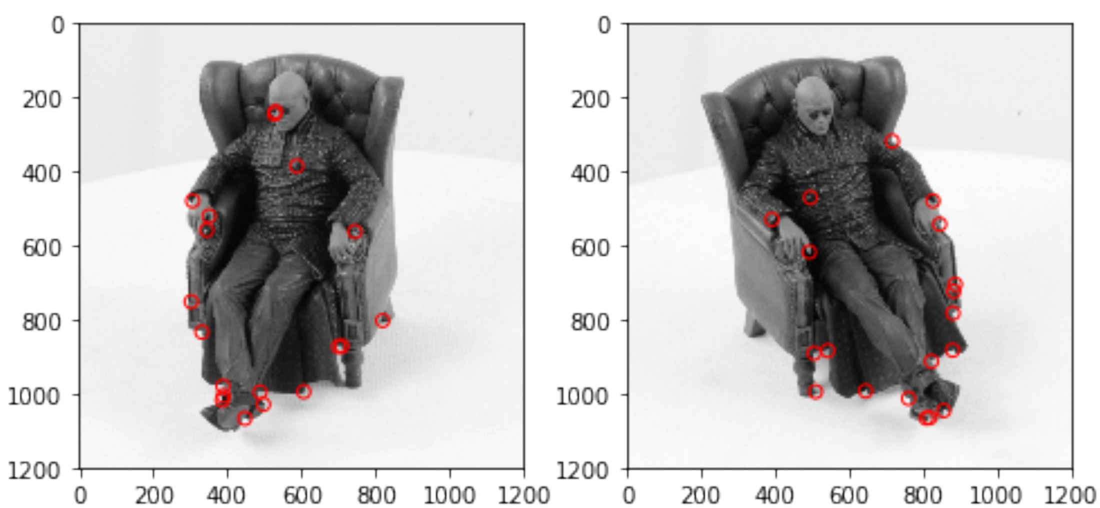


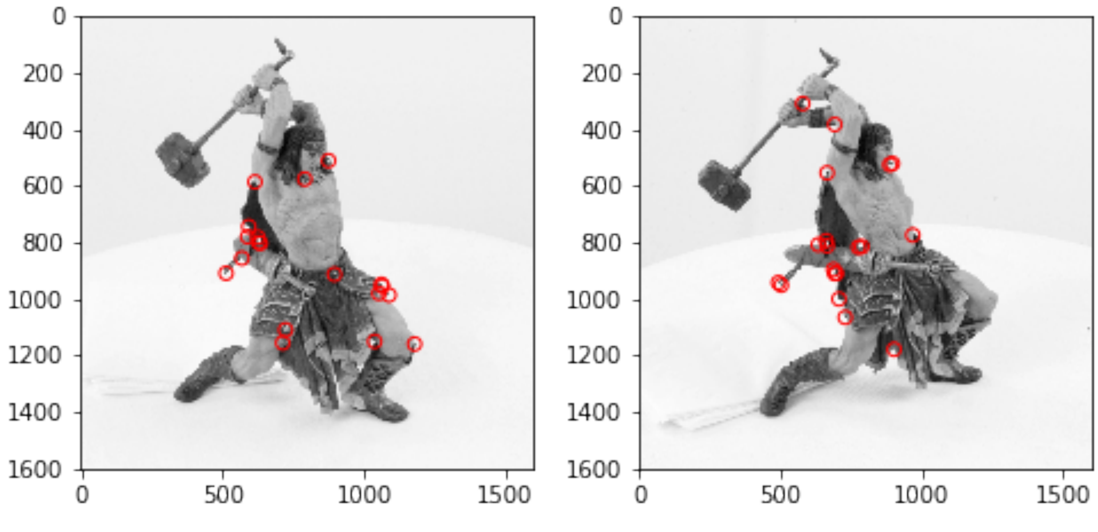
windowSize is 9, and smoothSTD is 1



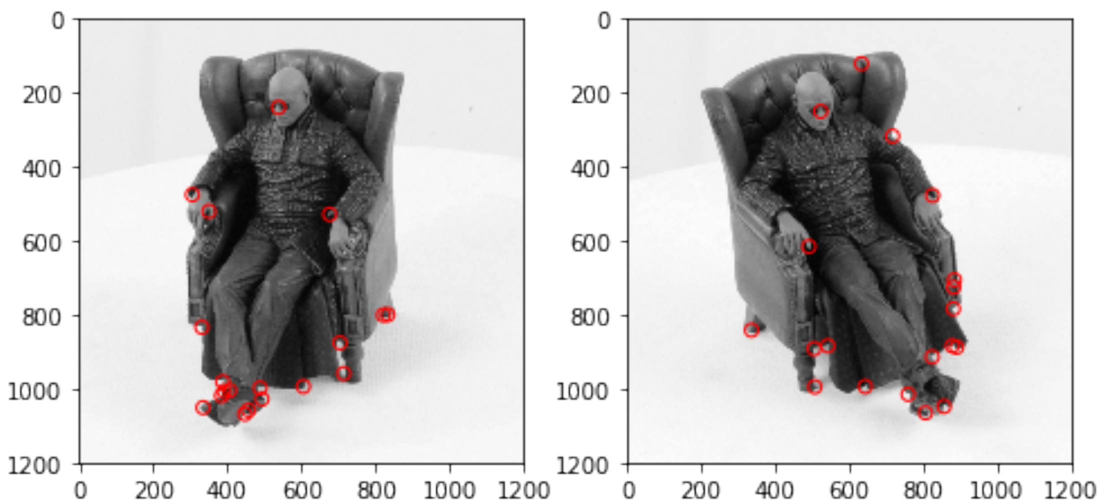


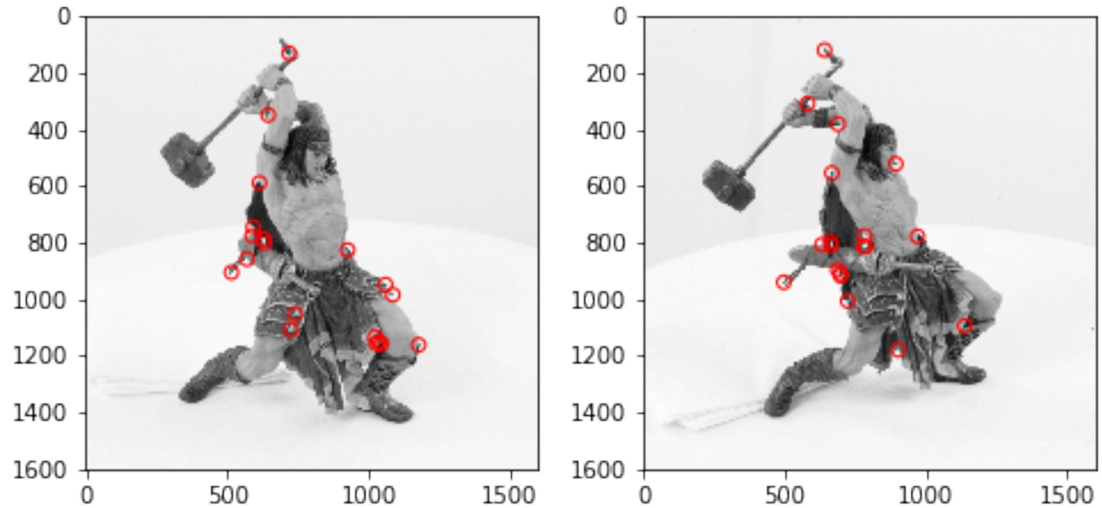
`windowSize is 9, and smoothSTD is 2`



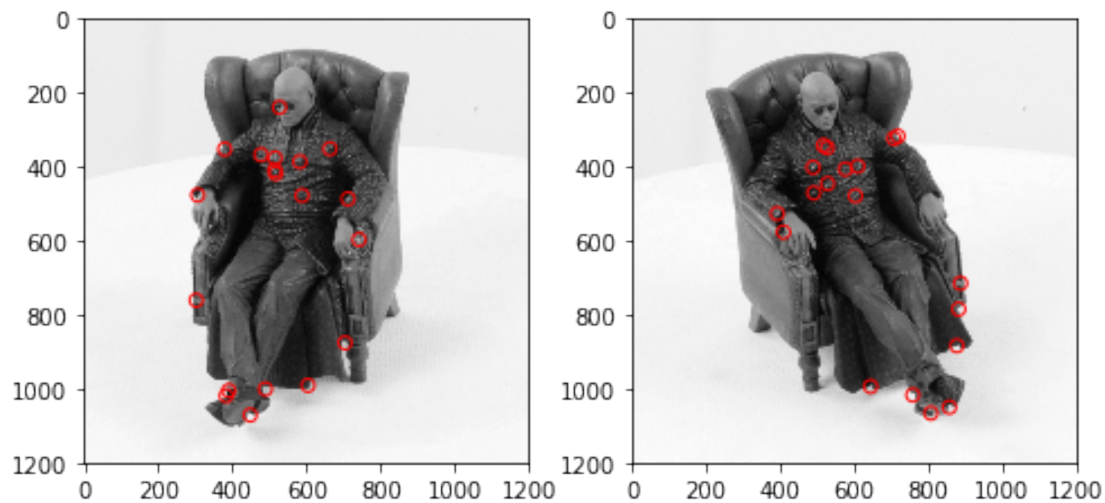


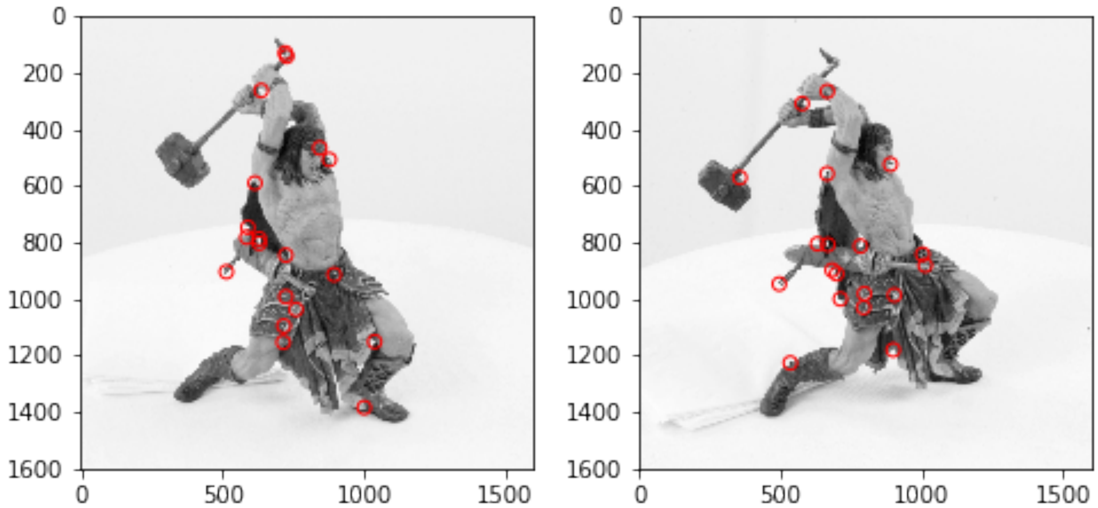
`windowSize is 9, and smoothSTD is 4`



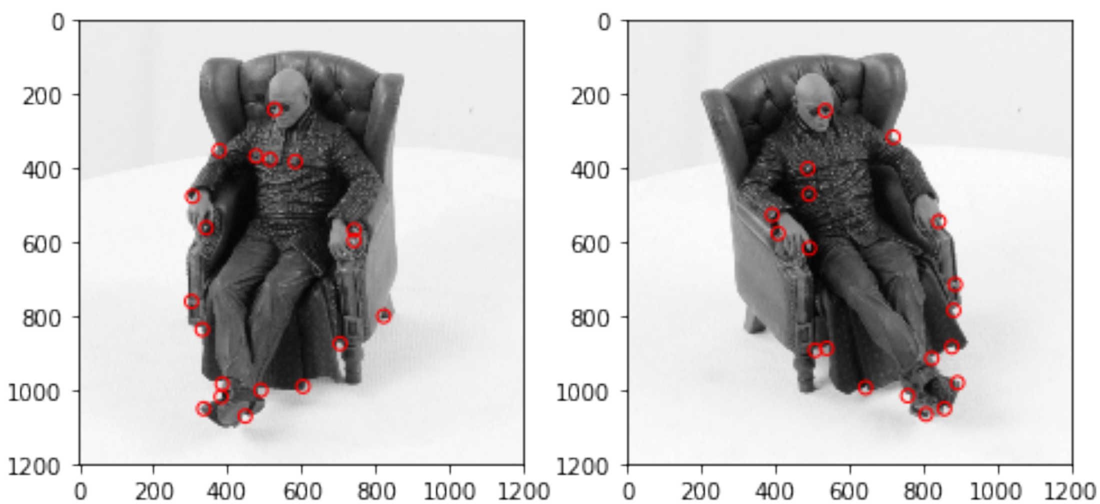


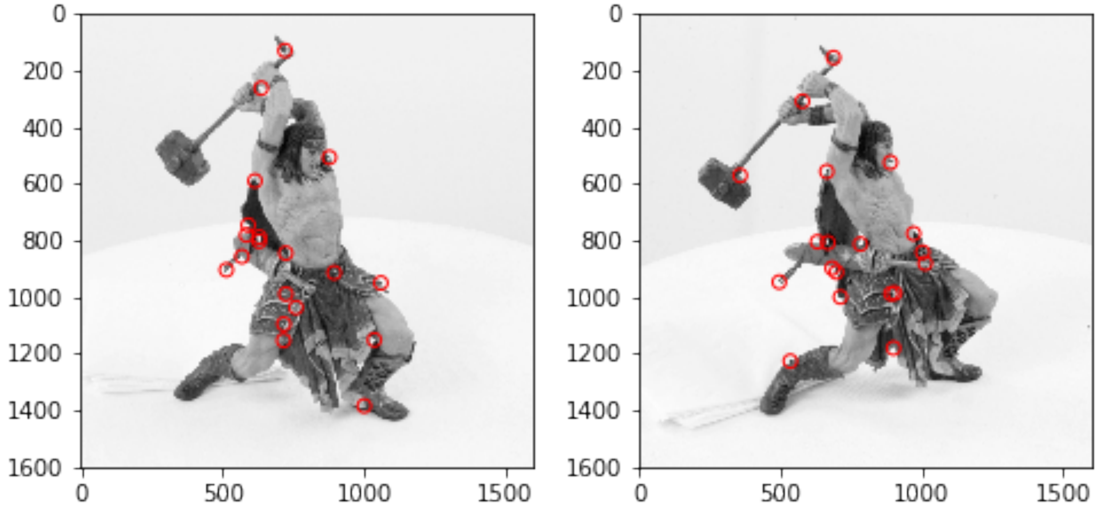
windowSize is 17, and smoothSTD is 0.5



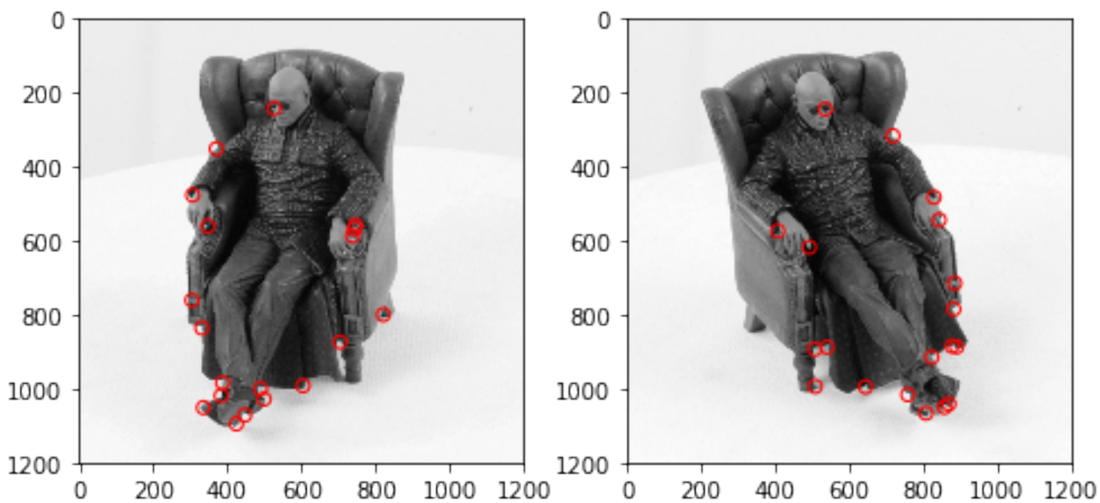


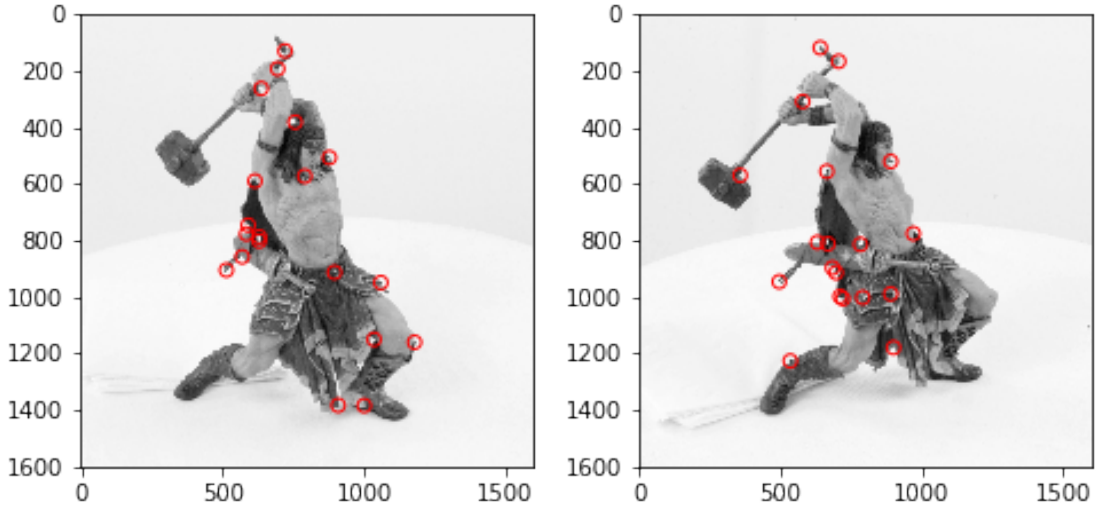
`windowSize is 17, and smoothSTD is 1`



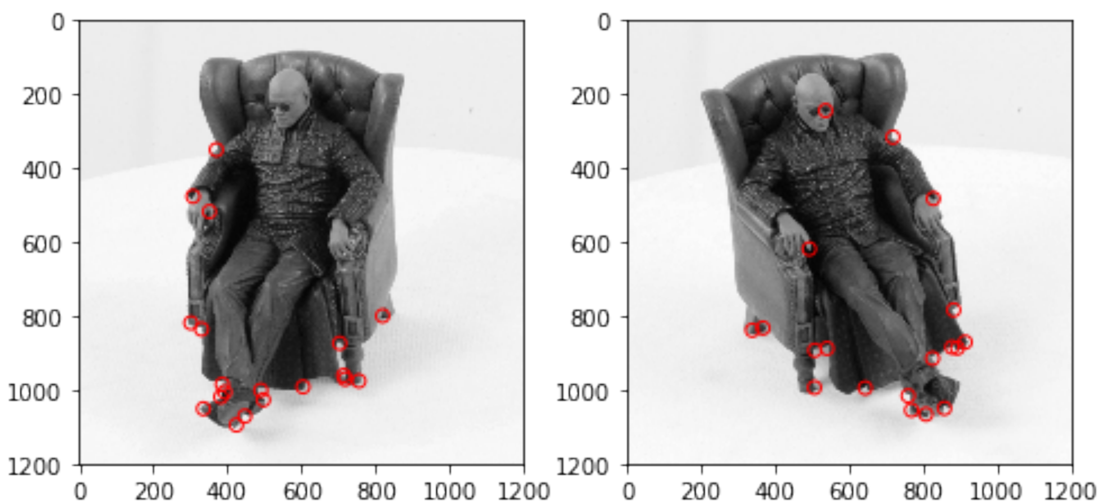


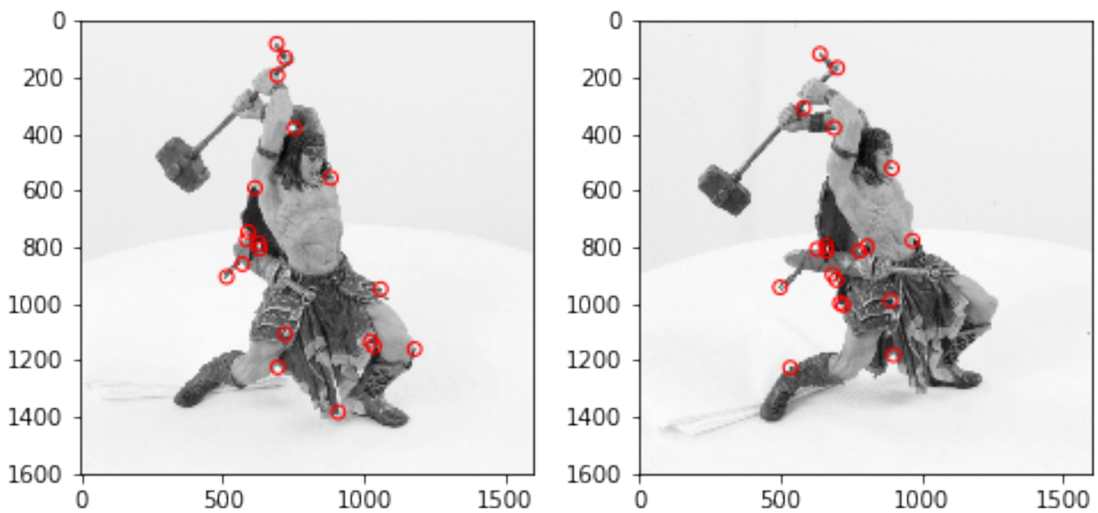
windowSize is 17, and smoothSTD is 2





windowSize is 17, and smoothSTD is 4





In general corner detection works well on large smoothSTD, since with large smoothSTD, some detail being averaged, the detection algorithm can focus on more abstract corner features. Also, with relative large window size, the detection variance can be decreased, which can make detection more accurate.

1.5.2 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 $\tilde{W} = \frac{W-\overline{W}}{\sqrt{\sum_{k,l}(W(k,l)-\overline{W})^2}}$ is a mean-shifted and normalized version of the window and $\overline{W}$ is the mean pixel value in the window W.

```
In [7]: 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.

            """
```

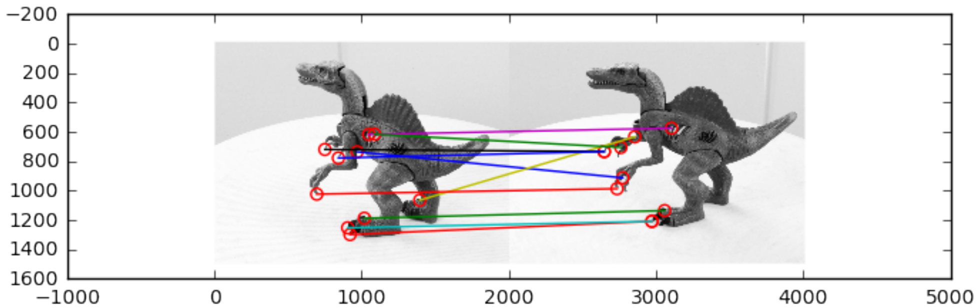
```
    """
    Your code here:
    """
    img1_patch = img1[(c1[1]-R):(c1[1]+R+1), (c1[0]-R):(c1[0]+R+1)]
    img1_patch_mean = np.mean(img1_patch)
    norm1 = np.sum((img1_patch - img1_patch_mean)**2)
    W1 = (img1_patch - img1_patch_mean) / np.sqrt(norm1)


    img2_patch = img2[(c2[1]-R):(c2[1]+R+1), (c2[0]-R):(c2[0]+R+1)]
    img2_patch_mean = np.mean(img2_patch)
    norm2 = np.sum((img2_patch - img2_patch_mean)**2)
    W2 = (img2_patch - img2_patch_mean) / np.sqrt(norm2)


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

```
In [8]: # 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
```

1.5.3 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

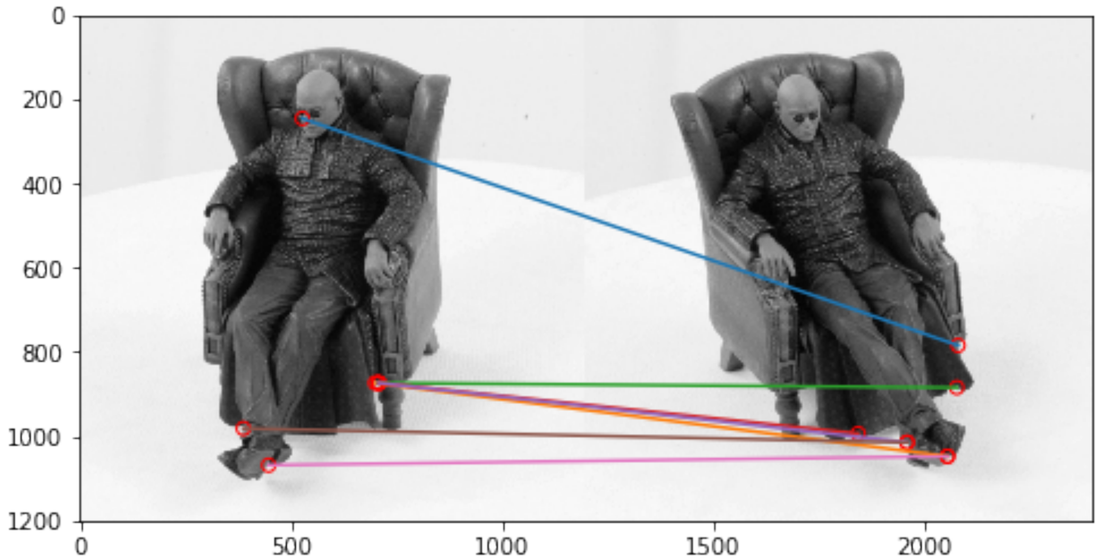


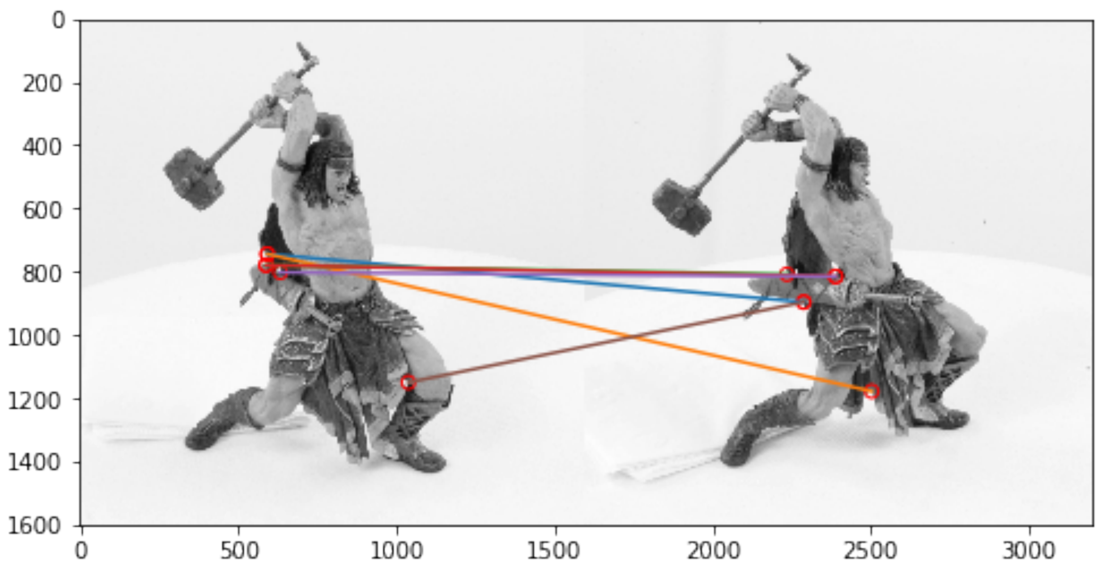
code.

```
In [161]: 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 c1 in corners1:
                  for c2 in corners2:
                      if ncc_match(img1, img2, c1, c2, R) > NCCth:
                          matching.append((c1, c2))
              return matching


          # 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 differ
              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='r
                  plt.plot([p1[0], p2[0] + img1.shape[1]], [p1[1], p2[1]])
              plt.savefig('dino_matching.png')
```

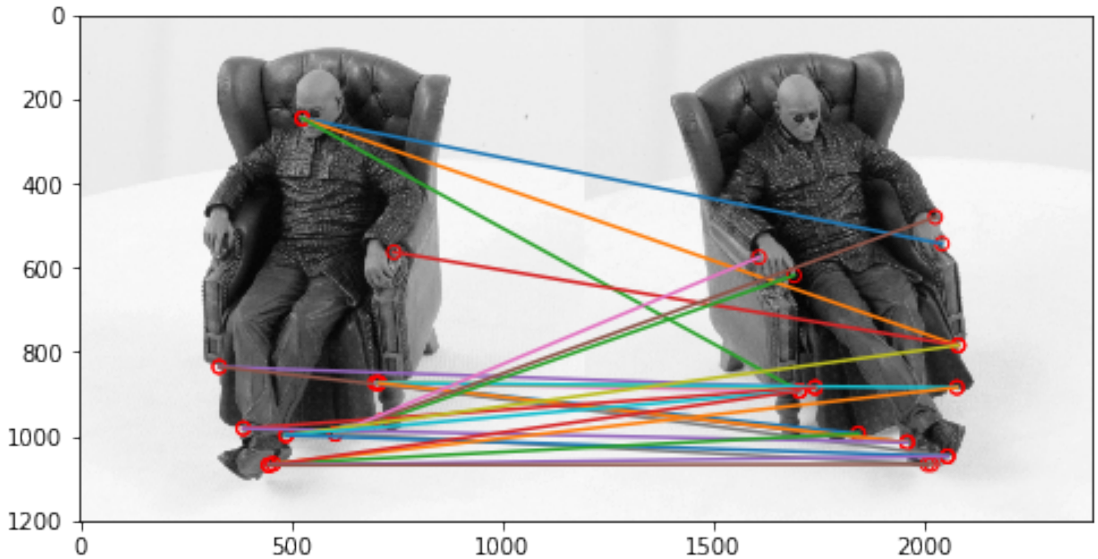
```
        plt.show()

In [162]: smoothSTD = 2
          windowSize = 11
          R = 15
          NCCth = 0.7

In [163]: # detect corners on warrior and matrix sets
          # adjust your corner detection parameters here

          for nCorners in [10, 20 , 30]:
              print('number of corner is ', nCorners)
              # 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))
                  # downsize your image in case corner_detect runs slow in test
                  # imgs_war.append(rgb2gray(img_war)[::2, ::2])
                  crns_war.append(corner_detect(imgs_war[i], nCorners, smoothSTD, windowSize))

              # match corners
              matching_mat = naive_matching(imgs_mat[0]/255, imgs_mat[1]/255, crns_mat[0], crns
              matching_war = naive_matching(imgs_war[0]/255, imgs_war[1]/255, crns_war[0], crns
              show_matching_result(imgs_mat[0], imgs_mat[1], matching_mat)
              show_matching_result(imgs_war[0], imgs_war[1], matching_war)
              print('\n')
```

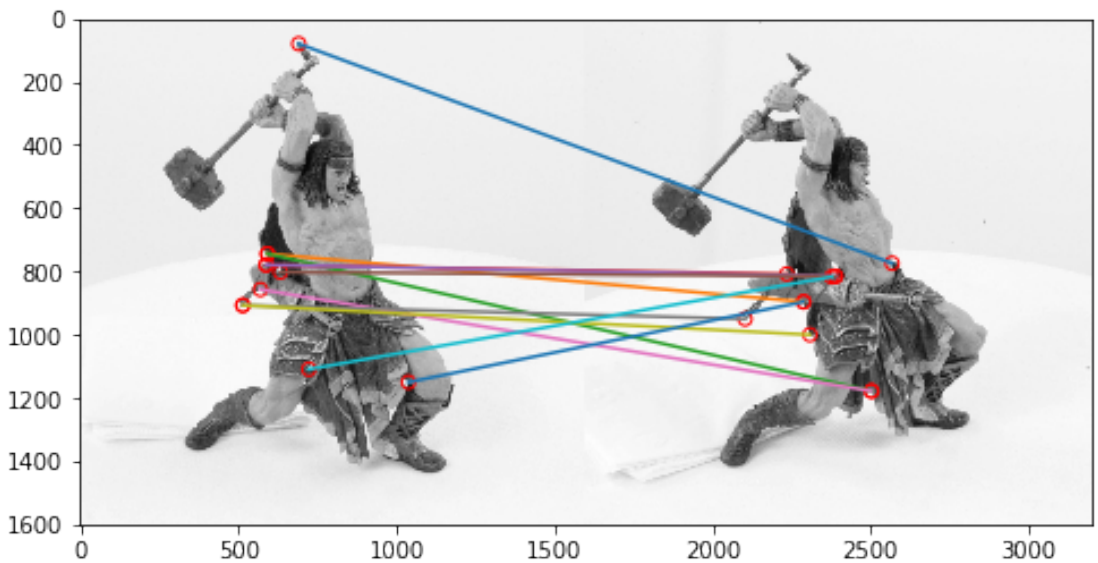
```
number of corner is  10


/usr/local/lib/python3.7/site-packages/ipykernel_launcher.py:12: DeprecationWarning: `imread` i
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  if sys.path[0] == '':
/usr/local/lib/python3.7/site-packages/ipykernel_launcher.py:18: DeprecationWarning: `imread` i
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
```



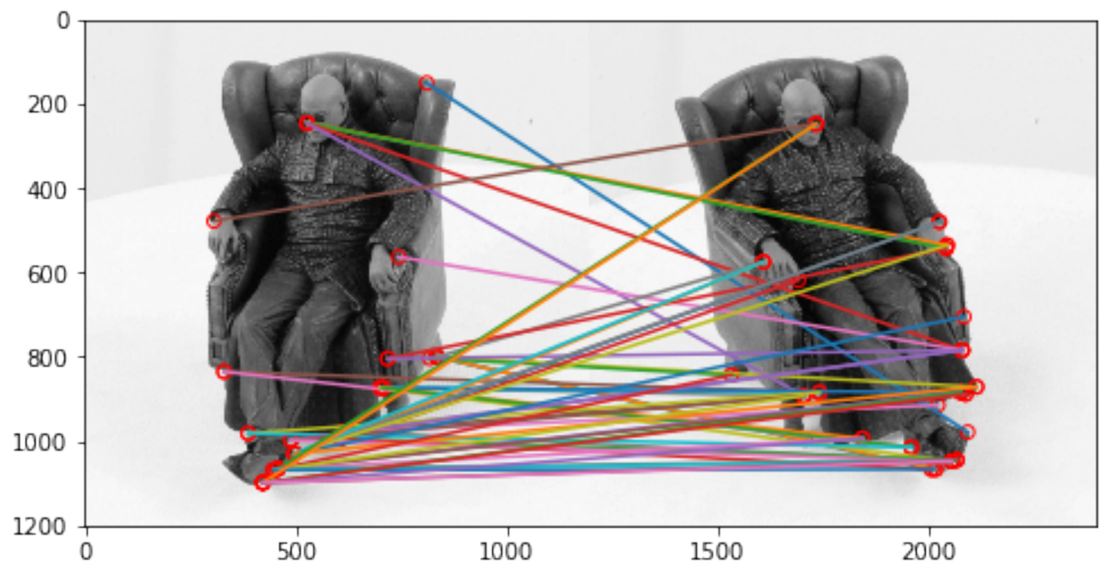


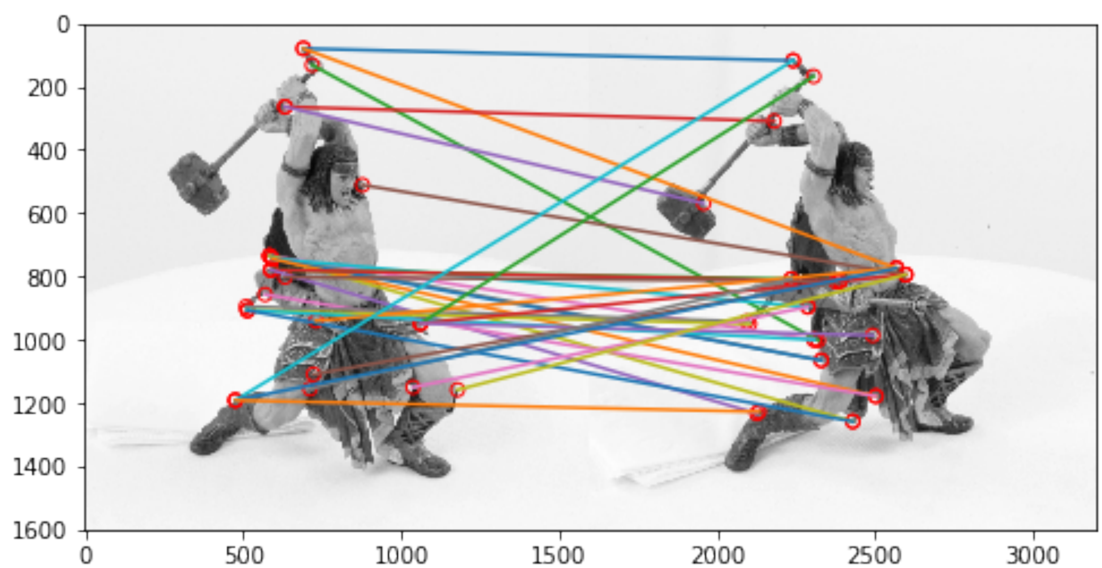

```
number of corner is  20
```






```
number of corner is  30
```

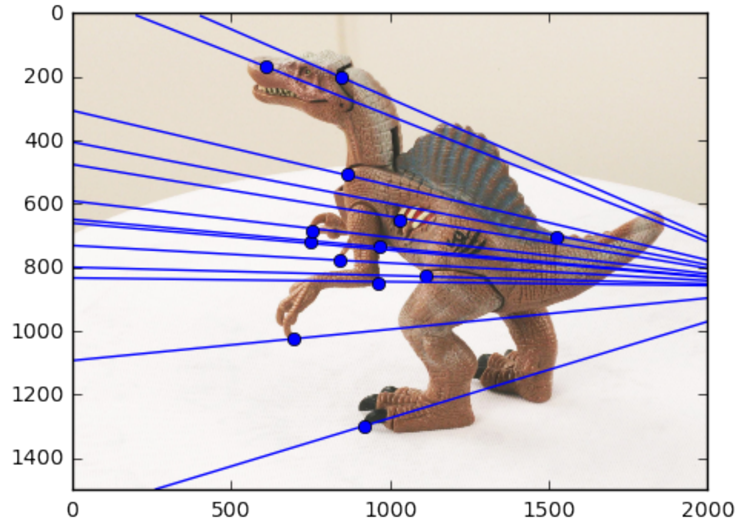




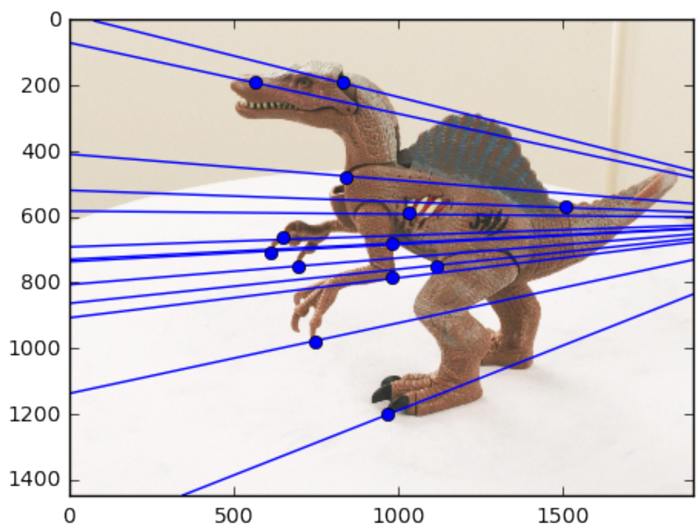
1.5.4 Epipolar Geometry [4 pts]

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.



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

```
In [46]: ## 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]
             """
```

```
    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) # minimum eigen value


    # 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):
    #3*n
    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 corner poi
        input:
        F--> Fundamental matrix
        output:
        e1--> corresponding epipole in image 1
        e2--> epipole in image2
        '''
        #your code here


    #     e1_lmbds, e1_egnvctrs = np.linalg.eig(F.T)
    #     e1_idx = np.argmin(abs(e1_lmbds))
    #     e1 = e1_egnvctrs[:, e1_idx]

    #     e2_lmbds, egnvctrs = np.linalg.eig(F)
    #     e2_idx = np.argmin(abs(e2_lmbds))
    #     e2 = e1_egnvctrs[:, e2_idx]
        U,S,Vh = np.linalg.svd(F)
        # e2 is last column of V
        # e1 is last column of U
        e1 = U[:, -1]
        e2 = Vh[-1]

        e1 = e1/e1[-1]
        e2 = e2/e2[-1]

        return e1, e2
```

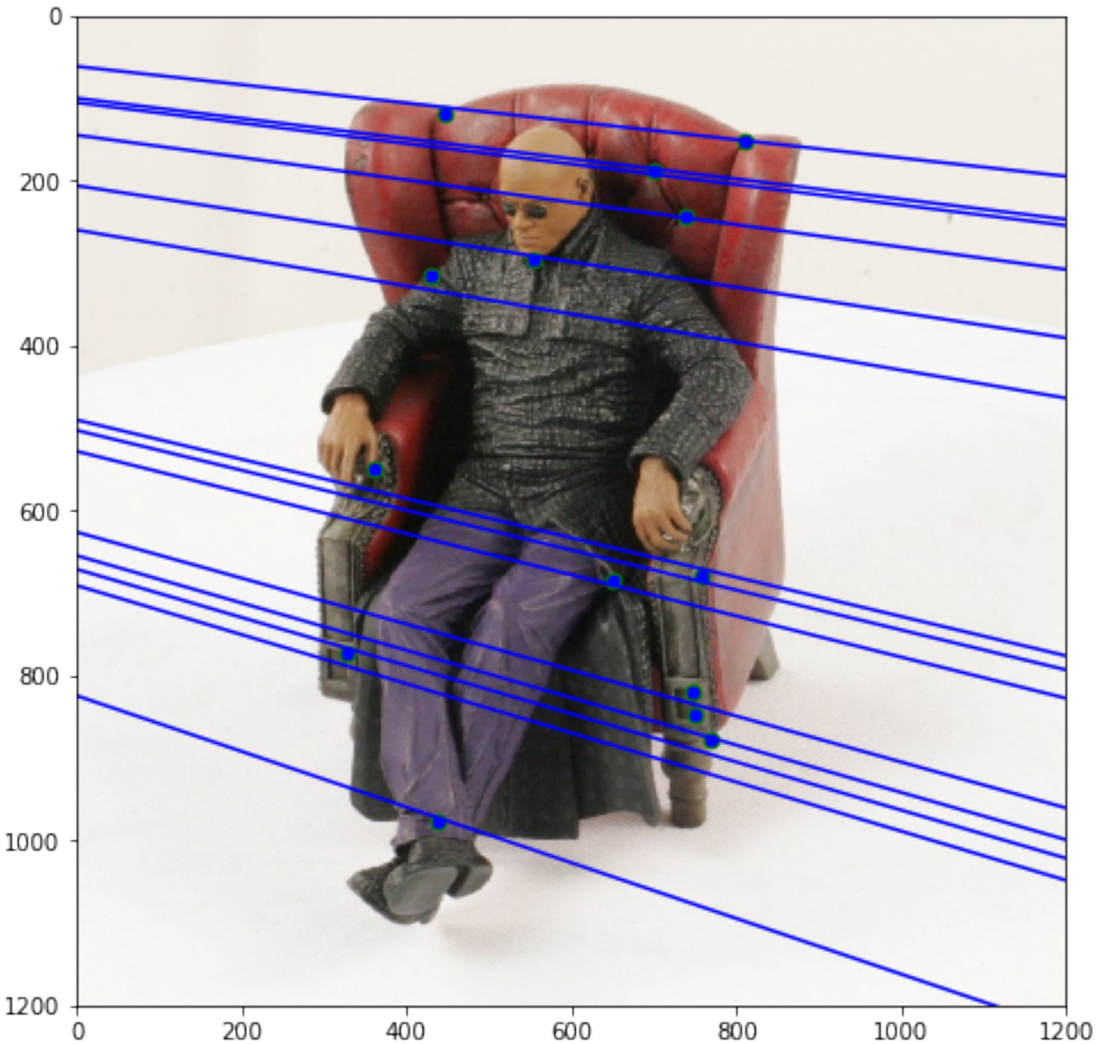
```
In [47]: def plot_epipolar_lines(img1,img2, cor1, cor2):
        """Plot epipolar lines on image given image, corners

        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)

        """

        """
```

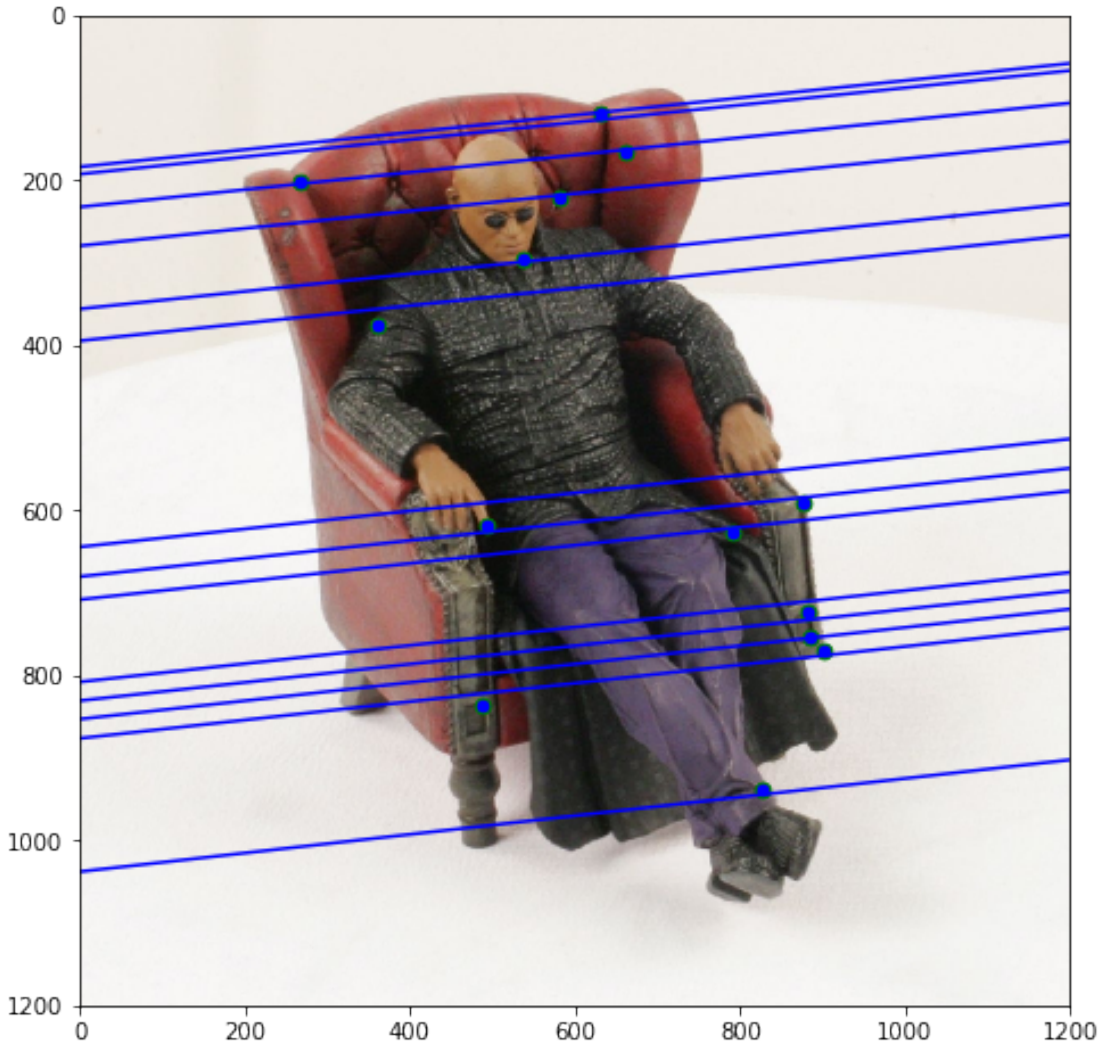
```
    Your code here:
    """
    F = fundamental_matrix(cor1,cor2)
    lines1 = np.dot(F,cor2)
    lines2 = np.dot(F.T,cor1)

    fig = plt.figure(figsize=(8, 8))
    plt.imshow(img1, cmap = 'gray')
#     for i in range(cor2.shape[1]):
    x0 = 0
    y0 = -lines1[2]/lines1[1]

    x1 = img1.shape[1]
    y1 = -lines1[0]/lines1[1]*x1 - lines1[2]/lines1[1]

    plt.plot([x0,x1],[y0,y1],color='b')
    plt.axis([0,img1.shape[1],img1.shape[0],0])
    plt.scatter(cor1[0,:],cor1[1,:],s=45,edgecolors='g',facecolors='b')
    plt.show()



    fig = plt.figure(figsize=(8, 8))
    plt.imshow(img2, cmap = 'gray')
#     for i in range(cor1.shape[1]):
    x0 = 0
    y0 = -lines2[2]/lines2[1]

    x1 = img1.shape[1]
    y1 = -lines2[0]/lines2[1]*x1 - lines2[2]/lines2[1]

    plt.plot([x0,x1],[y0,y1],color='b')
    plt.axis([0,img2.shape[1],img2.shape[0],0])
    plt.scatter(cor2[0,:],cor2[1,:],s=45,edgecolors='g',facecolors='b')
    plt.show()
```

In [164]:
```
# replace images and corners with those of matrix and warrior
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)
```

```
/usr/local/lib/python3.7/site-packages/ipykernel_launcher.py:2: DeprecationWarning: `imread` i
```

```
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.

/usr/local/lib/python3.7/site-packages/ipykernel_launcher.py:3: DeprecationWarning: `imread` is
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  This is separate from the ipykernel package so we can avoid doing imports until
```

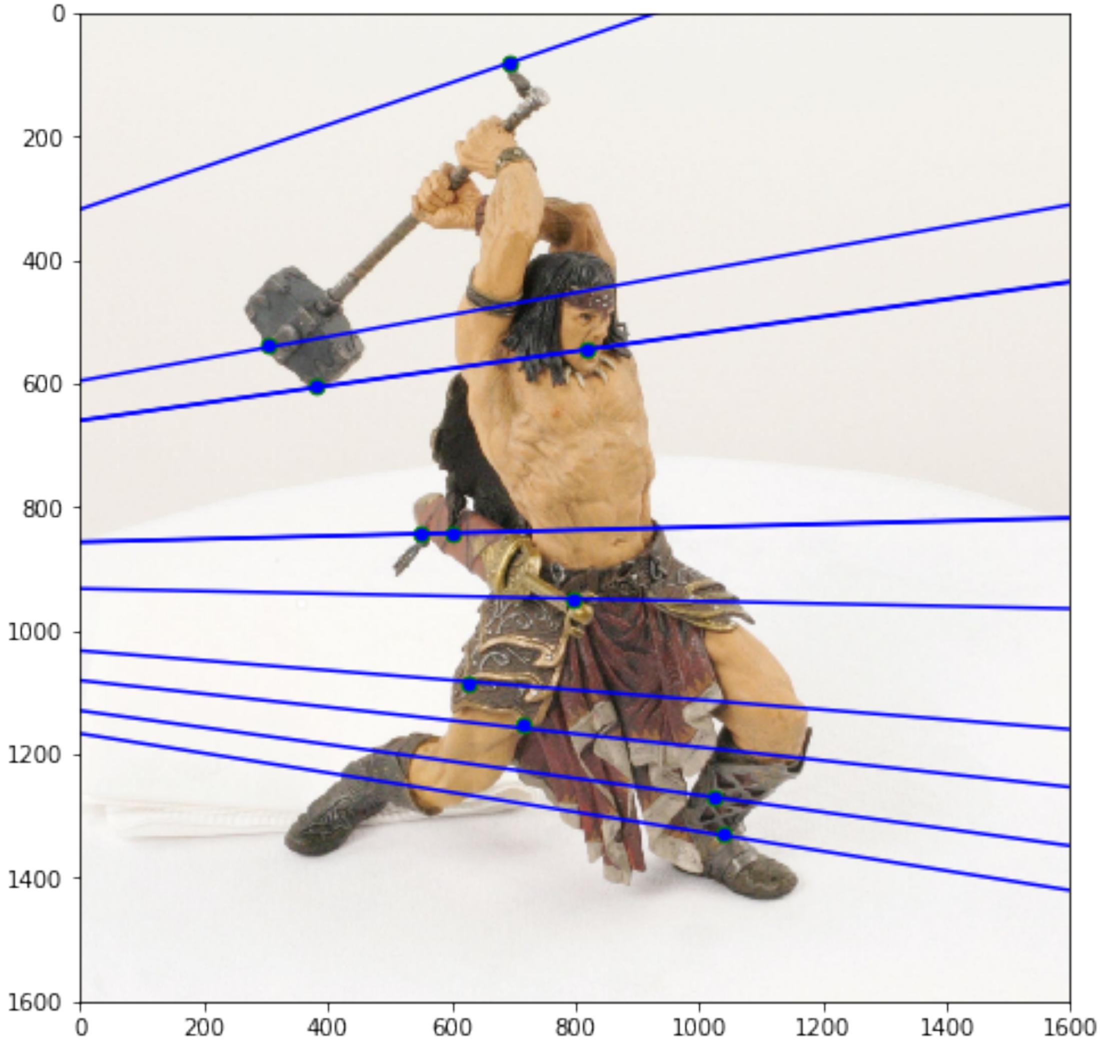


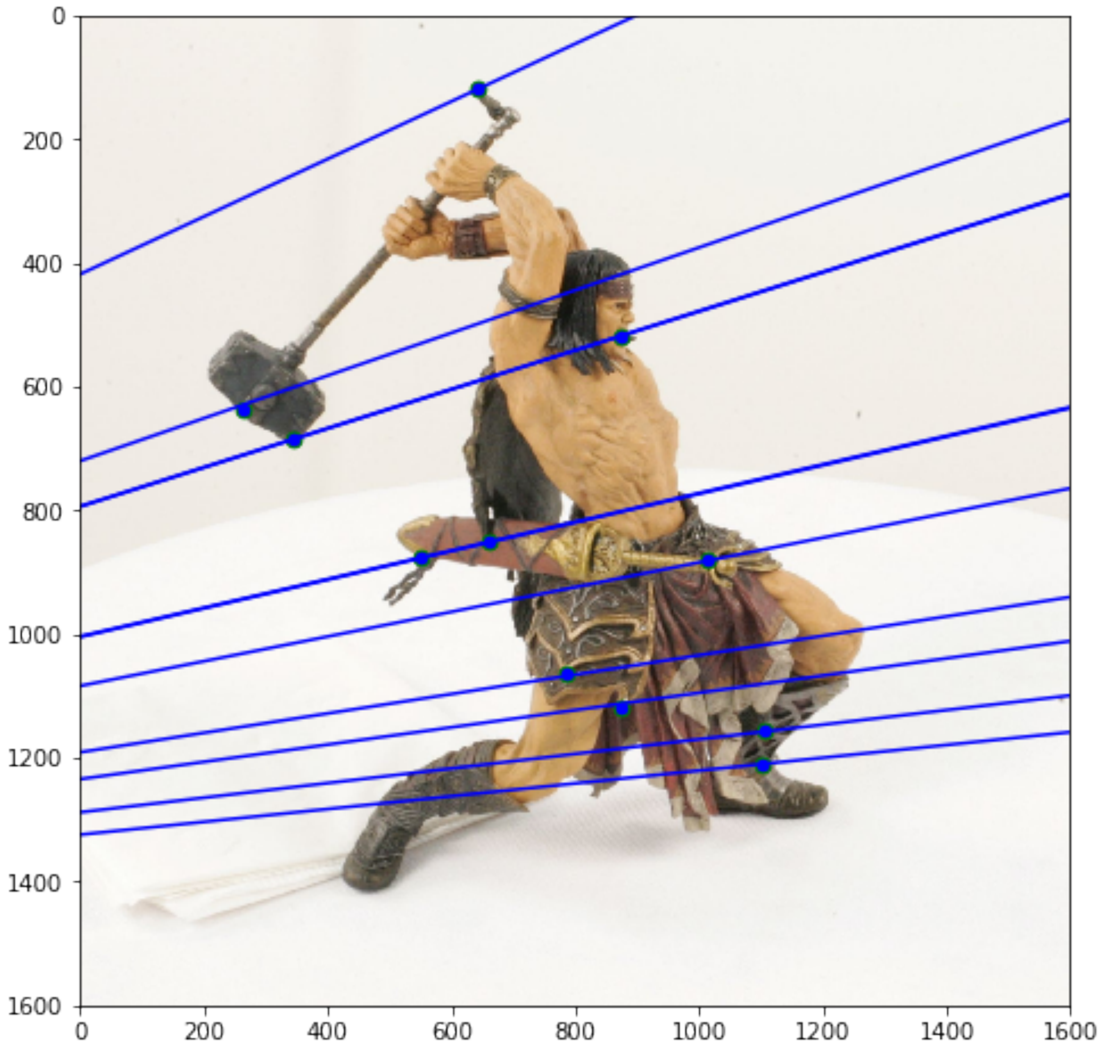



```
In [194]: # replace images and corners with those of matrix and warrior
          I1 = imread("./p4/warrior/warrior0.png")
          I2 = imread("./p4/warrior/warrior1.png")

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

          plot_epipolar_lines(I1,I2,cor1,cor2)
```

```
/usr/local/lib/python3.7/site-packages/ipykernel_launcher.py:2: DeprecationWarning: `imread` is
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.

/usr/local/lib/python3.7/site-packages/ipykernel_launcher.py:3: DeprecationWarning: `imread` is
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
```

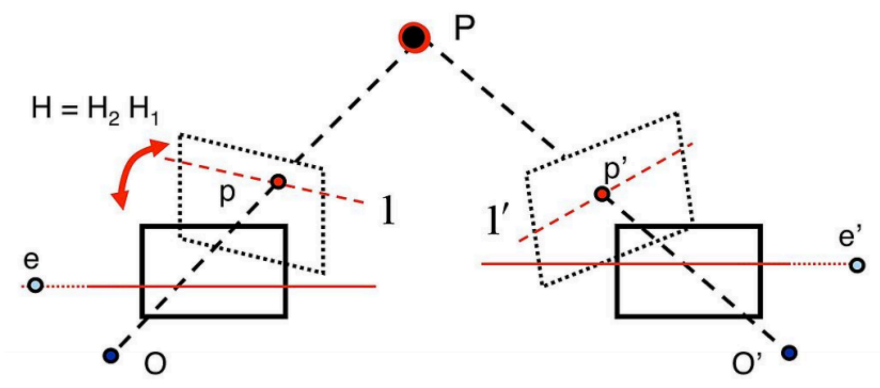
```
Use ``imageio.imread`` instead.
  This is separate from the ipykernel package so we can avoid doing imports until
```



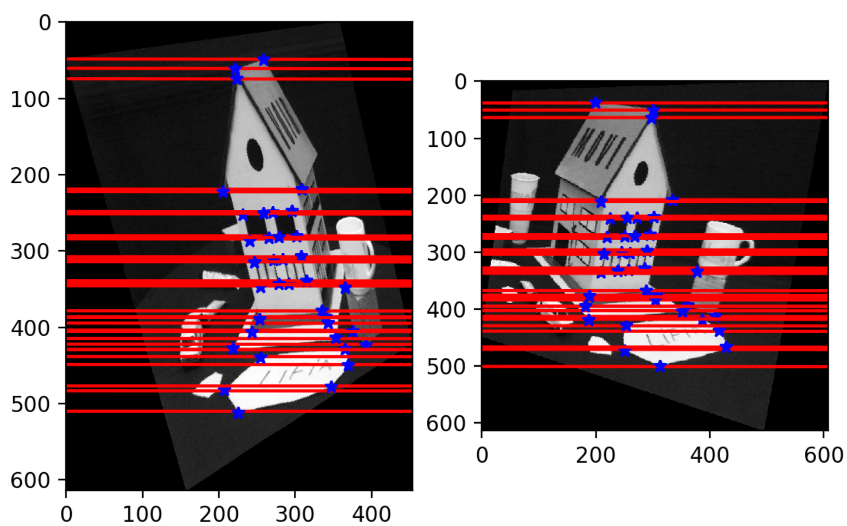


1.5.5 Image Rectification [3 pts]

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 $\mathtt{E} = [\boldsymbol{T_x}]\boldsymbol{R} = [\boldsymbol{T_x}]$. 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.
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:



```
In [177]: 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
points2_hat = H2.dot(points2).T
```

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

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

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

```
In [182]: # convert points from euclidian to homogeneous
    def to_homog(points):
        # write your code here
        points_homog = np.vstack((points, np.ones((1, points.shape[1]))))
        return points_homog


    # convert points from homogeneous to euclidian
    def from_homog(points_homog):
        # write your code here
        points = points_homog[:-1, :] / points_homog[-1, :]
        return points

    def warp2(source_img, target_size, offsetHW, H):
        # Create a target image and select target points to create a homography from tar
        # in other words map each target point to a source point, and then create a warp
        # of the image based on the homography by filling in the target image.
        # Make sure the new image (of size target_size) has the same number of color cha

        x_coords = [0, source_img.shape[1], source_img.shape[1], 0]
        y_coords = [0, 0, source_img.shape[0], source_img.shape[0]]
        source_points = np.vstack((x_coords, y_coords))

        target_region = from_homog(np.dot(H, to_homog(source_points))).astype(int)
        target_region[0, :] = target_region[0, :] + offsetHW[1]
        target_region[1, :] = target_region[1, :] + offsetHW[0]


        target_mask = np.ones(target_size)

        cv2.fillConvexPoly(target_mask, target_region.T, 1)
        target_mask = target_mask.astype(np.bool)
        row_y = np.linspace(0, target_size[0]-1, target_size[0]) # 161
        col_x = np.linspace(0, target_size[1]-1, target_size[1]) # 214
        col_idx, row_idx = np.meshgrid(col_x, row_y) # row 160, column 214

        target_mask = np.vstack((col_idx[target_mask], row_idx[target_mask])).astype(int
```

```python
    origin_target_mask = np.vstack((target_mask[0, :] - offsetHW[1], target_mask[1,
    inv_H = np.linalg.inv(H)

    source_mask = from_homog(np.dot(inv_H, to_homog(origin_target_mask)))


    mask_col = (source_mask[0, :]<source_img.shape[1]) * (source_mask[0, :] >= 0)
    mask_row = (source_mask[1, :]<source_img.shape[0]) * (source_mask[1, :] >= 0)
    mask = mask_row * mask_col


    target_mask_idx = target_mask[:, mask].astype(int)
    source_mask_idx = source_mask[:, mask].astype(int)

    target_img = np.ones(np.array(target_size).astype(int))

    target_img[target_mask_idx.T[:,1], target_mask_idx.T[:,0]] \
    = source_img[source_mask_idx.T[:,1], source_mask_idx.T[:,0]]

    return target_img


def getImageSize(im_maxHW, im_minHW):
    # consider H
    offsetH = - im_minHW[0]
    offsetW = - im_minHW[1]
    if im_minHW[0] < 0:
        height = im_maxHW[0] - im_minHW[0]

    else:
        height = im_maxHW[0]


    if im_minHW[1] < 0:
        width = im_maxHW[1] - im_minHW[1]
    else:
        width = im_maxHW[1]
    return (height, width), (offsetH, offsetW)



def new_Wx_Hy(Wx, Hy, H):
    x_coords = [0,Wx,0,Wx]
    y_coords = [0,0,Hy,Hy]
    xy_coords = np.vstack((x_coords, y_coords))
    homog_xy_coords = to_homog(xy_coords)
```

```
        new_xy_coords = from_homog(H.dot(homog_xy_coords))

        minH = np.floor(np.min(new_xy_coords[1]))
        minW = np.floor(np.min(new_xy_coords[0]))

        maxH = np.ceil(np.max(new_xy_coords[1]))
        maxW = np.ceil(np.max(new_xy_coords[1]))
        return (maxH, maxW), (minH, minW)
```

```
In [183]: def image_rectification(im1,im2,points1,points2):
        '''this function provides the rectified images along with the new corner points o
        images with corner correspondences
        input:
        im1--> image1
        im2--> image2
        points1--> corner points in image1 3*n
        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
        '''
        "your code here"


        F = fundamental_matrix(points1, points2)
        e1, e2 = compute_epipole(F)
        H1, H2 = compute_matching_homographies(e2, F, im2, points1, points2)

        im1_maxHW, im1_minHW = new_Wx_Hy(im1.shape[1], im1.shape[0], H1)
        im1_sizeHW, im1_offsetHW = getImageSize(im1_maxHW, im1_minHW) # height and width

        im2_maxHW, im2_minHW = new_Wx_Hy(im2.shape[1], im2.shape[0], H2)
        im2_sizeHW, im2_offsetHW = getImageSize(im2_maxHW, im2_minHW) # height and width


        img_sizeHW = (int(max(im2_sizeHW[0], im1_sizeHW[0])), int(max(im2_sizeHW[1], im1_

        rectified_im1 = warp2(im1, img_sizeHW, im1_offsetHW, H1)
        rectified_im2 = warp2(im2, img_sizeHW, im2_offsetHW, H2)


        new_cor1 = H1.dot(points1)/ H1.dot(points1)[-1] # 3* N
        new_cor1[0, :] = new_cor1[0, :] + im1_offsetHW[1]
        new_cor1[1, :] = new_cor1[1, :] + im1_offsetHW[0]
```

```
        new_cor2 = H2.dot(points2)/ H2.dot(points2)[-1]
        new_cor2[0, :] = new_cor2[0, :] + im2_offsetHW[1]
        new_cor2[1, :] = new_cor2[1, :] + im2_offsetHW[0]

        return rectified_im1, rectified_im2, new_cor1, new_cor2
```

1.5.6 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.

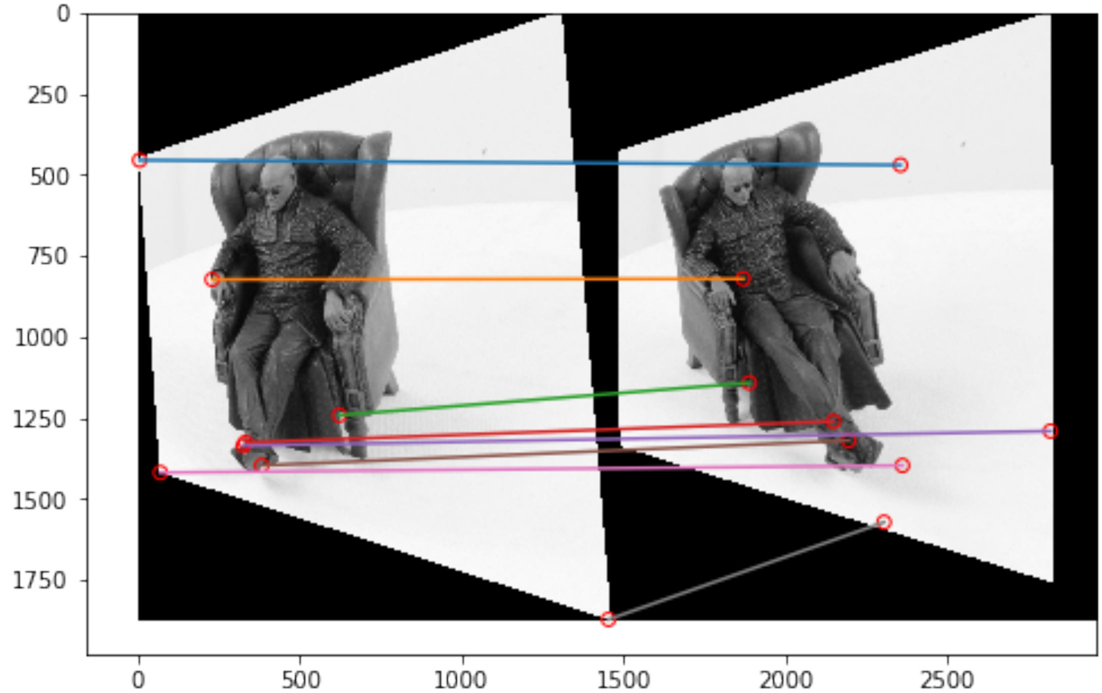
```
In [192]: def display_correspondence(img1, img2, corrs):
              """Plot matching result on image pair given images and correspondences

              Args:
                  img1: Image 1.
                  img2: Image 2.
                  corrs: Corner correspondence

              """

              """
              Your code here.
              You may refer to the show_matching_result function
              """
          #     cor1 = to_homog(np.array([i[0] for i in corrs]).T)
          #     cor2 = to_homog(np.array([i[1] for i in corrs]).T)

              fig = plt.figure(figsize=(8, 8))
              plt.imshow(np.hstack((img1, img2)), cmap='gray') # two dino images are of differ
              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', facecolors='
                  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.
```

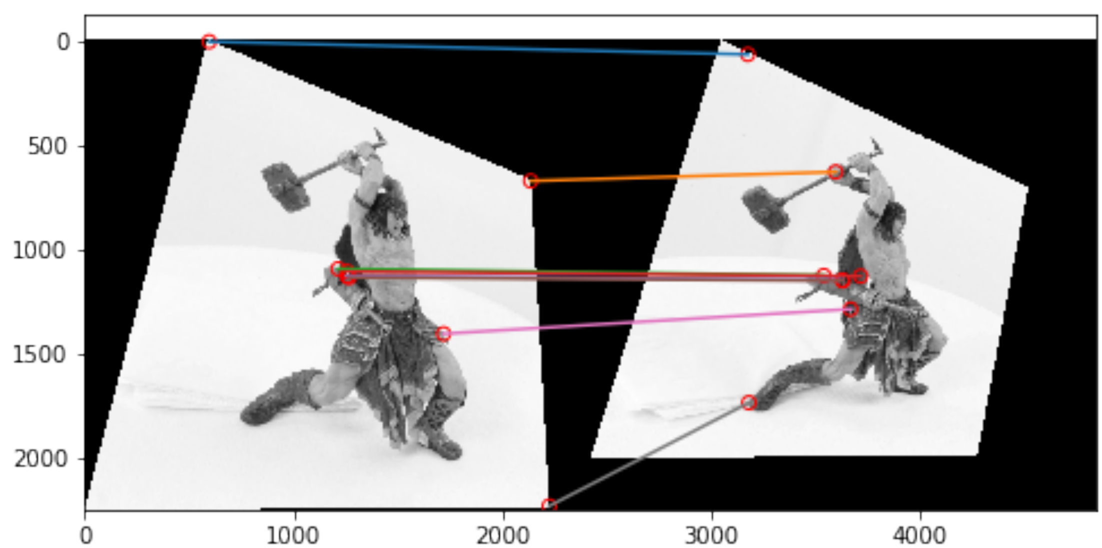
```
            corners1: Detected corners in image 1.
            F: Fundamental matrix calculated using given ground truth corner correspondes
            R: NCC matching window radius.
            NCCth: NCC matching threshold.

        Returns:
            Matching result to be used in display_correspondence function

        """
        """
        Your code here.
        """

        matching = []
        num_corners1 = corners1.shape[0] # n * 2

        corners1_homog = np.hstack((corners1, np.ones(num_corners1)[:, None]))
        lines2 = np.dot(F.T, corners1_homog.T) # corners1 epipolar line in image2, lines

        # c1 is the center of corner 1
        # c2 is the center of corner 2
        # both in non homogeneous
        for i, c1 in enumerate(corners1):
            maximum = -100000000

            for x_j in range(R, img2.shape[1] - R):

                y_j = int(-lines2[0][i]/lines2[1][i]* x_j - lines2[2][i]/lines2[1][i])
                if y_j >= R and y_j < img2.shape[0]-R:
                    temp_c2 = np.array([x_j, y_j]) # non homogeneous
                    ncc_score = ncc_match(img1, img2, c1, temp_c2, R)
                    if ncc_score > maximum:
                        maximum = ncc_score
                        best_c2 = temp_c2

            if maximum > NCCth:
                matching.append((c1, best_c2))

        return matching
```

In [193]:
```
nCorners = 10 # number of corners want to extract
#decide the NCC matching window radius
R = 5
NCCth = 0.7

I1=rgb2gray(imread("./p4/matrix/matrix0.png"))
I2=rgb2gray(imread("./p4/matrix/matrix1.png"))
cor1 = np.load("./p4/dino/cor1.npy")
```

```
cor2 = np.load("./p4/dino/cor2.npy")
I3=rgb2gray(imread("./p4/warrior/warrior0.png"))
I4=rgb2gray(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,co
rectified_im3, rectified_im4, new_cor3, new_cor4 = image_rectification(I3,I4,cor3,co

F_new = fundamental_matrix(new_cor1, new_cor2)
# detect corners using corner detector here, store in corners1
corners1 = corner_detect(rectified_im1, nCorners, smoothSTD, windowSize)
corrs = correspondence_matching_epipole(rectified_im1, rectified_im2, corners1, F_ne
display_correspondence(rectified_im1, rectified_im2, corrs)

F_new2 = fundamental_matrix(new_cor3, new_cor4)
# detect corners using corner detector here, store in corners2
corners2 = corner_detect(rectified_im3, nCorners, smoothSTD, windowSize)
corrs = correspondence_matching_epipole(rectified_im3, rectified_im4, corners2, F_ne
display_correspondence(rectified_im3, rectified_im4, corrs)
```

```
/usr/local/lib/python3.7/site-packages/ipykernel_launcher.py:6: DeprecationWarning: `imread` i
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.

/usr/local/lib/python3.7/site-packages/ipykernel_launcher.py:7: DeprecationWarning: `imread` i
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  import sys
/usr/local/lib/python3.7/site-packages/ipykernel_launcher.py:10: DeprecationWarning: `imread`
`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.
/usr/local/lib/python3.7/site-packages/ipykernel_launcher.py:11: DeprecationWarning: `imread`
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  # This is added back by InteractiveShellApp.init_path()
/usr/local/lib/python3.7/site-packages/ipykernel_launcher.py:61: FutureWarning: `rcond` parame
To use the future default and silence this warning we advise to pass `rcond=None`, to keep usi
/usr/local/lib/python3.7/site-packages/ipykernel_launcher.py:29: RuntimeWarning: invalid value
```


0
250
500
750
1000
1250
1500
1750
0
500
1000
1500
2000
2500


0
500
1000
1500
2000
0
1000
2000
3000
4000