Computer Vision - Sparse Stereo Matching

0.0.1 Yifan Xu

0.0.2 November 20, 2018

0.1 Sparse Stereo Matching

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).

0.1.1 Corner Detection

The first thing we need to do is to build a corner detector. 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

```
In [3]: import numpy as np
        import imageio
        from scipy.misc import imread
        import matplotlib.pyplot as plt
        from scipy.ndimage.filters import gaussian_filter
        from scipy import signal
        import operator
        from scipy import ndimage
        import math
        from scipy.ndimage.filters import maximum_filter
        import warnings
        warnings.filterwarnings('ignore')
```

```
In [4]: def rgb2gray(rgb):
            """ Convert rgb image to grayscale.
```

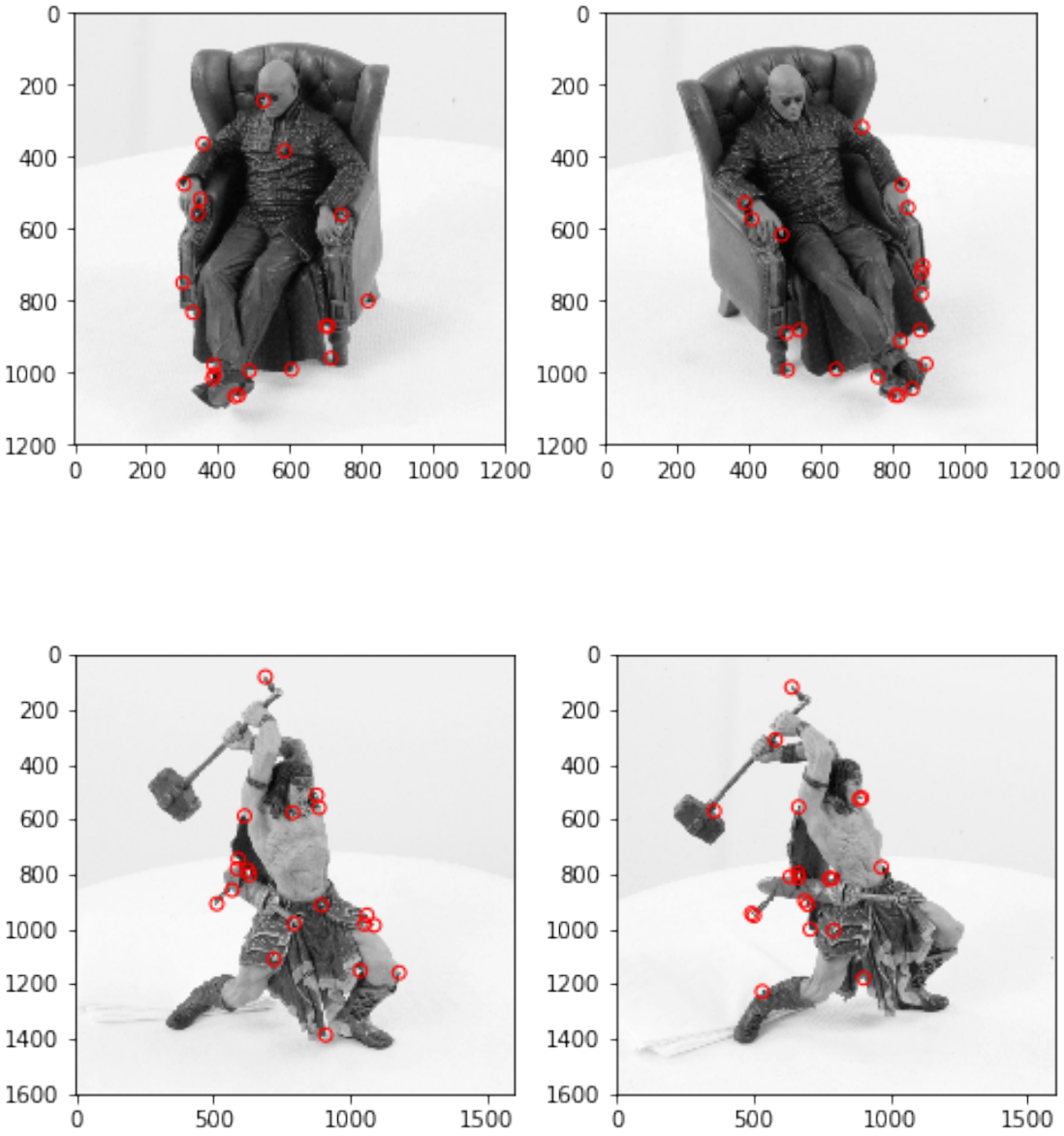
```
            """
            return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])

In [3]: 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:
            """

            h,w = image.shape[0], image.shape[1]
            # filter image with gaussian
            img_smooth = gaussian_filter(image, sigma=smoothSTD) # compute gradient everywhere
            kernal = np.ones((windowSize,windowSize))

            dx = np.array([[1,0,-1],[2,0,-2],[1,0,-1]])
            dy = np.array([[1,2,1] ,[0,0,0], [-1,-2,-1]])
            Ix = ndimage.convolve(img_smooth,dx)
            Iy = ndimage.convolve(img_smooth,dy)
            Ixx = signal.convolve2d(Ix*Ix, kernal, boundary='symm', mode='same')
            Iyy = signal.convolve2d(Iy*Iy, kernal, boundary='symm', mode='same')
            Ixy = signal.convolve2d(Ix*Iy, kernal, boundary='symm', mode='same')

            detC = Ixx*Iyy - Ixy*Ixy
            traceC = Ixx + Iyy
            e1 = 0.5 * (traceC + np.sqrt(traceC * traceC - 4 * detC))
            e2 = 0.5 * (traceC - np.sqrt(traceC * traceC - 4 * detC))
            e = np.minimum(e1,e2)

            locol_max = maximum_filter(e, size=windowSize)
            half = int(windowSize / 2)
            S = {}
            for i in range(half,h-half):
                for j in range(half,w-half):
                    locol = e[i-half:i+half+1,j-half:j+half+1]
                    locol_max = locol.max()
                    if e[i,j] == locol_max:
```

```
                    S[(j,i)] = e[i,j]

        sortedS = sorted(S.items(), key=operator.itemgetter(1), reverse=True)
        sortedS = sortedS[:nCorners]
        corners = []
        for pix in sortedS:
            corners.append(pix[0][0])
            corners.append(pix[0][1])
        corners = np.array(corners).reshape((nCorners,2))

        return corners
```

In [4]:
```
# 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 =  imageio.imread('p4/matrix/matrix' + str(i) + '.png')
    imgs_mat.append(rgb2gray(img_mat))
    # downsize your image in case corner_detect runs slow in test
    # imgs_mat.append(rgb2gray(img_mat)[::2, ::2])
    crns_mat.append(corner_detect(imgs_mat[i], nCorners, smoothSTD, windowSize))

    img_war = imageio.imread('p4/warrior/warrior' + str(i) + '.png')
    imgs_war.append(rgb2gray(img_war))
    # 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))
```

In [5]:
```
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='none')

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

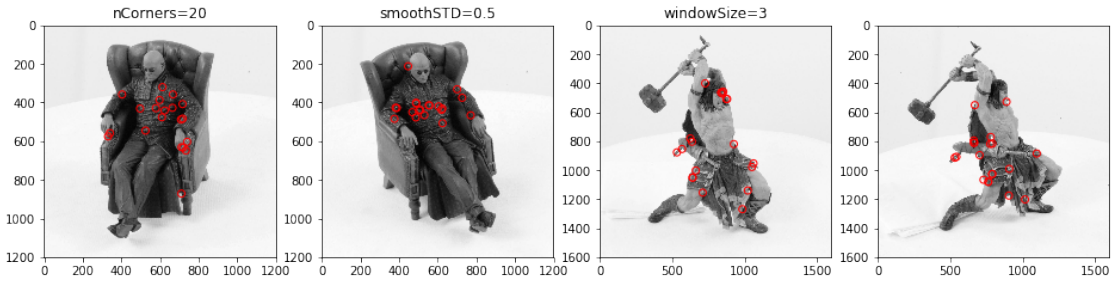
```
show_corners_result(imgs_mat, crns_mat)
show_corners_result(imgs_war, crns_war)
```

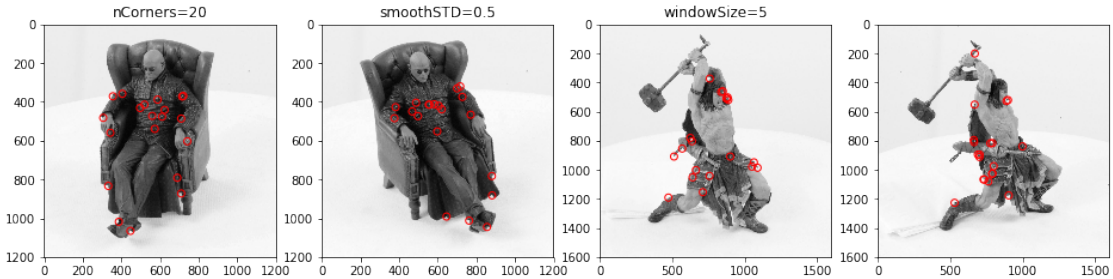


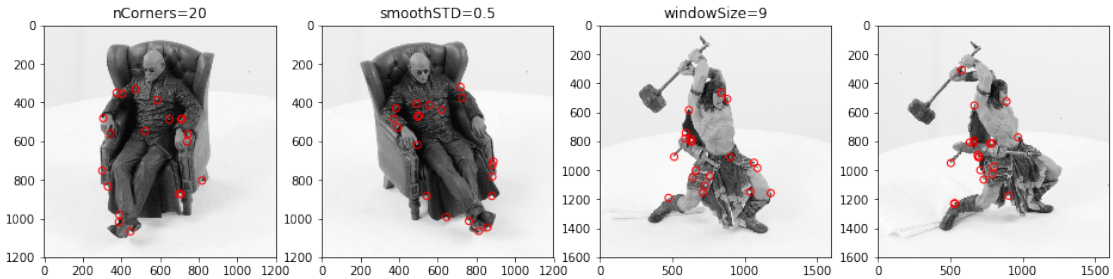
In [6]:
```
###### try different parameters:
def different_parameters(nCorners,smoothSTD,windowSize):
    # read images and detect corners on images
    imgs_mat = []
    crns_mat = []
    imgs_war = []
    crns_war = []
```

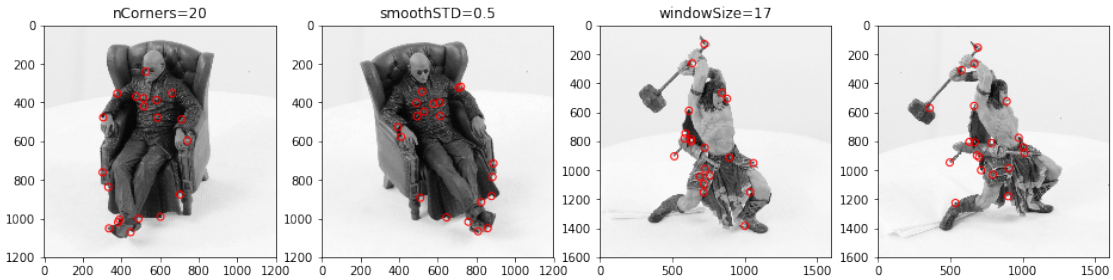
```
    for i in range(2):
        img_mat =  imageio.imread('p4/matrix/matrix' + str(i) + '.png')
        imgs_mat.append(rgb2gray(img_mat))
        crns_mat.append(corner_detect(imgs_mat[i], nCorners, smoothSTD, windowSize))

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

    plt.figure(figsize=(16, 16))
    plt.subplot(141)
    plt.imshow(imgs_mat[0], cmap='gray')
    plt.scatter(crns_mat[0][:, 0], crns_mat[0][:, 1], s=35, edgecolors='r', \
                facecolors='none')
    plt.title("nCorners={}".format(nCorners))
    plt.subplot(142)
    plt.imshow(imgs_mat[1], cmap='gray')
    plt.scatter(crns_mat[1][:, 0], crns_mat[1][:, 1], s=35, edgecolors='r', \
                facecolors='none')
    plt.title("smoothSTD={}".format(smoothSTD))
    plt.subplot(143)
    plt.imshow(imgs_war[0], cmap='gray')
    plt.scatter(crns_war[0][:, 0], crns_war[0][:, 1], s=35, edgecolors='r', \
                facecolors='none')
    plt.title("windowSize={}".format(windowSize))
    plt.subplot(144)
    plt.imshow(imgs_war[1], cmap='gray')
    plt.scatter(crns_war[1][:, 0], crns_war[1][:, 1], s=35, edgecolors='r', \
                facecolors='none')

    plt.show()
```

In [7]:
```
different_parameters(nCorners=20,smoothSTD=0.5,windowSize=3)
different_parameters(nCorners=20,smoothSTD=0.5,windowSize=5)
different_parameters(nCorners=20,smoothSTD=0.5,windowSize=9)
different_parameters(nCorners=20,smoothSTD=0.5,windowSize=17)
```

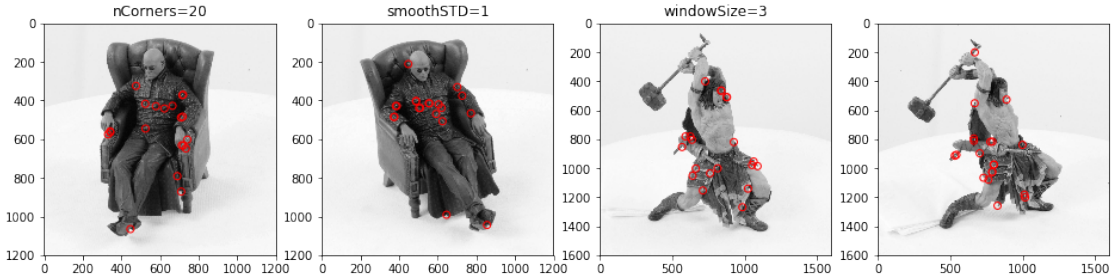


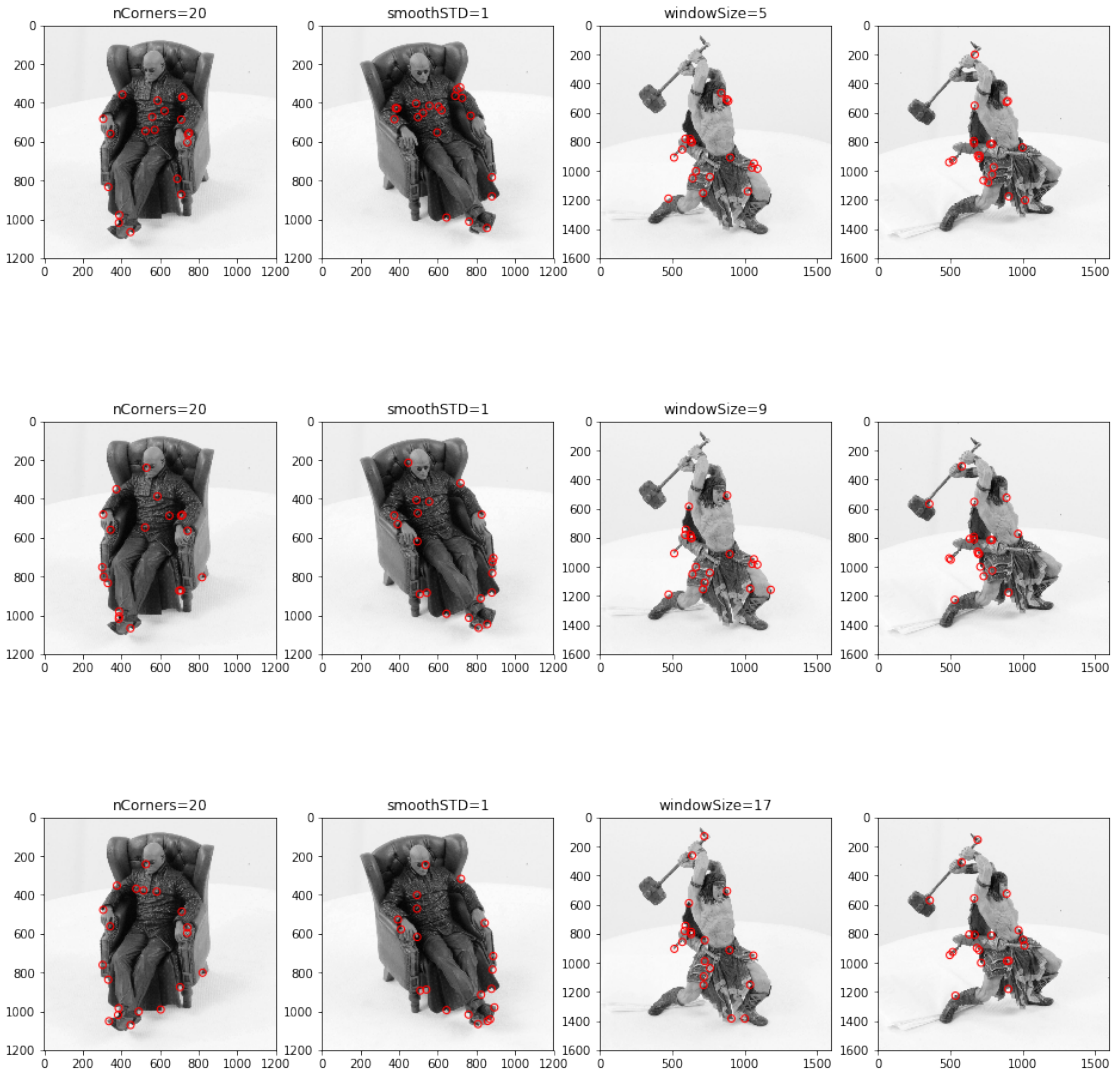




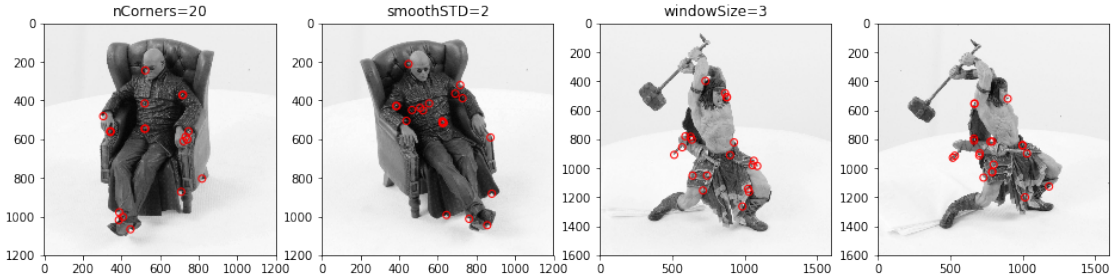


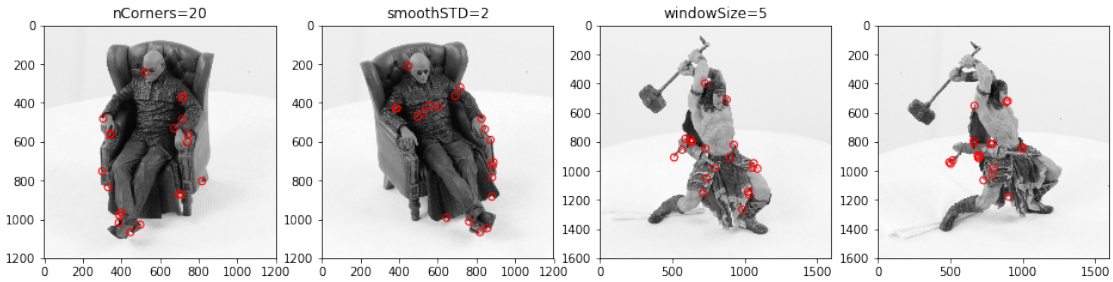
In [8]:
```
different_parameters(nCorners=20,smoothSTD=1,windowSize=3)
different_parameters(nCorners=20,smoothSTD=1,windowSize=5)
different_parameters(nCorners=20,smoothSTD=1,windowSize=9)
different_parameters(nCorners=20,smoothSTD=1,windowSize=17)
```

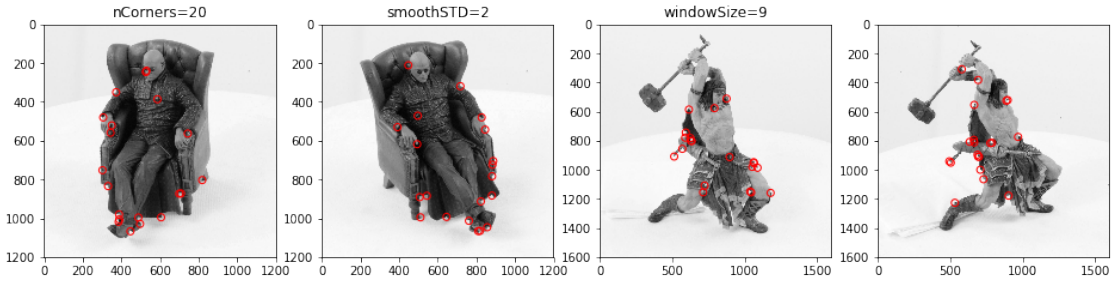


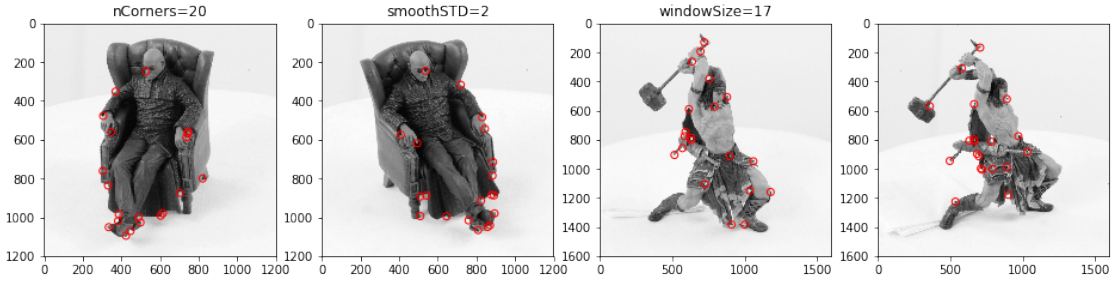


In [9]:
```
different_parameters(nCorners=20,smoothSTD=2,windowSize=3)
different_parameters(nCorners=20,smoothSTD=2,windowSize=5)
different_parameters(nCorners=20,smoothSTD=2,windowSize=9)
different_parameters(nCorners=20,smoothSTD=2,windowSize=17)
```

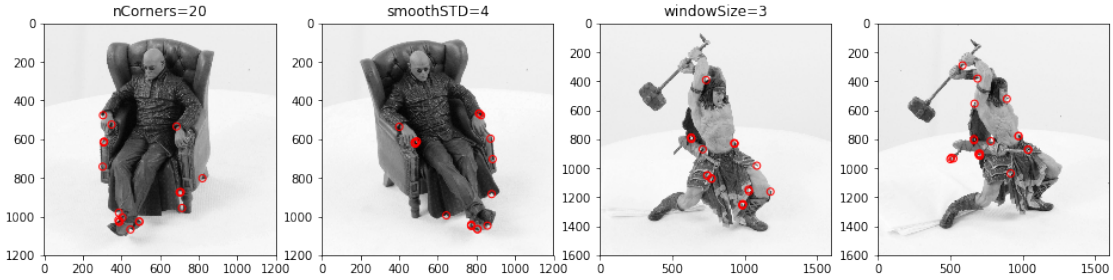


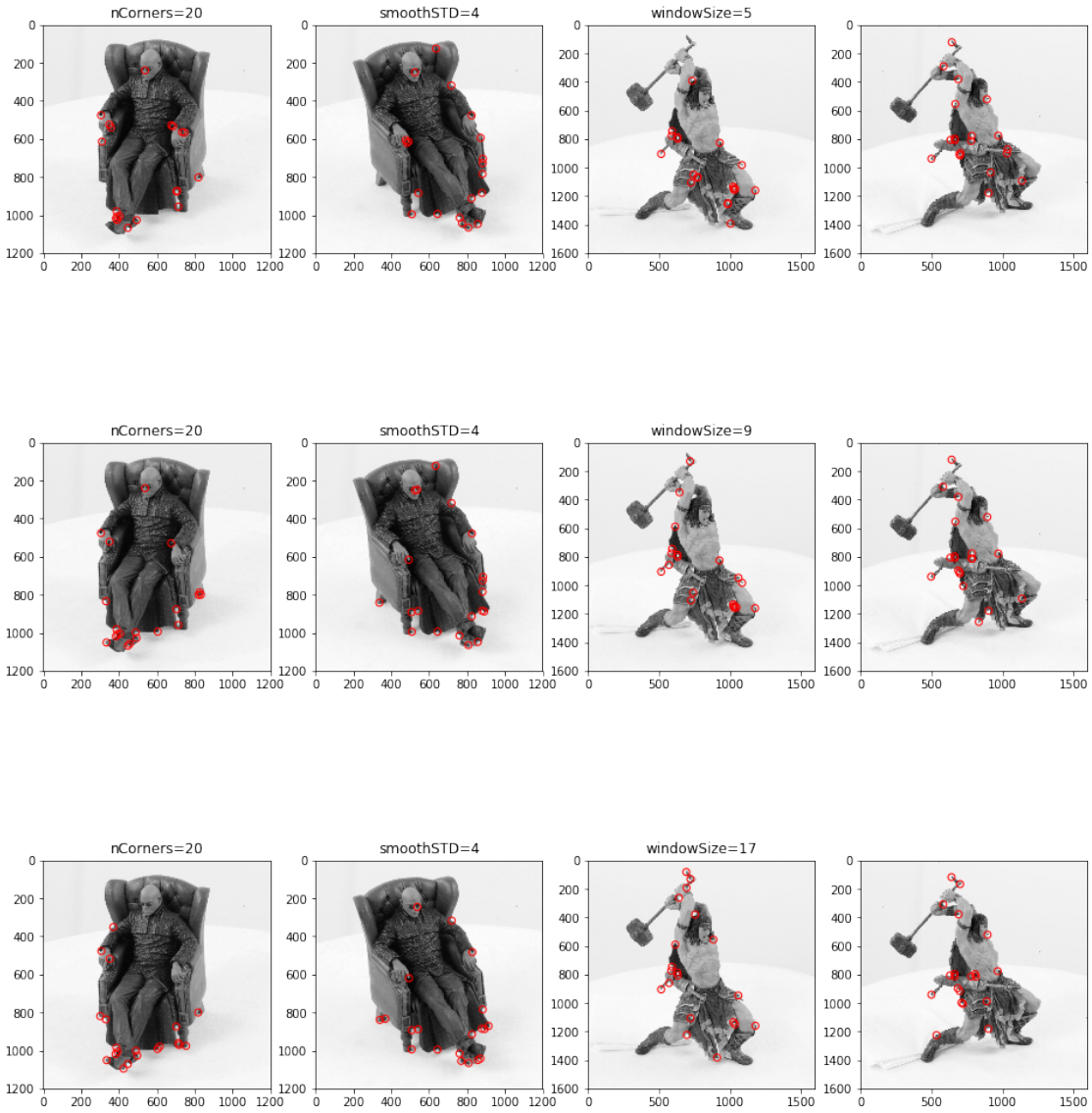






In [10]:
```
different_parameters(nCorners=20,smoothSTD=4,windowSize=3)
different_parameters(nCorners=20,smoothSTD=4,windowSize=5)
different_parameters(nCorners=20,smoothSTD=4,windowSize=9)
different_parameters(nCorners=20,smoothSTD=4,windowSize=17)
```





1. windowSize = 3, 5, 9, 17 (Window size for corner detector and non maximum suppression.) From the figures above, we can see when the windowSize increase, it's more likely to successfully detect corners than edges. That's because when the window size is bigger, it will find a local-maximum in a bigger area which is more likely to be a corner.

2. smoothSTD = 0.5, 1, 2, 4 (Standard deviation of the Gaussian smoothing kernel.) From the figures above, we can see when the smoothSTD is smaller, it's hard to successfully detect real corners. That's because when the standard deviation of the Gaussian smoothing kernel is small, it can't blur the pictures enough.

0.1.2 NCC (Normalized Cross-Correlation) Matching

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 [11]: 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:
             """
             matching_score = 0
             c1x = c1[1]
             c1y = c1[0]
             c2x = c2[1]
             c2y = c2[0]
             S1 = img1[c1x-R:c1x+R+1,c1y-R:c1y+R+1]
             S2 = img2[c2x-R:c2x+R+1,c2y-R:c2y+R+1]
             mean1 = S1.mean()
             mean2 = S2.mean()
             sum1 = ((S1-mean1)**2).sum()
             sum2 = ((S2-mean2)**2).sum()
             for i in range(2*R+1):
                 for j in range(2*R+1):
                     W1w = (S1[i,j]-mean1)/np.sqrt(sum1)
                     W2w = (S2[i,j]-mean2)/np.sqrt(sum2)
                     matching_score += W1w * W2w

             return matching_score
```

```
In [12]: # 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
0.845761528217442
0.6258689611426175
```

0.1.3 Naive Matching

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 [13]: 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 = []
             matchScores = np.ndarray((len(corners1), len(corners2)))
             for i, c1 in enumerate(list(corners1)):
                 for j, c2 in enumerate(list(corners2)):
                     matchScores[i,j] = ncc_match(img1, img2, c1, c2, R)
```

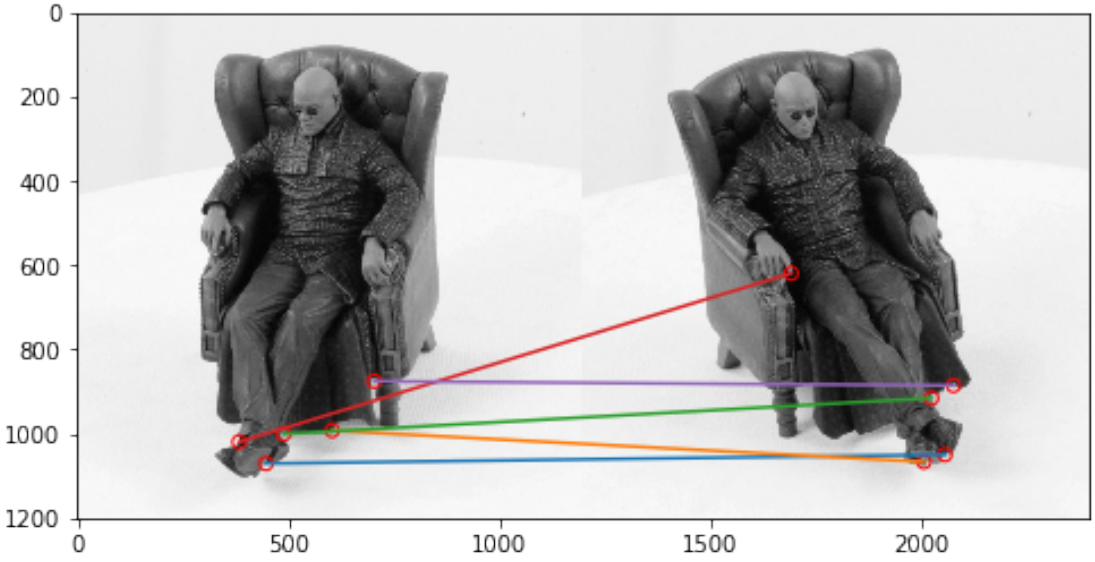
```
            for n in range(len(matchScores)):
                i,j = np.where(matchScores==np.max(matchScores))
                i,j = i[0],j[0]
                if matchScores[i,j] > NCCth:
                    matching.append((corners1[i],corners2[j]))
                matchScores[i,:]=matchScores[:,j]=0
            return matching
```

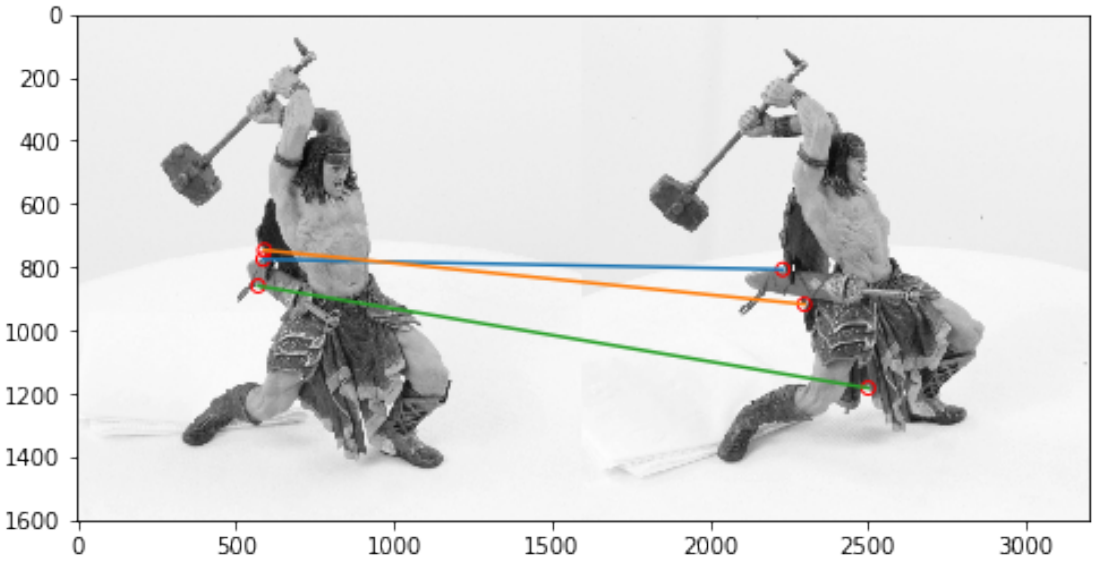
In [14]:
```
# detect corners on warrior and matrix sets
# adjust your corner detection parameters here
nCorners = 10
smoothSTD = 4
windowSize = 15

# read images and detect corners on images
imgs_mat = []
crns_mat = []
imgs_war = []
crns_war = []
for i in range(2):
    img_mat = imageio.imread('p4/matrix/matrix' + str(i) + '.png')
    imgs_mat.append(rgb2gray(img_mat))
    # downsize your image in case corner_detect runs slow in test
    # imgs_mat.append(rgb2gray(img_mat)[::2, ::2])
    crns_mat.append(corner_detect(imgs_mat[i], nCorners, smoothSTD, windowSize))

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

In [15]:
```
# match corners
R = 20
NCCth = 0.6
matching_mat = naive_matching(imgs_mat[0]/255, imgs_mat[1]/255, crns_mat[0], \
                              crns_mat[1], R, NCCth)
matching_war = naive_matching(imgs_war[0]/255, imgs_war[1]/255, crns_war[0], \
                              crns_war[1], R, NCCth)
```

In [17]:
```
# 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 differe
    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='n
        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)
```



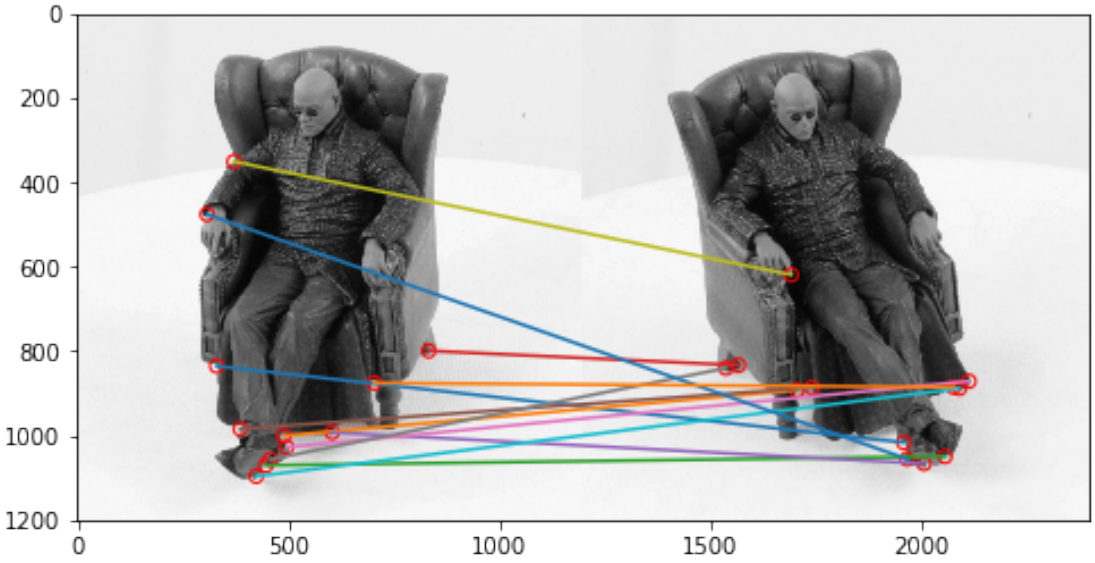


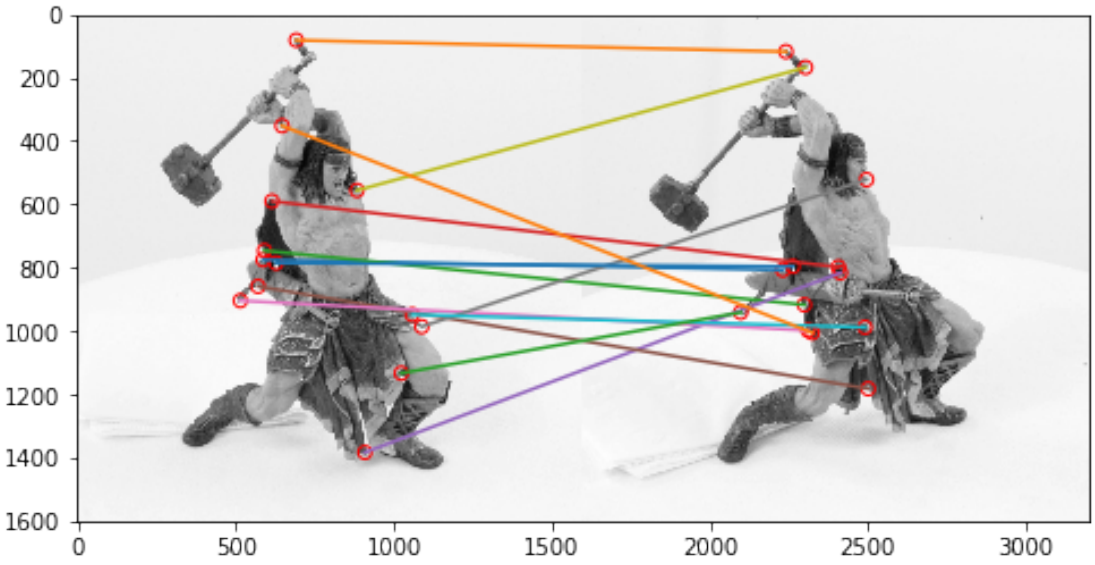
In [18]:
```
# detect corners on warrior and matrix sets
# adjust your corner detection parameters here
```

```
nCorners = 20
smoothSTD = 4
windowSize = 15

# read images and detect corners on images
imgs_mat = []
crns_mat = []
imgs_war = []
crns_war = []
for i in range(2):
    img_mat = imageio.imread('p4/matrix/matrix' + str(i) + '.png')
    imgs_mat.append(rgb2gray(img_mat))
    # downsize your image in case corner_detect runs slow in test
    # imgs_mat.append(rgb2gray(img_mat)[::2, ::2])
    crns_mat.append(corner_detect(imgs_mat[i], nCorners, smoothSTD, windowSize))

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

In [19]:
```
# match corners
R = 20
NCCth = 0.6
matching_mat = naive_matching(imgs_mat[0]/255, imgs_mat[1]/255, crns_mat[0], \
                              crns_mat[1], R, NCCth)
matching_war = naive_matching(imgs_war[0]/255, imgs_war[1]/255, crns_war[0], \
                              crns_war[1], R, NCCth)
```

In [20]:
```
# 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 differe
    for p1, p2 in matching:
        plt.scatter(p1[0], p1[1], s=35, edgecolors='r', facecolors='none')
        plt.scatter(p2[0] + img1.shape[1], p2[1], s=35, edgecolors='r', \
                    facecolors='none')
        plt.plot([p1[0], p2[0] + img1.shape[1]], [p1[1], p2[1]])
#     plt.savefig('dino_matching.png')
    plt.show()

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



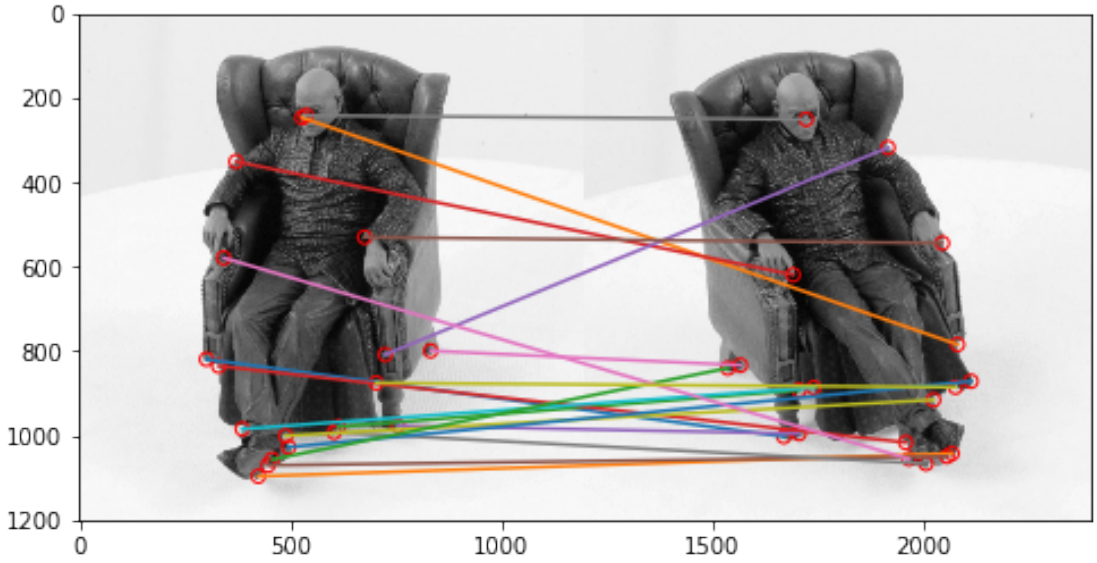


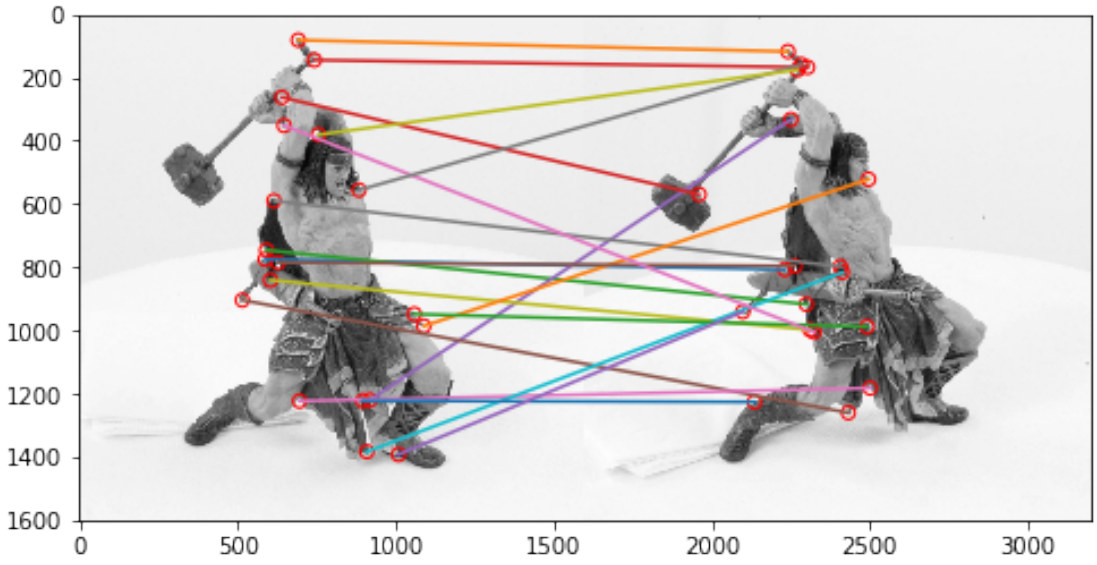
In [21]:
```
# detect corners on warrior and matrix sets
# adjust your corner detection parameters here
nCorners = 30
smoothSTD = 4
windowSize = 15

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

```
crns_mat = []
imgs_war = []
crns_war = []
for i in range(2):
    img_mat = imageio.imread('p4/matrix/matrix' + str(i) + '.png')
    imgs_mat.append(rgb2gray(img_mat))
    # downsize your image in case corner_detect runs slow in test
    # imgs_mat.append(rgb2gray(img_mat)[::2, ::2])
    crns_mat.append(corner_detect(imgs_mat[i], nCorners, smoothSTD, windowSize))

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

In [22]:
```
# match corners
R = 20
NCCth = 0.6
matching_mat = naive_matching(imgs_mat[0]/255, imgs_mat[1]/255, crns_mat[0], \
                              crns_mat[1], R, NCCth)
matching_war = naive_matching(imgs_war[0]/255, imgs_war[1]/255, crns_war[0], \
                              crns_war[1], R, NCCth)
```

In [23]:
```
# 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 differe
    for p1, p2 in matching:
        plt.scatter(p1[0], p1[1], s=35, edgecolors='r', facecolors='none')
        plt.scatter(p2[0] + img1.shape[1], p2[1], s=35, edgecolors='r', \
                    facecolors='none')
        plt.plot([p1[0], p2[0] + img1.shape[1]], [p1[1], p2[1]])
#     plt.savefig('dino_matching.png')
    plt.show()

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





0.1.4 Epipolar Geometry

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.

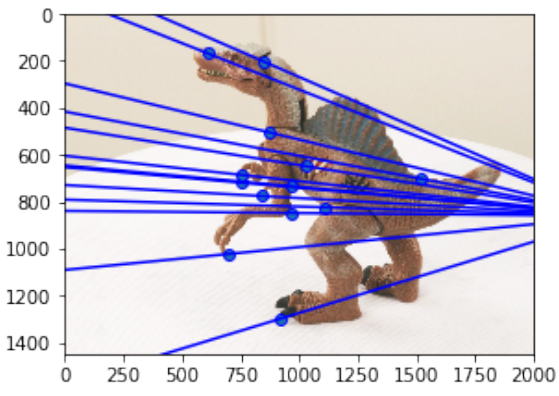
In [24]:
```
import numpy as np
from scipy.misc import imread
import matplotlib.pyplot as plt
from scipy.io import loadmat

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)

    # 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)
```

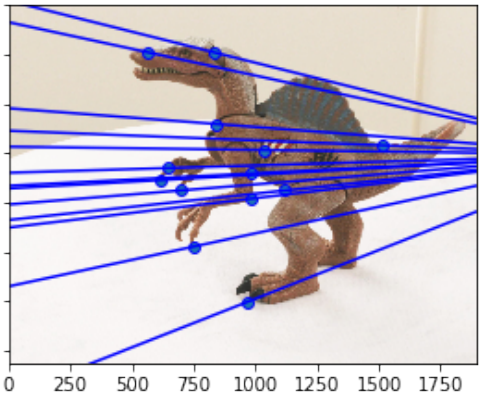
```
    x2 = x2 / x2[2]
    mean_2 = np.mean(x2[:2],axis=1)
    S2 = np.sqrt(2) / np.std(x2[:2])
    T2 = np.array([[S2,0,-S2*mean_2[0]],[0,S2,-S2*mean_2[1]],[0,0,1]])
    x2 = np.dot(T2,x2)

    # 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 corner poi
    input:
    F--> Fundamental matrix
    output:
    e1--> corresponding epipole in image 1
    e2--> epipole in image2
    '''
    #your code here
    U,S,V = np.linalg.svd(F.T)
    e1 = V.T[:,2]/V.T[:,2][-1]
    U,S,V = np.linalg.svd(F)
    e2 = V.T[:,2]/V.T[:,2][-1]
    return e1,e2
```

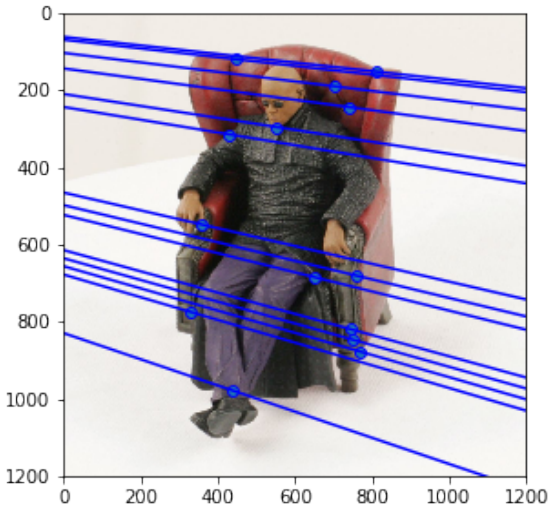
In [25]:
```
def plot_epipolar_lines(img1,img2, cor1, cor2):
    """Plot epipolar lines on image given image, corners

    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)
    e1, e2 = compute_epipole(F)
    n = cor1.shape[1]
    f, (ax1, ax2) = plt.subplots(1, 2, sharey=True,figsize=(10,10))
```

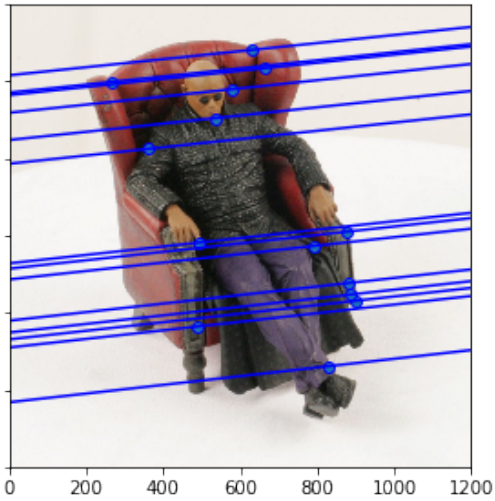
```
        ax1.imshow(img1)
        ax1.scatter(cor1[0,:],cor1[1,:],s=35,edgecolors='b')
        for i in range(cor1.shape[1]):
            k = (e1[1]-cor1[1,i])/(e1[0]-cor1[0,i])
            b = e1[1] - k * e1[0]
            x1 = 0
            x2 = img1.shape[1]-1
            y1 = k * x1 + b
            y2 = k * x2 + b
            ax1.plot((x1,x2), (y1,y2),'b')
        ax1.axis([0, img1.shape[1], img1.shape[0], 0])


        ax2.imshow(img2)
        ax2.scatter(cor2[0,:],cor2[1,:],s=35,edgecolors='b')
        for i in range(cor2.shape[1]):
            k = (e2[1]-cor2[1,i])/(e2[0]-cor2[0,i])
            b = e2[1] - k * e2[0]
            x1 = 0
            x2 = img2.shape[1]-1
            y1 = k * x1 + b
            y2 = k * x2 + b
            ax2.plot((x1,x2), (y1,y2),'b')
        ax2.axis([0, img2.shape[1], img2.shape[0], 0])
        plt.show()
```

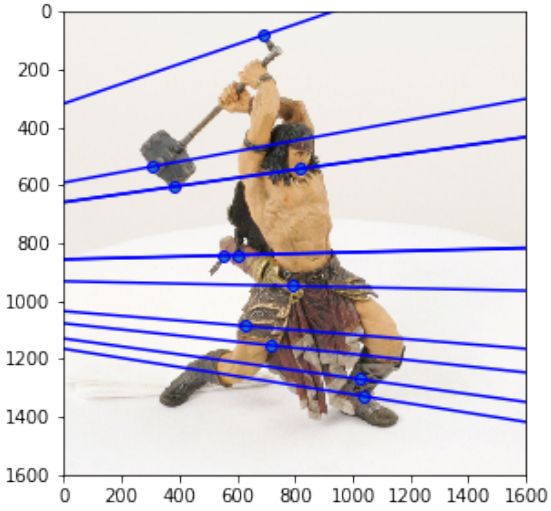
In [26]:
```
# replace images and corners with those of matrix and warrior
I1 = imageio.imread("./p4/dino/dino0.png")
I2 = imageio.imread("./p4/dino/dino1.png")
cor1 = np.load("./p4/dino/cor1.npy")
cor2 = np.load("./p4/dino/cor2.npy")
plot_epipolar_lines(I1,I2,cor1,cor2)

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

I1 = imageio.imread("./p4/warrior/warrior0.png")
I2 = imageio.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)
```

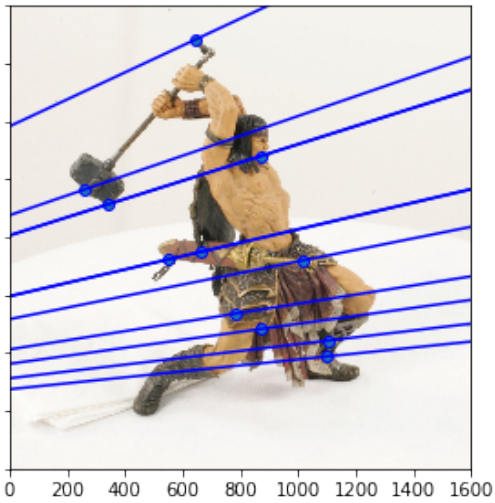

0
200
400
600
800
1000
1200
1400
0 250 500 750 1000 1250 1500 1750 2000


0 250 500 750 1000 1250 1500 1750


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


0 200 400 600 800 1000 1200


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


0 200 400 600 800 1000 1200 1400 1600

0.1.5 Image Rectification

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 $\mathtt{E} = [T_x]R = [T_x]$. Also if you observe the epipolar lines l and l' for parallel images, they are horizontal and consequently, the corresponding epipolar lines share the same vertical coordinate. Therefore the process of making images parallel becomes useful while discerning the relationships between corresponding points in images. Rectifying a pair of images can also be done for uncalibrated camera images (i.e. we do not require the camera matrix of intrinsic parameters). Using the fundamental matrix we can find the pair of epipolar lines l_i and l_i' for each of the correspondances. The intersection of these lines will give us the respective epipoles e and e'. Now to make the epipolar lines to be parallel we need to map the epipoles to infinity. Hence , we need to find a homography that maps the epipoles to infinity. The method to find the homography has been implemented for you. You can read more about the method used to estimate the homography in the paper "Theory and Practice of Projective Rectification" by Richard Hartley.

Using the compute_epipoles function from the previous part and the given compute_matching_homographies function, find the rectified images and plot the parallel epipolar lines using the plot_epipolar_lines function from above. You need to do this for both the matrix and the warrior images. A sample output will look as below:

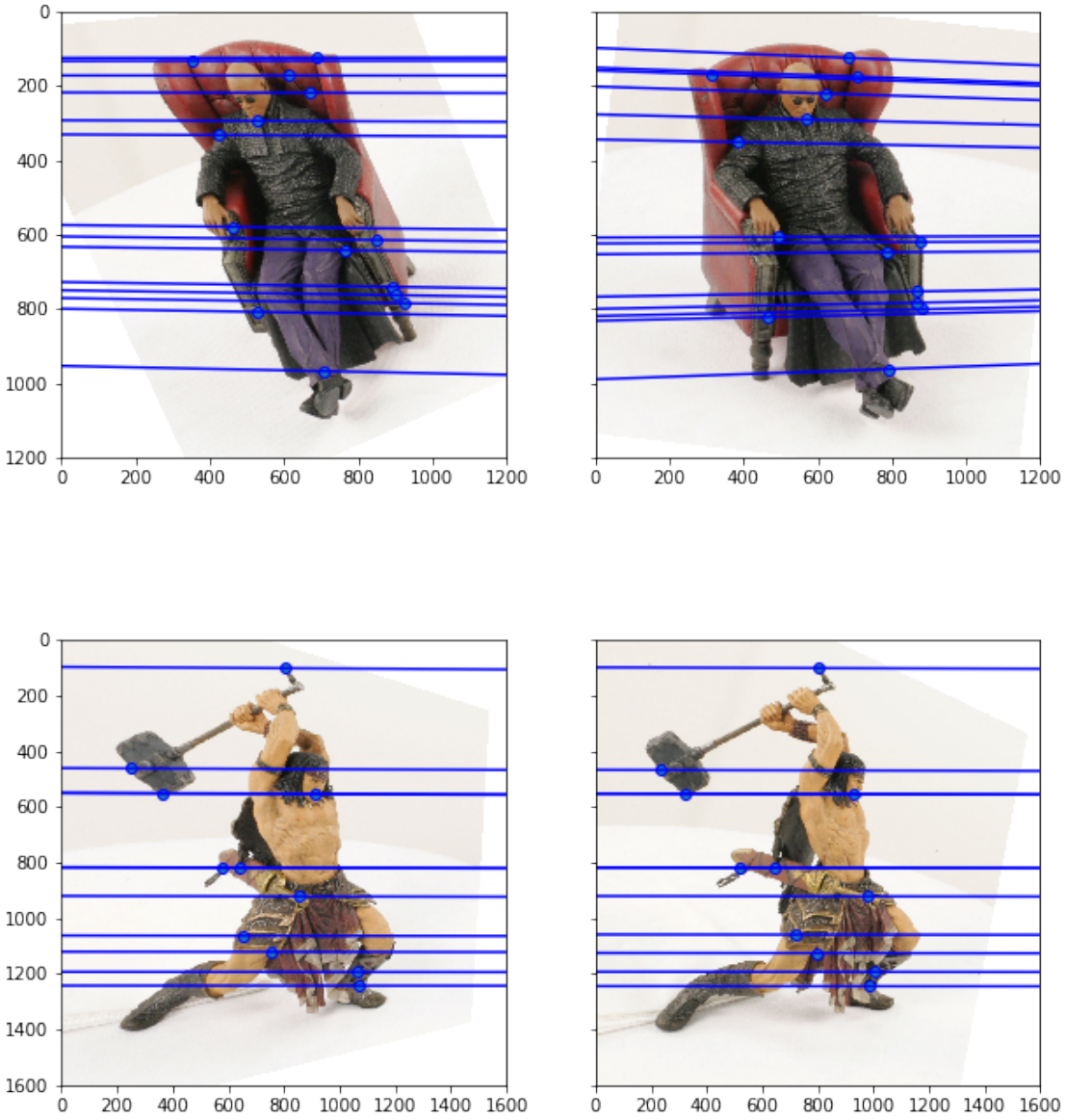
```
In [27]: import cv2
         def compute_matching_homographies(e2, F, im2, points1, points2):

             '''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
    output:
    rectified_im1-->rectified image 1
    rectified_im2-->rectified image 2
    new_cor1--> new corners in the rectified image 1
    new_cor2--> new corners in the rectified image 2
    '''
    "your code here"
    F = fundamental_matrix(points1,points2)
    e1, e2 = compute_epipole(F)
    H1, H2 = compute_matching_homographies(e2, F.T, im2, points1.T, points2.T)
    rectified_im1 = np.ones(np.shape(im1))*255
    rectified_im2 = np.ones(np.shape(im2))*255
    h,w = im2.shape[0], im2.shape[1]

    for i in range(h):
        for j in range(w):
            v = np.array([j,i,1]).T
            img1 = (np.linalg.inv(H1)).dot(v)
            img2 = (np.linalg.inv(H2)).dot(v)
            img1 = (img1/img1[2]).astype(int)
            img2 = (img2/img2[2]).astype(int)
            if img1[0]>=0 and img1[1]>=0 and img1[1]<h and img1[0]<w:
                rectified_im1[i,j,:] = im1[img1[1],img1[0],:]
            if img2[0]>=0 and img2[1]>=0 and img2[1]<h and img2[0]<w:
                rectified_im2[i,j,:] = im2[img2[1],img2[0],:]

    new_cor1 = H1.dot(points1)
#     print(new_cor1)
    new_cor1 = new_cor1/new_cor1[[2],:]
    new_cor2 = H2.dot(points2)
    new_cor2 = new_cor2/new_cor2[[2],:]

    return rectified_im1/255,rectified_im2/255,new_cor1,new_cor2
```

In [28]:
```
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,cor2)
rectified_im3,rectified_im4,new_cor3,new_cor4 = image_rectification(I3,I4,cor3,cor4)
```

```
plot_epipolar_lines(rectified_im1,rectified_im2,new_cor1,new_cor2)
plot_epipolar_lines(rectified_im3,rectified_im4,new_cor3,new_cor4)
```



0.1.6 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.

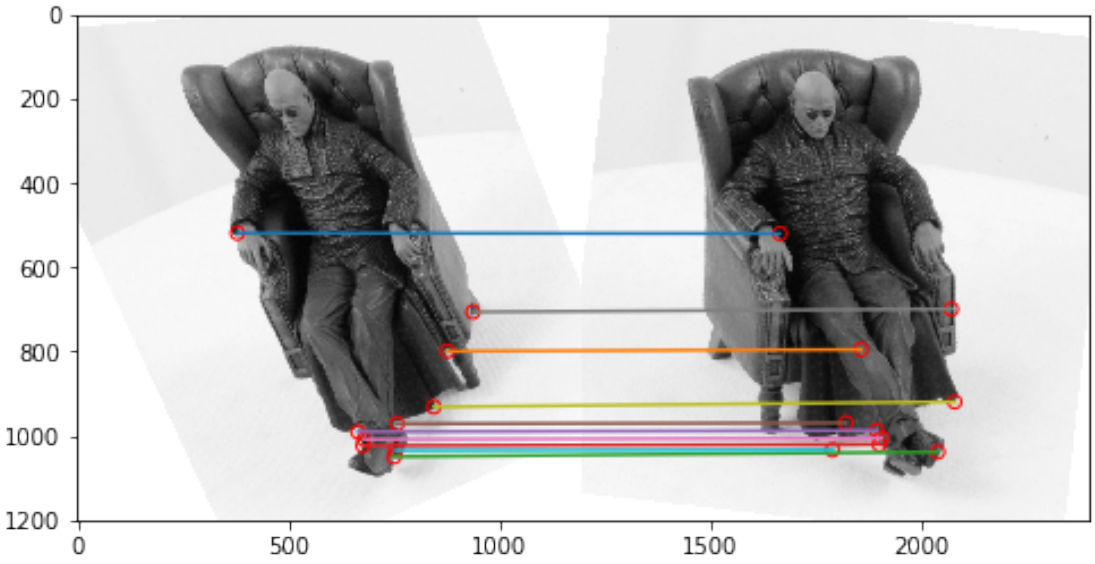
```
In [33]: def display_correspondence(img1, img2, corrs):
             """Plot matching result on image pair given images and correspondences

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

             """

             """
             Your code here.
             You may refer to the show_matching_result function
             """
             show_matching_result(img1, img2, corrs)


         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 correspondenc
                 R: NCC matching window radius.
                 NCCth: NCC matching threshold.


             Returns:
                 Matching result to be used in display_correspondence function

             """
             """
             Your code here.
             """
             matching = []
             n = len(corners1)
             cor1 = corners1.T
             cor1 = np.vstack((cor1, np.ones((1,n))))
             corners2 = []

             for i in range(len(corners1)):
                 x1 = corners1[i,0]
                 y1 = corners1[i,1]
                 b = (F.T).dot(np.array([x1,y1,1]).T)
                 b = b[:, np.newaxis]
```

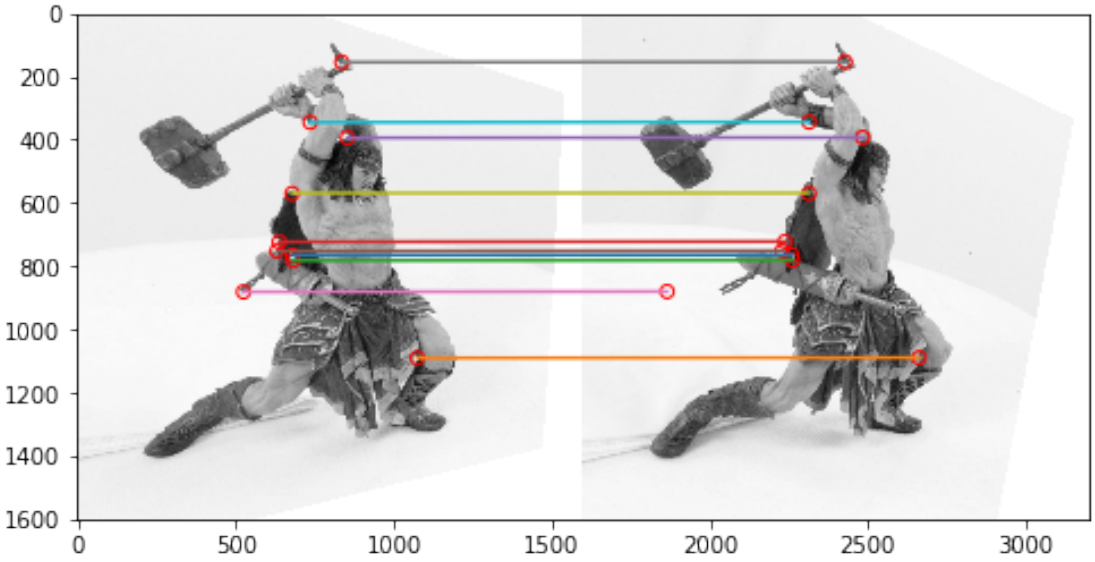
```
            n = 0
            maxscore = NCCth
            for j in range(R,img2.shape[1]-R):
                x2 = j
                y2 = int(-(b[0]*x2 + b[2])/b[1])
                c1 = np.array([x1,y1])
                c2 = np.array([x2,y2])
                matchscore = ncc_match(img1, img2, c1, c2, R)
                if matchscore >= maxscore:
                    bestpoint = np.array([x2,y2])
                    maxscore = matchscore
                    n = 1
            if n != 0:
                matching.append((corners1[i,:],bestpoint))

        return matching
```

In [34]:
```
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,cor2)
rectified_im3,rectified_im4,new_cor3,new_cor4 = image_rectification(I3,I4,cor3,cor4)
```

In [35]:
```
nCorners = 10
#decide the NCC matching window radius
R = 20
NCCth = 0.5

rectified_im1 = rgb2gray(rectified_im1)
rectified_im2 = rgb2gray(rectified_im2)
rectified_im3 = rgb2gray(rectified_im3)
rectified_im4 = rgb2gray(rectified_im4)

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_new, R, NCCth)
display_correspondence(rectified_im1, rectified_im2, corrs)

F_new2=fundamental_matrix(new_cor3, new_cor4)

corners2 = corner_detect(rectified_im3, nCorners, smoothSTD, windowSize)
```

```
corrs = correspondence_matching_epipole(rectified_im3, rectified_im4, corners2, \
                                        F_new2, R, NCCth)
display_correspondence(rectified_im3, rectified_im4, corrs)
```





In [ ]: