CSE 252A Computer Vision I Fall 2021 - Assignment 2

Instructor: Ben Ochoa

- Assignment Published On: **Wed, Oct. 20, 2021**.
- Due On: **Wed, Nov. 3, 2021 11:59 PM (Pacific Time)**.

Instructions

Please answer the questions below using Python in the attached Jupyter notebook and follow the guidelines below:

- This assignment must be completed **individually**. For more details, please follow the Academic Integrity Policy and Collaboration Policy on [Canvas (https://canvas.ucsd.edu/courses/29614)](https://canvas.ucsd.edu/courses/29614).
- All the solutions must be written in this Jupyter notebook.
- After finishing the assignment in the notebook, please export the notebook as a PDF and submit both the notebook and the PDF (i.e. the `.ipynb` and the `.pdf` files) on Gradescope.
- You may use basic algebra packages (e.g. `NumPy`, `SciPy`, etc) but you are not allowed to use the packages that directly solve the problems. Feel free to ask the instructor and the teaching assistants if you are unsure about the packages to use.
- It is highly recommended that you begin working on this assignment early.

Late Policy: Assignments submitted late will receive a 15% grade reduction for each 12 hours late (i.e., 30% per day). Assignments will not be accepted 72 hours after the due date. If you require an extension (for personal reasons only) to a due date, you must request one as far in advance as possible. Extensions requested close to or after the due date will only be granted for clear emergencies or clearly unforeseeable circumstances.

Problem 1 Image filtering [15 pts]

Problem 1.1 Implementing Convolution[5 pts]

In this problem, you will implement the convolution filtering operation in NumPy.

As shown in the lecture, a convolution can be considered as a sliding window that computes a sum of the pixel values weighted by the flipped kernel. Your version will i) zero-pad an image, ii) flip the kernel horizontally and vertically, and iii) compute a weighted sum of the neighborhood at each pixel.

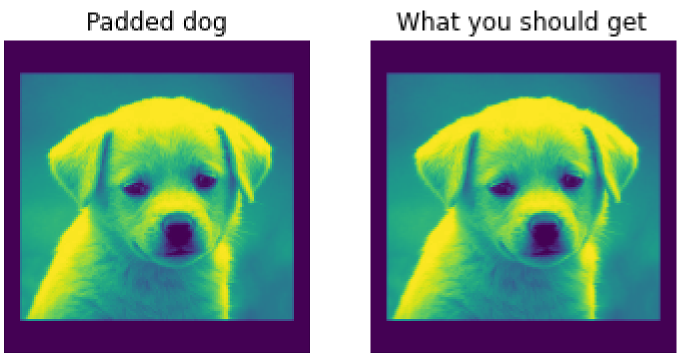
Problem 1.1.1 [1 pts]

First you will want to implement the **zero_pad** function.

In [1]:
```
import numpy as np
from time import time
from skimage import io
from heapq import *
%matplotlib inline
import matplotlib.pyplot as plt
```

In [2]:
```
def zero_pad(image, pad_top, pad_down, pad_left, pad_right):
    """ Zero-pad an image.

    Ex: a 1x1 image [[1]] with pad_top = 1, pad_down = 1, pad_left = 2, pad_right = 2 becor

        [[0, 0, 0, 0, 0],
         [0, 0, 1, 0, 0],
         [0, 0, 0, 0, 0]]         of shape (3, 5)

    Args:
        image: numpy array of shape (H, W)
        pad_left: width of the zero padding to the left of the first column
        pad_right: width of the zero padding to the right of the last column
        pad_top: height of the zero padding above the first row
        pad_down: height of the zero padding below the last row

    Returns:
        out: numpy array of shape (H + pad_top + pad_down, W + pad_left + pad_right)
    """
    """ ==========
    YOUR CODE HERE
    ========== """

    height = np.size(image,0)
    width = np.size(image,1)
    out = np.zeros([height+pad_top+pad_down, width+pad_left+pad_right])
    out[pad_top:pad_top+height, pad_left:pad_left+width] = image

    return out

# Open image as grayscale
img = io.imread('dog.jpg', as_gray=True)

# Show image
plt.imshow(img)
plt.axis('off')
plt.show()

pad_width = 20 # width of the padding on the left and right
pad_height = 40 # height of the padding on the top and bottom

padded_img = zero_pad(img, pad_height, pad_height, pad_width, pad_width)

# Plot your padded dog
plt.subplot(1,2,1)
plt.imshow(padded_img)
plt.title('Padded dog')
plt.axis('off')

# Plot what you should get
solution_img = io.imread('padded_dog.jpg', as_gray=True)
plt.subplot(1,2,2)
plt.imshow(solution_img)
plt.title('What you should get')
plt.axis('off')
```

```
plt.show()
```



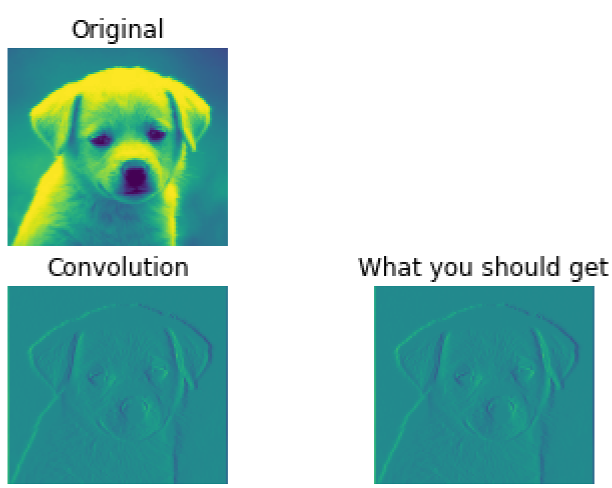
Problem 1.1.2 [2 pts]

Now implement the function `conv` , **using at most 2 loops**. This function should take an image f and a kernel h as inputs and output the convolved image $(f * h)$ that has the same shape as the input image (use zero padding to accomplish this). Depending on the computer, your implementation should take around a second or less to run.

In [3]:
```python
def conv(image, kernel):
    """ An efficient implementation of a convolution filter.

    This function uses element-wise multiplication and np.sum()
    to efficiently compute a weighted sum of the neighborhood at each
    pixel.

    Hints:
        - Use the zero_pad function you implemented above
        - You should need at most two nested for-loops
        - You may find np.flip() and np.sum() useful
        - You need to handle both odd and even kernel size

    Args:
        image: numpy array of shape (Hi, Wi)
        kernel: numpy array of shape (Hk, Wk)

    Returns:
        out: numpy array of shape (Hi, Wi)
    """
    Hi, Wi = image.shape
    Hk, Wk = kernel.shape
    out = np.zeros((Hi, Wi))

    """ ==========
    YOUR CODE HERE
    ========== """

    kernel = np.flip(kernel, 0)
    kernel = np.flip(kernel, 1)

    pad_top = 0
    pad_down = 0
    pad_left = 0
    pad_right = 0

    if Hk % 2 == 0:
        pad_top = Hk//2 - 1
        pad_down = Hk//2
    else:
        pad_top = Hk//2
        pad_down = Hk//2

    if Wk % 2 == 0:
        pad_left = Wk//2 - 1
        pad_right = Wk//2
    else:
        pad_left = Wk//2
        pad_right = Wk//2

    image_pad = zero_pad(image, pad_top, pad_down, pad_left, pad_right)

    for i in range(Hi):
        for j in range(Wi):
            out[i,j]=np.sum(image_pad[i:i+Hk,j:j+Wk]*kernel)
```

```
    return out

# Simple convolution kernel.
# Feel free to change the kernel and to see different outputs.
kernel = np.array(
[
    [1,0,-1],
    [2,0,-2],
    [1,0,-1]
])

t1 = time()
out = conv(img, kernel)
t2 = time()
print("took %f seconds." % (t2 - t1))

# Plot original image
plt.subplot(2,2,1)
plt.imshow(img)
plt.title('Original')
plt.axis('off')

# Plot your convolved image
plt.subplot(2,2,3)
plt.imshow(out)

plt.title('Convolution')
plt.axis('off')

# Plot what you should get
solution_img = io.imread('convolved_dog.jpg', as_gray=True)
plt.subplot(2,2,4)
plt.imshow(solution_img)
plt.title('What you should get')
plt.axis('off')

plt.show()
```

took 0.887101 seconds.

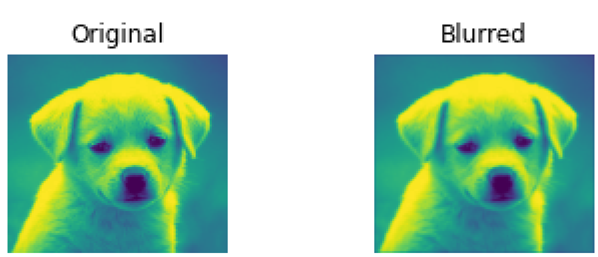


Problem 1.1.3 [1 pt]

Now let's filter some images! Here, you will apply the convolution function that you just implemented in order to bring about some interesting image effects. More specifically, we will use convolution to blur and sharpen our images.

First we will apply convolution for image blurring. To accomplish this, convolve the dog image with a 7x7 Gaussian filter for $\sigma = 1.0$. You can use the included function to obtain the Gaussian kernel.

In [4]:
```
def gaussian2d(filter_size=7, sig=1.0):
    """
    Creates 2D Gaussian kernel with side length `filter_size` and a sigma of `sig`.
    """
    ax = np.arange(-filter_size // 2 + 1., filter_size // 2 + 1.)
    xx, yy = np.meshgrid(ax, ax)
    kernel = np.exp(-0.5 * (np.square(xx) + np.square(yy)) / np.square(sig))
    return kernel / np.sum(kernel)

def blur_image(img):
    """Blur the image by convolving with a Gaussian filter."""
    blurred_img = np.zeros_like(img)
    """ ==========
    YOUR CODE HERE
    ========== """

    kernel = gaussian2d()
    blurred_img = conv(img, kernel)

    return blurred_img

# Plot original image
plt.subplot(2,2,1)
plt.imshow(img)
plt.title('Original')
plt.axis('off')

# Plot blurred image
plt.subplot(2,2,2)
plt.imshow(blur_image(img))
plt.title('Blurred')
plt.axis('off')

plt.show()
```



Problem 1.1.4 [1 pt]

Next, we will use convolution to sharpen the images. Convolve the image with the following filter to produce a sharpened result. For convenience, we have defined the filter for you:

In [5]:
```
sharpening_kernel = np.array([
    [1,  4,     6,  4,  1],
    [4,  16,   24,  16,  4],
    [6,  24, -476,  24,  6],
    [4,  16,   24,  16,  4],
    [1,   4,    6,   4,  1],
]) * -1.0 / 256.0
```

In [6]:
```
def sharpen_image(img):
    """Sharpen the image by convolving with a sharpening filter."""
    sharpened_img = np.zeros_like(img)
    """ ==========
    YOUR CODE HERE
    ========== """

    sharpened_img = conv(img, sharpening_kernel)

    return sharpened_img

# Plot original image
plt.subplot(2,2,1)
plt.imshow(img, vmin=0.0, vmax=1.0)
plt.title('Original')
plt.axis('off')

# Plot sharpened image
plt.subplot(2,2,2)
plt.imshow(sharpen_image(img), vmin=0.0, vmax=1.0)
plt.title('Sharpened')
plt.axis('off')

plt.show()
```

Original

Sharpened

Problem 1.2: Convolution Theory [5 pts]

Problem 1.2.1 [2 pts]

Consider (1) smoothing an image with a 3x3 box filter and then computing the derivative in the x-direction. Also consider (2) computing the derivative first, then smoothing. What is a single convolution kernel that will simultaneously implement both (1) and (2)? Try to give a brief justification for how you arrived at the kernel.

for how you arrived at the kernel.
Use the x-derivative filter

$$[-1/2, 0, 1/2]$$

for this problem.

```
Assume that the gradient convolution kernel is D, the smooth convolution kernel is K,
and the image is I
so (1) can express as I*(K*D), (2) can express as (I*D)*K Convolution has commutative
and associative laws
(I*K)*D=I*(K*D),(I*D)*K=I*(D*K), Due to commutative law K*D=D*K,so (I*K)*D=(I*D)*K,
So we can define a new convolution kernel M,
M=K*D=D*K,so I*M can process 1 and 2 operations at the same time
```

Problem 1.2.2 [3 pts]

Certain 2D filters can be expressed as a convolution of two 1D filters. Such filters are called separable filters. Give an example of a 3x3 separable filter and compare the number of arithmetic operations it takes to convolve an n x n image using that filter before and after separation. Count both, the number of multiplication and addition operations in each case.

Assume that the convolution of the image and filter is performed in "valid" mode, i.e., the image is not padded before convolution.

Type *Markdown* and LaTeX: α^2

```
For the picture of n*n, the convolution of 3*3 once requires 9 times of multiplication
and 8 times of addition,
so a total of 9n^2 times of multiplication is required and 8n^2 times of addition is
required

For the picture of n*n, the convolution of 1*3 once requires 3 times of multiplication
and 2 times of addition,
so a total of 3n^2 times of multiplication is required and 2*n^2 additions,
performing a convolution of 3*1 needs to do 3 times multiplications and 2 times
additions, so a total of 3*n^2 multiplications and
2*n^2 Times addition. Therefore, a total of 6*n^2 times  multiplications and 4*n^2
times additions are required.
It can be found that solving the convolution of into two one-dimensional convolutions
can reduce the amount of calculation while reducing the amount of parameters
```

Problem 1.3 Template Matching [5 pts]

Suppose that you are a clerk at a grocery store. One of your responsibilites is to check the shelves periodically and stock them up whenever there are sold-out items. You got tired of this laborious task and decided to build a computer vision system that keeps track of the items on the shelf.

Luckily, you have learned in CSE 252A (or are learning right now) that convolution can be used for template matching: a template g is multiplied with regions of a larger image f to measure how similar each region is to the template. Note that you will want to flip the filter before giving it to your convolution function, so that it is overall not flipped when making comparisons. You will also want to subtract off the mean value of the image or template (whichever you choose, subtract the same value from both the image and template) so that our solution is not biased toward higher-intensity (white) regions.

The template of a product (template.jpg) and the image of the shelf (shelf.jpg) is provided. We will use convolution to find the product in the shelf.

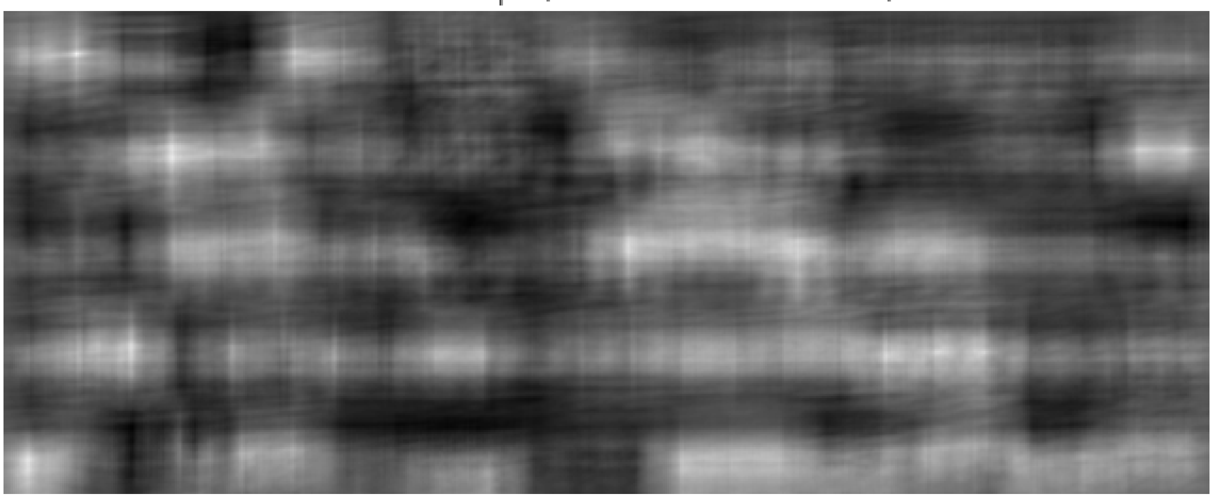
template

shelf

In [7]:
```
# Load template and image in grayscale
img = io.imread('shelf.jpg')
img_gray = io.imread('shelf.jpg', as_gray=True)
temp = io.imread('template.jpg')
temp_gray = io.imread('template.jpg', as_gray=True)

# Perform a convolution between the image and the template and store
# the result in the out variable
""" ==========
YOUR CODE HERE
========== """

temp_gray = np.flip(np.flip(temp_gray, 0), 1)
out = conv(img_gray-np.mean(img_gray), temp_gray-np.mean(img_gray))
mx = np.max(out)
h, w = out.shape
x = []
y = []
for i in range(h):
    for j in range(w):
        if out[i, j] ==  mx:
            y.append(i)
            x.append(j)

# Display product template
plt.figure(figsize=(20,16))
plt.subplot(3, 1, 1)
plt.imshow(temp_gray, cmap="gray")
plt.title('Template')
plt.axis('off')

# Display convolution output
plt.subplot(3, 1, 2)
plt.imshow(out, cmap="gray")
plt.title('Convolution output (white means more correlated)')
plt.axis('off')

# Display image
plt.subplot(3, 1, 3)
plt.imshow(img, cmap="gray")
plt.title('Result (blue marker on the detected location)')
plt.axis('off')

# Draw marker at detected location
plt.plot(x, y, 'bx', ms=40, mew=10)
plt.show()
```

Template

Convolution output (white means more correlated)



Result (blue marker on the detected location)

Problem 2: Edge detection [21 pts]

In this problem, you will write a function to perform Canny edge detection. The following steps need to be implemented.

Problem 2.1 Smoothing [1 pt]

First, we need to smooth the images in order to prevent noise from being considered as edges. For this assignment, use a 9x9 Gaussian kernel filter with $\sigma = 1.4$ to smooth the images.

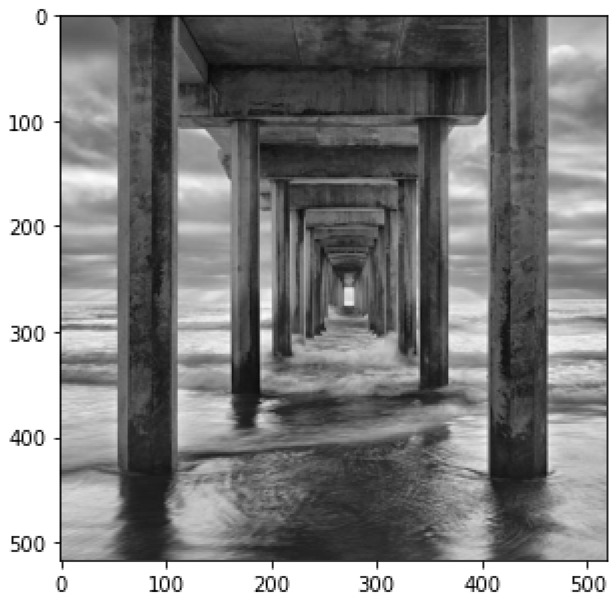
In [8]:
```
import numpy as np
from skimage import io
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from scipy.signal import convolve
%matplotlib inline

import matplotlib
matplotlib.rcParams['figure.figsize'] = [5, 5]
```

In [9]:
```
def gaussian2d(filter_size=None, sig=None):
    """Creates a 2D Gaussian kernel with
    side length `filter_size` and a sigma of `sig`."""
    ax = np.arange(-filter_size // 2 + 1., filter_size // 2 + 1.)
    xx, yy = np.meshgrid(ax, ax)
    kernel = np.exp(-0.5 * (np.square(xx) + np.square(yy)) / np.square(sig))
    return kernel / np.sum(kernel)
```

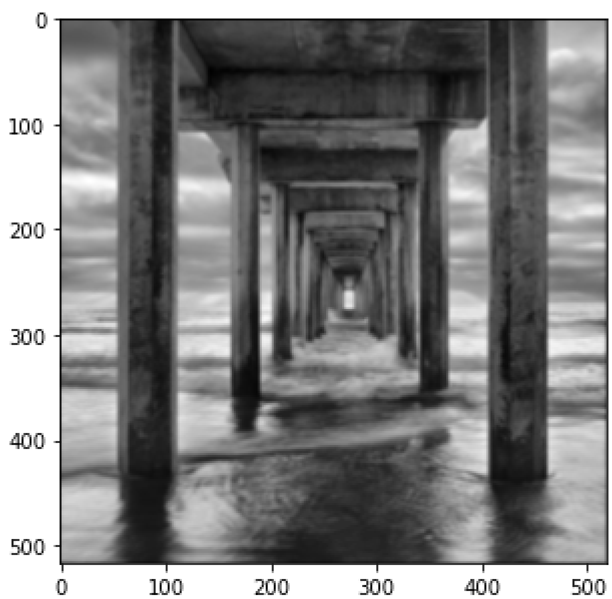
In [10]:
```
def smooth(image):
    """ ==========
    YOUR CODE HERE
    ========== """
    kernel = gaussian2d(9, 1.4)
    out = conv(image, kernel)
    return out
```

In [11]:
```
# Load image in grayscale
image = io.imread('sio_pier.jpg', as_gray=True)
assert len(image.shape) == 2, 'image should be grayscale; check your Python/skimage versi
smoothed = smooth(image)
print('Original:')
plt.imshow(image, cmap=cm.gray)
plt.show()

print('Smoothed:')
plt.imshow(smoothed, cmap=cm.gray)
plt.show()
```

Original:



Smoothed:



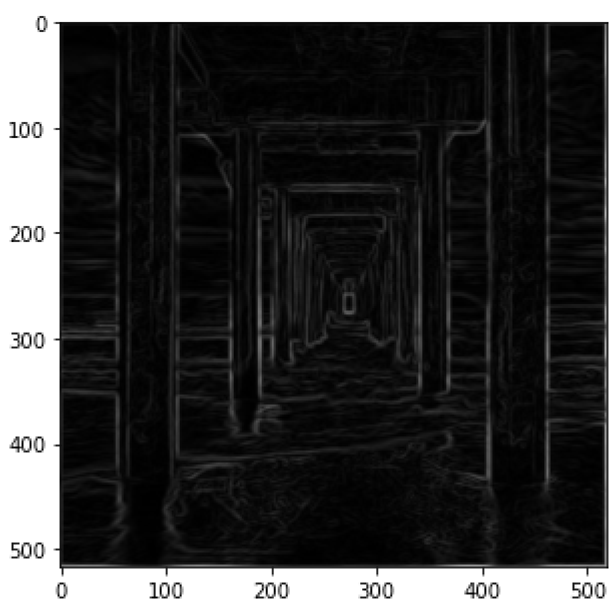
Problem 2.2 Gradient Computation [5 pts]

After you have finished smoothing, find the image gradient in the horizontal and vertical directions. Compute the gradient magnitude image as $|G| = \sqrt{G_x^2 + G_y^2}$. The edge direction for each pixel is given by $G_\theta = \tan^{-1}\left(\frac{G_y}{G_x}\right)$.

In [12]:
```
def gradient(image):
    """ ==========
    YOUR CODE HERE
    ========== """
    kernel_h = np.array([[-1/2,0,1/2]])
    kernel_v = np.array([-1/2,0,1/2]).reshape(-1, 1)
    G_x = conv(image,kernel_h)
    G_y = conv(image,kernel_v)
    g_mag = np.sqrt(G_x*G_x+G_y*G_y)
    g_theta = np.arctan(G_y/G_x)
    return g_mag, g_theta
```

In [13]:
```
g_mag, g_theta = gradient(smoothed)
print('Gradient magnitude:')
plt.imshow(g_mag, cmap=cm.gray)
plt.show()
```

Gradient magnitude:



Problem 2.3 Non-Maximum Suppression [7 pts]

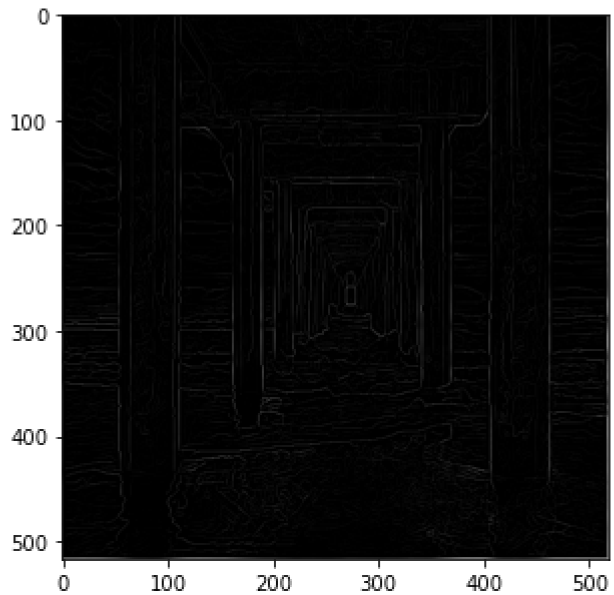
We would like our edges to be sharp, unlike the ones in the gradient image. Use non-maximum suppression to preserve all local maxima and discard the rest. You can use the following method to do so:

- For each pixel in the gradient magnitude image:
  - Round the gradient direction θ to the nearest multiple of 45° (which we will refer to as ve).
  - Compare the edge strength at the current pixel to the pixels along the $+ve$ and $-ve$ gradient direction in the 8-connected neighborhood.
  - If the pixel does not have a larger value than both of its two neighbors in the $+ve$ and $-ve$ gradient directions, suppress the pixel's value (set it to 0). By following this process, we preserve the values of only those pixels which have maximum gradient magnitudes in the neighborhood along the $+ve$ and $-ve$ gradient directions.
- Return the result as the NMS response.

In [14]:
```
def nms(g_mag, g_theta):
    """ ==========
    YOUR CODE HERE
    ========== """
    nms_response = np.zeros_like(g_mag)
    h, w = g_theta.shape
    for i in range(h):
        for j in range(w):
            cur_theta = g_theta[i, j]
            cur_theta /= np.pi/2
            cur_theta = np.round(cur_theta)
            flag = True
            if cur_theta == 0.0:
                if j+1 < w and g_mag[i, j] < g_mag[i, j+1]:
                    flag = False
                if j-1 >=0 and g_mag[i, j] < g_mag[i, j-1]:
                    flag = False
            elif cur_theta == 1.0:
                if i+1 < h and j+1 < w and g_mag[i, j] < g_mag[i+1, j+1]:
                    flag = False
                if i-1 >=0 and j-1 >= 0 and g_mag[i, j] < g_mag[i-1, j-1]:
                    flag = False
            elif cur_theta == 2.0:
                if i+1 < h and g_mag[i, j] < g_mag[i+1, j]:
                    flag = False
                if i-1 >= 0 and g_mag[i, j] < g_mag[i-1,  j]:
                    flag = False
            elif cur_theta == -1.0:
                if i-1 >= 0 and j+1 < w and g_mag[i, j] < g_mag[i-1, j+1]:
                    flag = False
                if i+1 < h and j-1 >= 0 and g_mag[i, j] < g_mag[i+1, j-1]:
                    flag = False
            elif cur_theta ==  -2.0:
                if i-1 >= 0 and g_mag[i, j] < g_mag[i-1, j]:
                    flag = False
                if i+1 < h and g_mag[i, j] < g_mag[i+1, j]:
                    flag = False
            if flag:
                nms_response[i, j] = g_mag[i, j]
            else:
                nms_response[i, j] = 0

    return nms_response
```

In [15]:
```
nms_image = nms(g_mag, g_theta)
print('NMS:')
plt.imshow(nms_image, cmap=cm.gray)
plt.show()
```

NMS:



Problem 2.4 Hysteresis Thresholding [8 pts]

Choose suitable values of thresholds and use the thresholding approach decribed in lecture 6. This will remove the edges caused by noise and color variations.

- Define two thresholds `t_min` and `t_max` .
- If the `nms > t_max` , then we select that pixel as an edge.
- If `nms < t_min` , we reject that pixel.
- If `t_min < nms < t_max` , we select the pixel only if there is a path from to another pixel with `nms > t_max` . (Hint: Think of all pixels with `nms > t_max` as starting points and run BFS/DFS from these starting points).
- The choice of value of low and high thresholds depends on the range of values in the gradient magnitude image. You can start by setting the high threshold to some percentage of the max value in the gradient magnitude image, e.g. thres_high = 0.2 * image.max(), and the low threshold to some percentage of the high threshold, e.g. thres_low = 0.85 * thres_high. And then you can tune those values however you want.

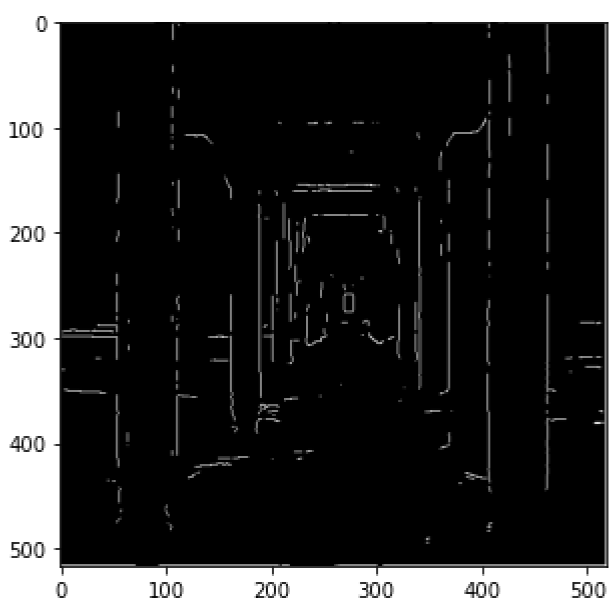
In [16]:
```
def hysteresis_threshold(image, g_theta, use_g_theta=False):
    """ ==========
    YOUR CODE HERE
    ========== """
    h, w = image.shape
    result = np.zeros_like(image)
    t_max = np.max(image)*0.2
    t_min = t_max*0.85
    def valid_coordinate(y,x):
        if image[y,x] > t_min and image[y,x] < t_max:
            return True
        else:
            return False
    vis = np.zeros([h,w])
    for i in range(h):
        for j in range(w):
            if image[i,j] > t_max and result[i,j] != 1:
                result[i,j] = 1
                que = [(i,j)]
                while len(que)>0:
                    y,x = que.pop()
                    theta = g_theta[y,x]
                    theta /= np.pi/2
                    theta = np.round(theta)
                    if theta == 0.0:
                        if y+1<h and result[y+1,x]!=1 and vis[y+1,x]!=1:
                            que.append((y+1,x))
                            vis[y+1,x]=1;
                            if valid_coordinate(y+1,x):
                                result[y+1,x]=1;
                        if y-1>=0 and result[y-1,x]!=1 and vis[y-1,x]!=1:
                            que.append((y-1,x))
                            vis[y-1,x]=1
                            if valid_coordinate(y-1,x):
                                result[y-1,x]=1
                    elif theta == 1.0:
                        if y+1<h and x-1>=0 and result[y+1,x-1]!=1 and vis[y+1,x-1]!=1:
                            que.append((y+1,x-1))
                            vis[y+1,x-1]=1
                            if valid_coordinate(y+1,x-1):
                                result[y+1,x-1]=1;
                        if y-1>=0 and x+1<w and result[y-1,x+1]!=1 and vis[y-1,x+1]!=1:
                            que.append((y-1,x+1))
                            vis[y-1,x+1]=1
                            if valid_coordinate(y-1,x+1):
                                result[y-1,x+1]=1
                    elif theta == 2.0 or theta == -2.0:
                        if x+1<w and result[y,x+1]!=1 and vis[y,x+1]!=1:
                            que.append((y,x+1))
                            vis[y,x+1]=1
                            if valid_coordinate(y,x+1):
                                result[y,x+1]=1;
                        if x-1>=0 and result[y,x-1]!=1 and vis[y,x-1]!=1:
                            que.append((y,x-1))
                            vis[y,x-1]=1
                            if valid_coordinate(y,x-1):
```

```
                                result[y,x-1]=1
                    elif theta == -1.0:
                        if y+1<h and x+1<w and result[y+1,x+1]!=1 and vis[y+1,x+1]!=1:
                            que.append((y+1,x+1))
                            vis[y+1,x+1]=1
                            if valid_coordinate(y+1,x+1):
                                result[y+1,x+1]=1;
                        if y-1>=0 and x-1>=0 and result[y-1,x-1]!=1 and vis[y-1,x-1]!=1
                            que.append((y-1,x-1))
                            vis[y-1,x-1]=1
                            if valid_coordinate(y-1,x-1):
                                result[y-1,x-1]=1
            elif image[i,j] < t_min:
                result[i,j]=0
    return result
```

In [17]:

```
thresholded = hysteresis_threshold(nms_image, g_theta)
print('Thresholded:')
plt.imshow(thresholded, cmap=cm.gray)
plt.show()
```

Thresholded:



Problem 3 Corner detection [13 pts]

Problem 3.1 [12 pts]

In this problem, we are going to build a corner detector. This should be done according to the lecture slides. You should fill in the function `corner_detect` below, which takes as input `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; 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 display outputs as shown below. You may find `scipy.ndimage.filters.gaussian_filter` helpful for smoothing.

In this problem, try the following different standard deviation (σ) parameters for the Gausian smoothing kernel: 0.5, 1, 2 and 4. For a particular σ, you should take the kernel size to be $6 \times \sigma$ (add 1 if the kernel size is even). So for example if $\sigma = 2$, corner detection kernel size should be 13. This should be followed throughout all of the experiments in this assignment.

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

dinoCorner1

dinoCorner2

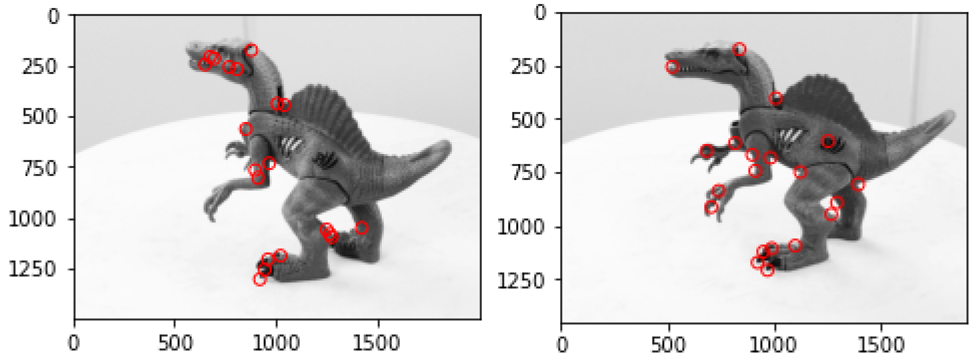
In [18]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage.filters import gaussian_filter
import imageio
from scipy.signal import convolve

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

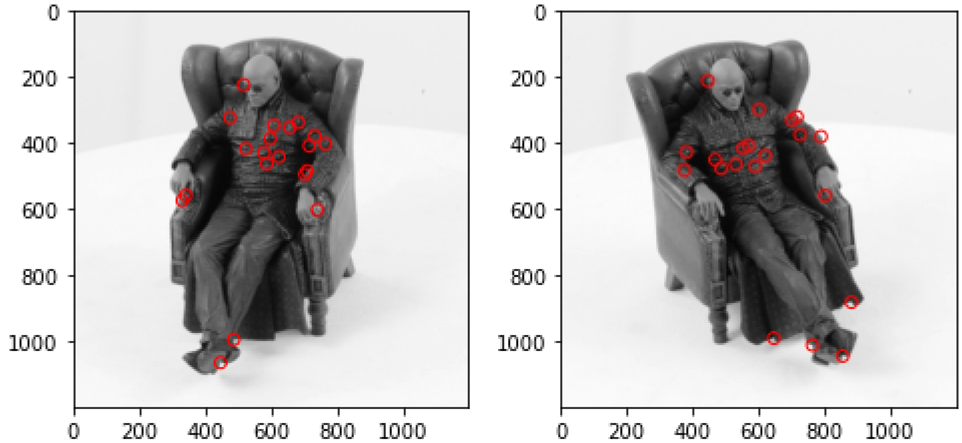
In [19]:
```
def corner_detect(image, nCorners, smoothSTD, windowSize):
    """Detect corners on a given image.

    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

    filter_size = 6 * smoothSTD
    if filter_size % 2 == 0:
        filter_size += 1

    kernel = gaussian2d(filter_size, smoothSTD)

    image_f = conv(image, kernel)

    corners = np.zeros((nCorners,2))
    ix = np.zeros((h, w))
    iy = np.zeros((h, w))

    for i in range(h):
        for j in range(w):
            if j>0:
                ix[i,j] = image[i,j]-image[i,j-1]
            else:
                ix[i,j] = image[i,j]
            if i>0:
                iy[i,j] = image[i,j]-image[i-1,j]
            else:
                iy[i,j] = image[i,j]
    ixx = ix*ix
    ixy = ix*iy
    iyy = iy*iy

    def comput_i(mat,i,j):
        if i>0:
            mat[i,j]+=mat[i-1,j]
        if i>0 and j>0:
            mat[i,j]-=mat[i-1,j-1]
        if j>0:
            mat[i,j]+=mat[i,j-1]

    for i in range(h):
        for j in range(w):
```

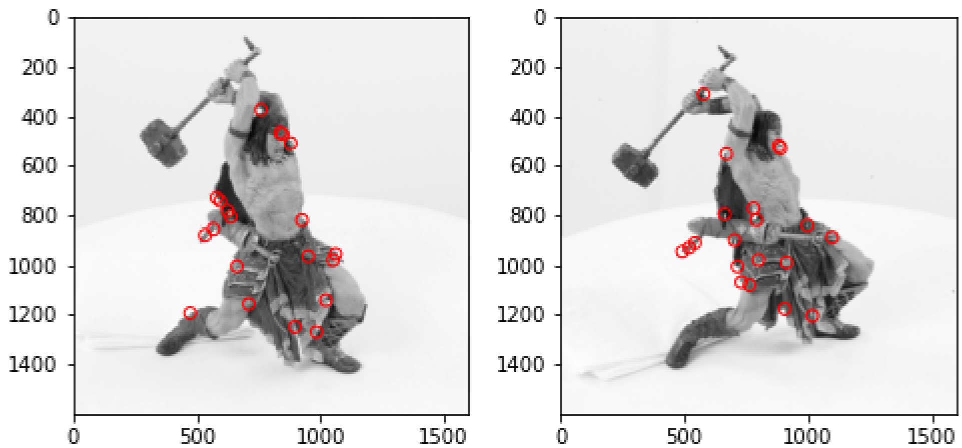
```
            comput_i(ixx, i, j)
            comput_i(iyy, i, j)
            comput_i(ixy, i, j)

    def comput_m(m,  i,  j,  w):
        res = m[i+w-1, j+w-1]+0
        if i>0:
            res -= m[i-1, j+w-1]
        if j>0:
            res -= m[i+w-1, j-1]
        if i>0 and j>0:
            res += m[i-1, j-1]
        return res

    score = np.zeros((h,  w))

    for i in range(h-windowSize):
        for j in range(w-windowSize):
            M = np.zeros((2,2))
            M[0,0]=comput_m(ixx,i,j,windowSize)
            M[0,1]=comput_m(ixy,i,j,windowSize)
            M[1,0]=comput_m(ixy,i,j,windowSize)
            M[1,1]=comput_m(iyy,i,j,windowSize)
            l,  v =  np.linalg.eig(M)
            score[i,j] = l[0]*l[1]-0.05*(l[0]+l[1])*(l[0]+l[1])

    lst = []
    for i in range(windowSize,h-2*windowSize):
        for j in range(windowSize,w-2*windowSize):
            flag = True
            for si in range(-1,1):
                for sj in range(-1,1):
                    ci = si + i
                    cj = sj + j
                    if score[i,j] < score[ci,cj]:
                        flag = False
            if flag:
                lst.append((score[i,j],i,j))

    lst.sort()
    cnt = 0
    while cnt<nCorners and len(lst)>0:
        score,  i,  j = lst.pop()
        flag = True
        for idx in range(cnt):
            if abs(j-corners[idx,0])<2*windowSize or abs(i-corners[idx,1])<2*windowSize:
                flag = False
                break
        if flag:
            corners[cnt,0] = j
            corners[cnt,1] = i
            cnt += 1

    corners += windowSize / 2

    return corners
```

In [240]:
```
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()

for smoothSTD in (0.5, 1, 2, 4):
    windowSize = int(smoothSTD * 6)
    if windowSize % 2 == 0:
        windowSize += 1

    print('smooth stdev: %r' % smoothSTD)
    print('window size: %r' % windowSize)

    nCorners = 20

    # read images and detect corners on images

    imgs_din = []
    crns_din = []
    imgs_mat = []
    crns_mat = []
    imgs_war = []
    crns_war = []

    for i in range(2):
        img_din = imageio.imread('dino/dino' + str(i) + '.png')
        imgs_din.append(rgb2gray(img_din))
        # downsize your image in case corner_detect runs slow in test
        # imgs_din.append(rgb2gray(img_din)[::2, ::2])
        crns_din.append(corner_detect(imgs_din[i], nCorners, smoothSTD, windowSize))

        img_mat = imageio.imread('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('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_din, crns_din)
    show_corners_result(imgs_mat, crns_mat)
    show_corners_result(imgs_war, crns_war)
```

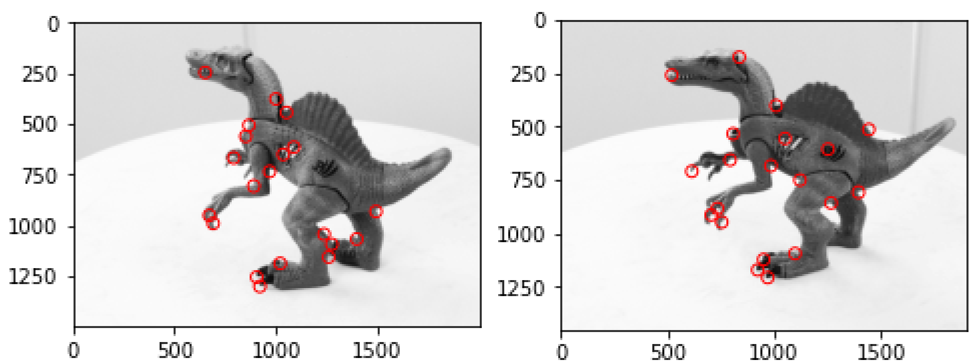
```
smooth stdev: 0.5
window size: 3
```

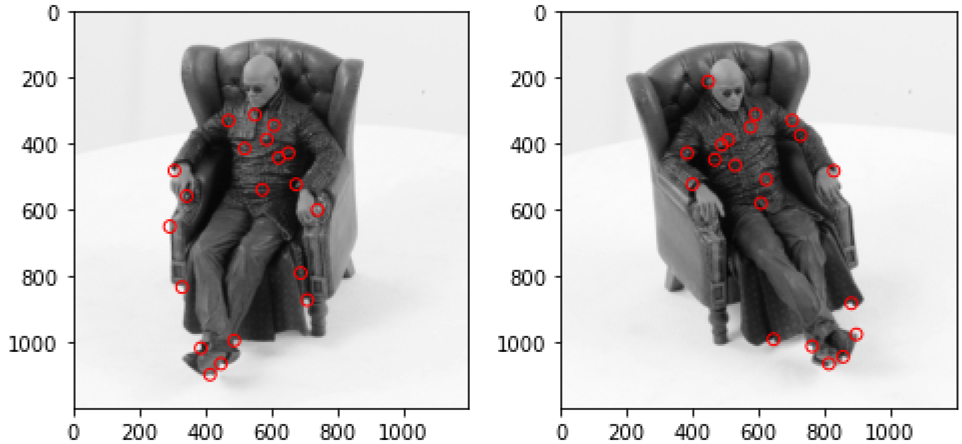


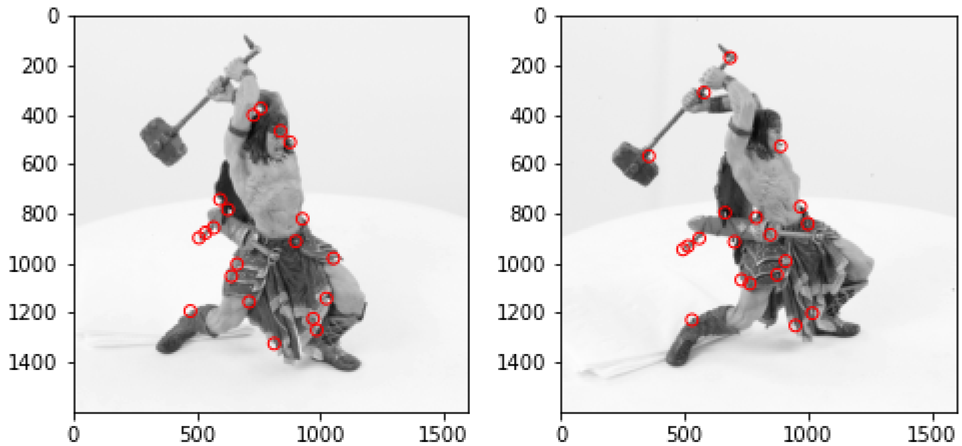




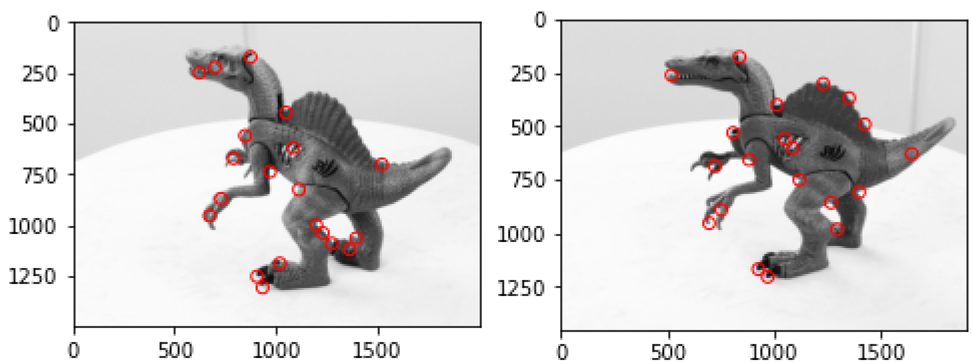
```
smooth stdev: 1
window size: 7
```

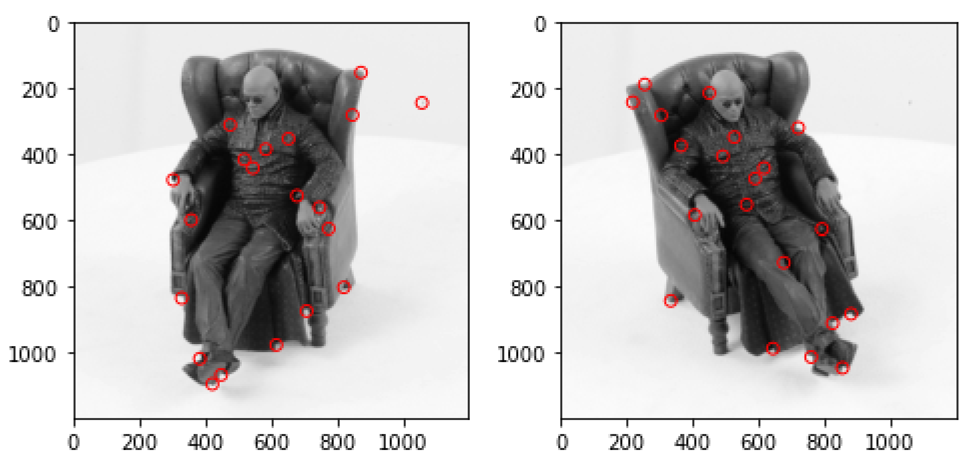


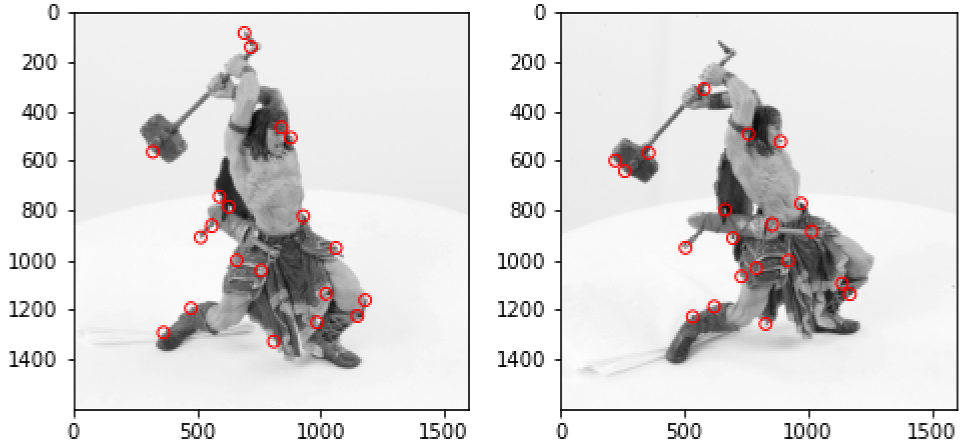




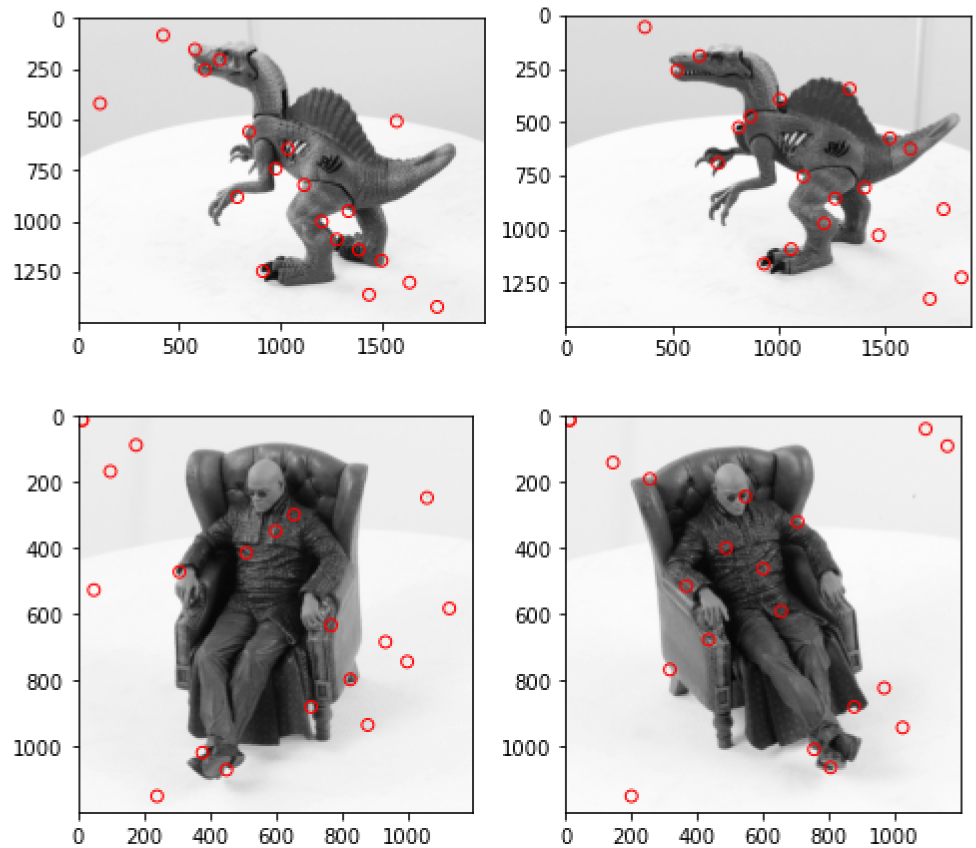
```
smooth stdev: 2
window size: 13
```

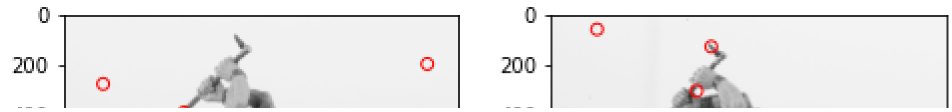







```
smooth stdev: 4
window size: 25
```





Problem 3.2 [1 pts]

Comment on your results and observations. You don't need to comment per output; just discuss any trends you see for the detected corners as you change the windowSize and increase the smoothing w.r.t the two pairs of images (warrior and matrix). Also discuss whether you are able to find corresponding corners for the pairs of images.

```
Increase the window size, the detection range becomes larger, and the detection points are more scattered,
but no small inflection points can be detected. When the variance reaches 4, only A small number of inflection points were detected.
```

Problem 4 Epipolar rectification and feature matching [43 pts]

4.1 Epipolar rectification [22 pts]

In this problem, we are going to perform epipolar rectification. Given calibrated stereo cameras (i.e., calibration matrices K_1 and K_2, camera rotation matrices R_1 and R_2, camera translation vectors t_1 and t_2), you are expected to determine the rotation matrix R and calibration matrix K of the virtual cameras. Your goal is to complete the function $epipolarRecification$, which determines the calibration matrix and rotation matrix of both cameras, the translation vector of each of the cameras, and matching planar transformations that epipolar rectify the two images acquired by the cameras. The destination virtual cameras have the same centers as the source real cameras.

4.1.1 Camera translation matrices and Projective Transformation matrices [6 pts]

To calculate the camera translation from cameras with the same camera center, you will have to complete the $cameraTranslation$ first. Another function you need to complete is $calcProjectiveTransformation$, which calculates the planar projective transformation from cameras with the same camera center. The camera calibration matrix (same for both cameras) will be calculated by $calcDestinateK$. This is provided for you. To get the rotation matrix R of the virtual camera, we usually interpolate halfway between the two 3D rotations embodied by R_1 and R_2. For simplicity, this will be also given to you.

In [20]:
```python
from imageio import imread
import matplotlib.pyplot as plt
import numpy as np
import math
import pickle
from math import floor, ceil
```

In [21]:
```
def cameraTranslation(R1, t1, R2):
    '''
    Calculate the camera translation from cameras with the same camera center.

    Args:
    R1: The rotation matrix of the first camera.
    t1: The translation vector of the first camera.
    R2: The rotation matrix of the second camera.

    Returns:
    The translation vector of the second camera.
    '''
    """ ==========
    YOUR CODE HERE
    ========== """

    sz = len(R1)

    I = np.eye(sz)

    M1 = I - R2
    M2 = np.linalg.inv(I-R1)

    return M1.dot(M2).dot(t1)
```

In [22]:
```
def calcProjectiveTransformation(K1, R1, K2, R2):
    '''
    Calculates the planar projective transformation from cameras with the same camera cente
    This function determines the planar projective transformation from the image of a 3D po

    Args:
    K1: The calibration matrix of the first camera.
    R1: The rotation matrix of the first camera.
    K2: The calibration matrix of the second camera.
    R2: The rotation matrix of the second camera.

    Returns:
    The transformation matrix.
    '''
    """ ==========
    YOUR CODE HERE
    ========== """

    M1 = K1.dot(R1)
    M1 = np.linalg.inv(M1)
    M2 = K2.dot(R2)

    trans = M2.dot(M1)

    return trans
```

In [23]:
```
def calcDestinateK(srcK1, srcK2):
    '''
    Camera calibration matrix (same for both cameras)
    '''
    alpha = (srcK1[0][0] + srcK2[0][0] + srcK1[1][1] + srcK2[1][1]) // 4
    x0 = (srcK1[0][2] + srcK2[0][2]) // 2
    y0 = (srcK1[1][2] + srcK2[1][2]) // 2
    dstK = np.zeros((3, 3))
    dstK[0][0] = alpha
    dstK[0][2] = x0
    dstK[1][1] = alpha
    dstK[1][2] = y0
    dstK[2][2] = 1
    return dstK
```

In [24]:
```
def epipolarRecification(srcK1, srcR1, src_t1,
                         srcK2, srcR2, src_t2, dstR):
    '''
    Given two calibrated cameras, this function determines the calibration matrix and rota

    Args:
    srcK1: The calibration matrix of the first source camera.
    srcR1: The rotation matrix of the first source camera.
    src_t1: The translation vector of the first source camera.
    srcK2: The calibration matrix of the second source camera.
    srcR2: The rotation matrix of the second source camera.
    src_t2: The translation vector of the second source camera.
    dstR: The rotation matrix of the destination cameras.

    Returns:
    dstK: The calibration matrix of the destination cameras.
    dst_t1: The translation vector of the first destination camera.
    dst_t2: The translation vector of the second destination camera.
    H1, H2: The image rectification transformation matrices.
    '''

    dst_t1 = cameraTranslation(srcR1, src_t1, dstR)
    dst_t2 = cameraTranslation(srcR2, src_t2, dstR)

    dstK = calcDestinateK(srcK1, srcK2)

    H1 = calcProjectiveTransformation(srcK1, srcR1, dstK, dstR)
    H2 = calcProjectiveTransformation(srcK2, srcR2, dstK, dstR)

    return dstK, dst_t1, dst_t2, H1, H2
```

Problem 4.1.2 Warp Image [10 pts]

After calling $epipolarRectification$, we can get the projective transformation matrices $H1$ and $H2$. Next, we will geometrically transform (i.e., 'warp') the image so that the epipolar lines are image rows. You have to complete $warpImage$ using the backward method in Lecture 7. Note the destination images are required to be the same size as the source images.

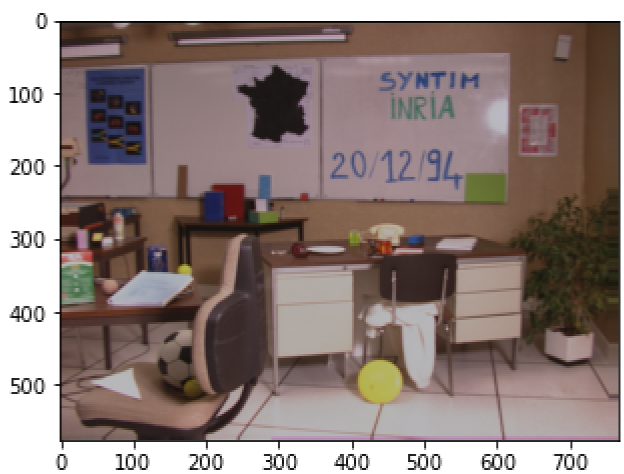
In [25]:
```
def warpImage(image, H, out_height, out_width):
    """
    Performs the warp of the full image content.
    Calculates bounding box by piping four corners through the transformation.

    Args:
    image: Image to warp
    H: The image rectification transformation matrices.
    out_height, out_width: The shape of output image.

    Returns:
    Out: An inverse warp of the image, given a homography.
    min_x, min_y, max_x, max_y: The minimum/maxmum of warped image bound.
    """
    """ ==========
    YOUR CODE HERE
    ========== """

    out = np.zeros((out_height,out_width,3),np.int64)
    min_x = out_height
    min_y = out_width
    max_x = 0
    max_y = 0
    for i in range(out_height):
        for j in range(out_width):
            x = H.dot(np.array([[i],[j],[1]]))[:,0]
            min_x = min(min_x, x[0])
            min_y = min(min_y, x[1])
            max_x = max(max_x, x[0])
            max_y = max(max_y, x[1])

            cx = int(round(x[0]))
            cy = int(round(x[1]))

            if cy < 0:
                cy = 0
            elif cy >= out_width:
                cy = out_width - 1
            if cx < 0:
                cx = 0
            elif cx >= out_height:
                cx = out_height - 1

            out[cx,cy,:]=image[i,j,:]

    return out, min_x, min_y, max_x, max_y
```

In [37]:
```
file_param = open('param.pkl', 'rb')
param = pickle.load(file_param)
file_param.close()
srcK1, srcR1, src_t1 = param['srcK1'], param['srcR1'], param['src_t1']
srcK2, srcR2, src_t2 = param['srcK2'], param['srcR2'], param['src_t2']
dstR = param['dstR']
```

In [38]:
```
dstK, dst_t1, dst_t2, H1, H2 = epipolarRecification(srcK1, srcR1, src_t1,
                                                    srcK2, srcR2, src_t2, dstR)
```

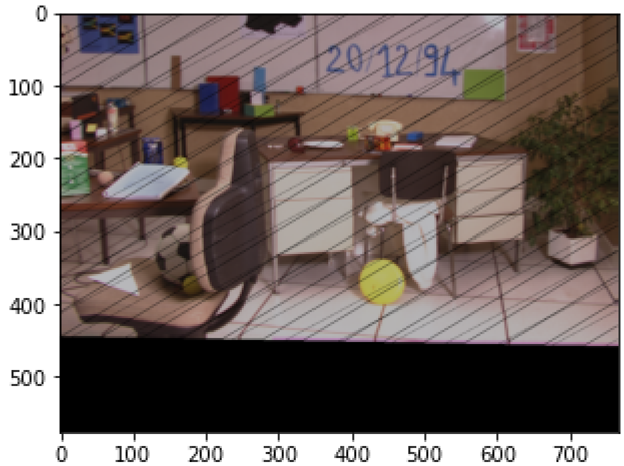
In [39]:
```
src1 = imread('Sport0_OG0.bmp')
plt.imshow(src1)
print('Original image 1:')
```

```
Original image 1:
```



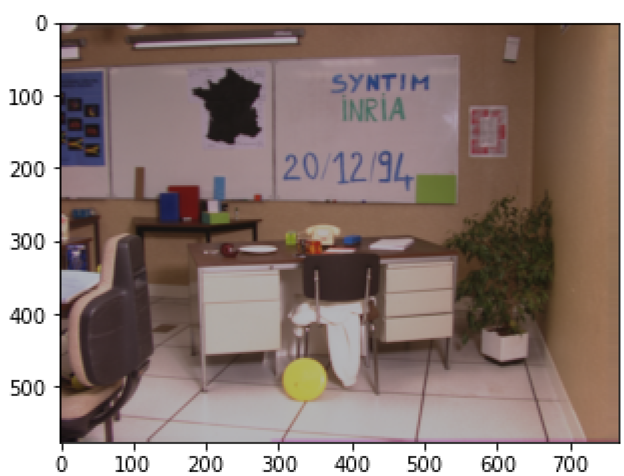
In [40]:
```
height1, width1, _ = src1.shape
rectified_im1_unbounded, min_x1, min_y1, max_x1, max_y1 = warpImage(src1, H1, height1, wid
plt.imshow(rectified_im1_unbounded)
print('Unbounded Rectified image 1:')
```

Unbounded Rectified image 1:



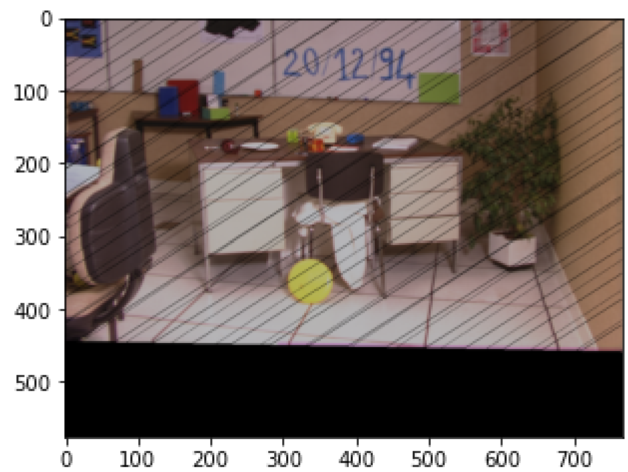
In [41]:
```
src2 = imread('Sport1_OGO.bmp')
plt.imshow(src2)
print('Original image 2:')
```

Original image 2:



In [42]:
```
height2, width2, _ = src2.shape
rectified_im2_unbounded, min_x2, min_y2, max_x2, max_y2 = warpImage(src2, H2, height2, wid
plt.imshow(rectified_im2_unbounded)
print('Unbounded Rectified image 2:')
```

```
Unbounded Rectified image 2:
```



4.1.3 Partial bounded retification [3 pts]

In the resulting images, although they are epipolar rectified, you should observe portions of the source images being transformed "out of bounds" of the destination images. To fix this problem, we can introduced a 2D transformation containing a translation (i.e., $T1$ and $T2$).

$$T1 = \begin{bmatrix} 1 & 0 & -min\_x1 - 0.5 \\ 0 & 1 & -min(min\_y1, min\_y2) - 0.5 \\ 0 & 0 & 1 \end{bmatrix}$$

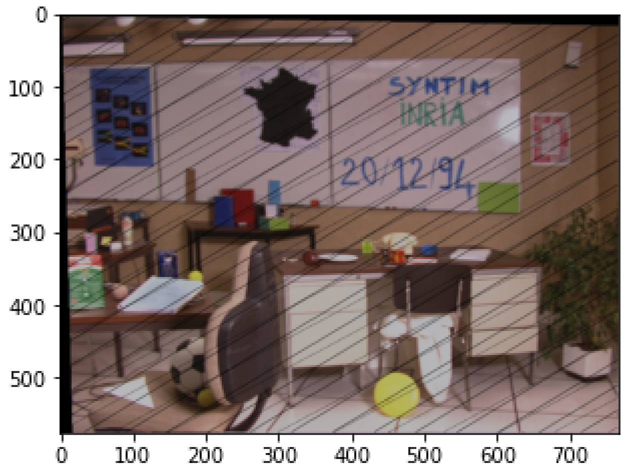
$$T2 = \begin{bmatrix} 1 & 0 & -min\_x2 - 0.5 \\ 0 & 1 & -min(min\_y1, min\_y2) - 0.5 \\ 0 & 0 & 1 \end{bmatrix}$$

$H1$, $H2$ can be updated by left multiplying $T1$, $T2$, respectivley. Again, geometrically tranform the images under the updated $H1$, $H2$. The destination image is required to be the same size as the source images. In the resulting images, although they are (still) epipolar rectified, you should observe the portions of the source images being transformed are no longer "out of bounds" on the top and left of the destination images.

In [26]:
```python
def partialboundedRetification(min_x1, min_y1, min_x2, min_y2, H1, H2):
    '''
    Update the projective transformation matries so that the rectified images are no longe
    '''
    """ ==========
    YOUR CODE HERE
    ========== """

    T1 = np.array([[1, 0, -min_x1-0.5],[0,1,-min(min_y1,min_y2)-0.5],[0,0,1]])
    T2 = np.array([[1, 0, -min_x2-0.5],[0,1,-min(min_y1,min_y2)-0.5],[0,0,1]])
    H1_bounded = H1.dot(T1)
    H2_bounded = H2.dot(T2)

    return H1_bounded, H2_bounded
```

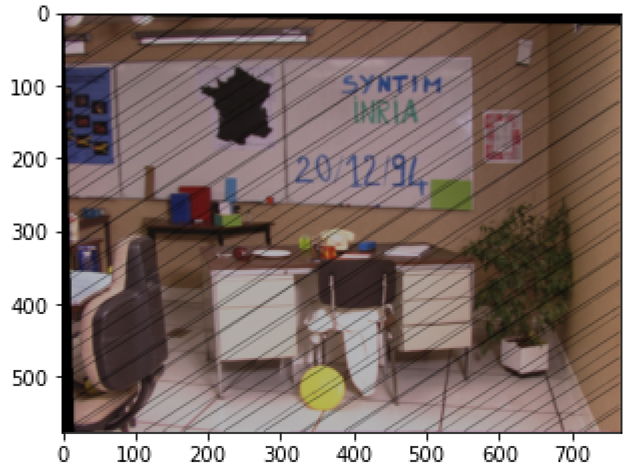
In [43]:
```python
H1_bounded, H2_bounded = partialboundedRetification(min_x1, min_y1, min_x2, min_y2, H1, H2
rectified_im1_bounded, min_x1_bounded, min_y1_bounded, max_x1_bounded, max_y1_bounded = wa
plt.imshow(rectified_im1_bounded)
print('The partial bounded rectified image 1:')
```

The partial bounded rectified image 1:



In [44]:
```
rectified_im2_bounded, min_x2_bounded, min_y2_bounded, max_x2_bounded, max_y2_bounded = wa
plt.imshow(rectified_im2_bounded)
print('The partial bounded rectified image 2:')
```

```
The partial bounded rectified image 2:
```



4.1.4 Completely bounded rectification [3 pts]

Finally, determine the size of the destination images that completely bound the transformed images. Again geometrically transform the images under the updated 2D projective transformation matrices $H1$ and $H2$ (these are not updated a second time). You should complete the function $completelyBoundedRectification$. The destination images are required to be the size you just calculated. In the resulting images, you should observe the source images being transformed such that they are epipolar rectified and are completely bounded.

In [27]:
```
def completelyBoundedRectification(src1, src2, H1_bounded, H2_bounded,
                                   min_x1_bounded, max_x1_bounded,
                                   min_y1_bounded, max_y1_bounded):
    '''
    Determine the size of the destination images (same size for both) that completely boun
    '''
    """ ==========
    YOUR CODE HERE
    ========== """

    min_x1_bounded = int(min_x1_bounded) - 1
    max_x1_bounded = int(max_x1_bounded) + 1
    min_y1_bounded = int(min_y1_bounded) - 1
    max_y1_bounded = int(max_y1_bounded) + 1

    h = max_x1_bounded - min_x1_bounded + 1
    w = max_y1_bounded - min_y1_bounded + 1

    rectified_im1_final = np.zeros((h, w, 3),np.int64)
    rectified_im2_final = np.zeros((h, w, 3),np.int64)

    h1, w1, _ = src1.shape
    for i in range(h1):
        for j in range(w1):
            x = H1_bounded.dot(np.array([[i],[j],[1]]))[:,0]

            cx = int(x[0])-min_x1_bounded
            cy = int(x[1])-min_y1_bounded

            rectified_im1_final[cx,cy,:]=src1[i,j,:]

    h2, w2, _ = src2.shape
    for i in range(h2):
        for j in range(w2):
            x = H2_bounded.dot(np.array([[i],[j],[1]]))[:,0]

            cx = int(x[0])-min_x1_bounded
            cy = int(x[1])-min_y1_bounded

            if cx >= h:
                cx = h-1
            if cy >= w:
                cy = w-1

            rectified_im2_final[cx,cy,:]=src2[i,j,:]

    return rectified_im1_final, rectified_im2_final
```

In [65]:
```
rectified_im1_final, rectified_im2_final = completelyBoundedRectification(src1, src2, H1_b
                                   min_x1_bounded, max_x1_bounded,
                                   min_y1_bounded, max_y1_bounded)
```

Problem 4.2 Feature matching [3 pts]

4.2.1 SSD (Sum Squared Distance) Matching [1 pts]

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

In [28]:
```
def ssdMatch(img1, img2, c1, c2, R):
    """Compute SSD 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:
        SSD matching score for two input windows.

    """
    """ ==========
    YOUR CODE HERE
    ========== """

    matching_score = 0
    for i in range(-R,R+1):
        for j in range(-R,R+1):
            dist = img1[c1[1]+i,c1[0]+j]-img2[c2[1]+i,c2[0]+j]
            matching_score += dist * dist

    return matching_score
```

In [19]:
```
# Here is the code for you to test your implementation
img1 = np.array([[1, 2, 3, 4], [4, 5, 6, 8], [7, 8, 9, 4]])
img2 = np.array([[1, 2, 1, 3], [6, 5, 4, 4], [9, 8, 7, 3]])
print(ssdMatch(img1, img2, np.array([1, 1]), np.array([1, 1]), 1))
# should print 20
print(ssdMatch(img1, img2, np.array([2, 1]), np.array([2, 1]), 1))
# should print 30
print(ssdMatch(img1, img2, np.array([1, 1]), np.array([2, 1]), 1))
# should print 46
```

```
20
30
46
```

Problem 4.2.2 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 [85]:
```
def normalize_window(window):
    _mean = np.mean(window)
    _stdev = np.sqrt(np.sum((window - _mean) ** 2))
    return (window - _mean) / (_stdev + 1e-6)

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

    Args:
        img1: Image 1.
        img2: Image 2.
        c1: Center (in image coordinate) of the window in image 1.
        c2: Center (in image coordinate) of the window in image 2.
        R: R is the radius of the patch, 2 * R + 1 is the window size

    Returns:
        NCC matching score for two input windows.

    """

    """ ==========
    YOUR CODE HERE
    ========== """

    mean1 = 0
    mean2 = 0
    sz = 2*R+1
    for i in range(-R,R+1):
        for j in range(-R,R+1):
            mean1 += img1[c1[1]+i, c1[0]+j]/(sz*sz)
            mean2 += img2[c2[1]+i, c2[0]+j]/(sz*sz)

    d_1 = 0
    d_2 = 0
    for i in range(-R,R+1):
        for j in range(-R,R+1):
            t_1 = img1[c1[1]+i, c1[0]+j]-mean1
            d_1 += t_1 * t_1
            t_2 = img2[c2[1]+i, c2[0]+j]-mean2
            d_2 += t_2 * t_2

    d_1 = np.sqrt(d_1)
    d_2 = np.sqrt(d_2)

    w_1 = (img1[-R+c1[1]:R+c1[1]+1,-R+c1[0]:R+c1[0]+1]-mean1)/d_1
    w_2 = (img2[-R+c2[1]:R+c2[1]+1,-R+c2[0]:R+c2[0]+1]-mean2)/d_2

    ans = 0
    for i in range(sz):
        for j in range(sz):
            ans += w_1[i, j] * w_2[i, j]

    return ans
```

In [21]:
```
# 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
```

Problem 4.3 Naive Matching [8 pts]

Equipped with the corner detector and the NCC matching function, we are ready to start finding correspondences. One naive strategy is to try and find the best match between the two sets of corner points. Write a script that does this, namely, for each corner in image1, find the best match from the detected corners in image2 (or, if the NCC match score is too low, then return no match for that point). You will have to figure out a good threshold (NCCth) value by experimentation.

Write a function `naive_matching` and call it as below. Examine your results for 10, 20, and 30 detected corners in each image. Choose the number of detected corners to maximize the number of correct matching pairs. `naive_matching` will call your NCC matching code.

Properly label or mention which output corresponds to which choice of number of corners. The total number of outputs is 6 images: (3 choices of number of corners for each of `matrix` and `warrior`), where each figure might look like the following:

Number of corners: 10

dino match

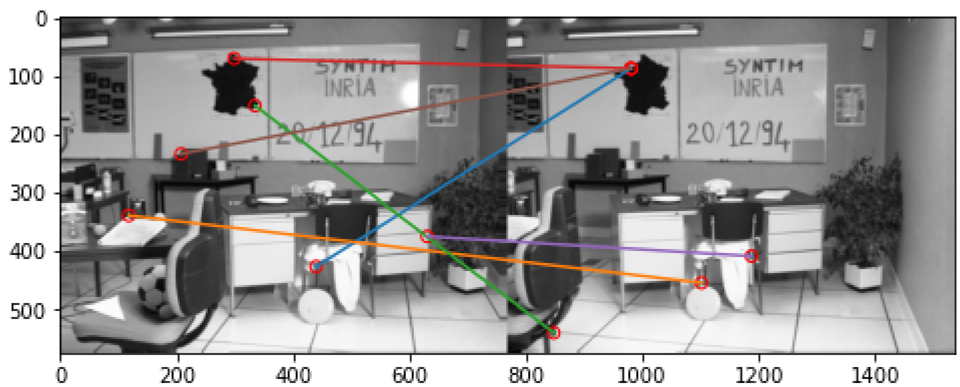
In [30]:
```
def naive_matching(img1, img2, corners1, corners2, R, NCCth):
    """Compute NCC given two windows.

    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 = []

    h1, w1 = img1.shape
    h2, w2 = img2.shape

    def is_valid(h,w,x,y):
        if x-R<0 or x+R+1>w:
            return False
        if y-R<0 or y+R+1>h:
            return False
        return True

    for x1,y1 in corners1:
        if not is_valid(h1, w1, x1, y1):
            continue
        bx = 0
        by = 0
        bscore = 0
        for x2,y2 in corners2:
            if not is_valid(h2, w2, x2, y2):
                continue
            score = ncc_match(img1,img2,np.array([x1,y1],np.int64),np.array([x2,y2],np.int
            if bscore < score:
                bscore = score
                bx = x2
                by = y2
        if bscore > NCCth:
            matching.append(([x1,y1],[bx,by]))

    return matching
```

In [123]:
```
# detect corners on warrior and matrix sets
# you are free to modify code here, create your helper functions, etc.

nCorners = 50  # do this for 10, 20 and 30 corners
smoothSTD = 1
windowSize = 17

# read images and detect corners on images

imgs_sport = []
crns_sport = []

for i in range(2):
    img_sport = imageio.imread('Sport' + str(i) + '_OG0.bmp')
    imgs_sport.append(rgb2gray(img_sport))
    # downsize your image in case corner_detect runs slow in test
    # imgs_mat.append(rgb2gray(img_mat)[::2, ::2])
    crns_sport.append(corner_detect(imgs_sport[i], nCorners, smoothSTD, windowSize))
```

In [125]:
```
# match corners
R = 16
NCCth = 0.7  # put your threshold here
matching_sport = naive_matching(imgs_sport[0]/255,
                                imgs_sport[1]/255,
                                crns_sport[0],
                                crns_sport[1],
                                R, NCCth)
```

In [126]:
```
# plot matching result
def show_matching_result(img1, img2, matching):
    """ ==========
    YOUR CODE HERE
    ========== """
    img = np.hstack([img1,img2])
    h1,w1 = img1.shape
    h2,w2 = img2.shape
    print(img.shape)
    fig = plt.figure(figsize=(8, 8))
    ax = fig.add_subplot()
    x = []
    y = []
    for node1, node2 in matching:
        x.append(node1[0])
        y.append(node1[1])
        x.append(node2[0]+w1)
        y.append(node2[1])
        ax.plot([node2[0]+w1,node1[0]],[node2[1],node1[1]])
    ax.imshow(img, cmap='gray')
    ax.scatter(x, y, s=35, edgecolors='r', facecolors='none')

print("Number of Corners:", nCorners)
show_matching_result(imgs_sport[0], imgs_sport[1], matching_sport)
```

```
Number of Corners: 50
(576, 1536)
```



Problem 4.4 Matching using epipolar geometry [10 pts]

Next, we will 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 line search 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.

In [125]:
```
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 want to refer to the `show_matching_result` function.
    ========== """
    img = np.hstack([img1,img2])
    h1,w1,_ = img1.shape
    h2,w2,_ = img2.shape
    fig = plt.figure(figsize=(8, 8))
    ax = fig.add_subplot()
    x = []
    y = []
    for node1, node2 in corrs:
        x.append(node1[0])
        y.append(node1[1])
        x.append(node2[0]+w1)
        y.append(node2[1])
        ax.plot([node2[0]+w1,node1[0]],[node2[1],node1[1]])
    ax.imshow(img, cmap='gray')
    ax.scatter(x, y, s=35, edgecolors='r', facecolors='none')


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

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

    Returns:
        Matching result to be used in display_correspondence function
    """

    """ ==========
    YOUR CODE HERE
    ========== """

    h2, w2,_ = img2.shape

    matching = []
    for x, y in corners1:
        if x < R or x >= w2 + R - 1 or y < R or y >= h2 + R - 1:
            continue
        bscore = 0
```

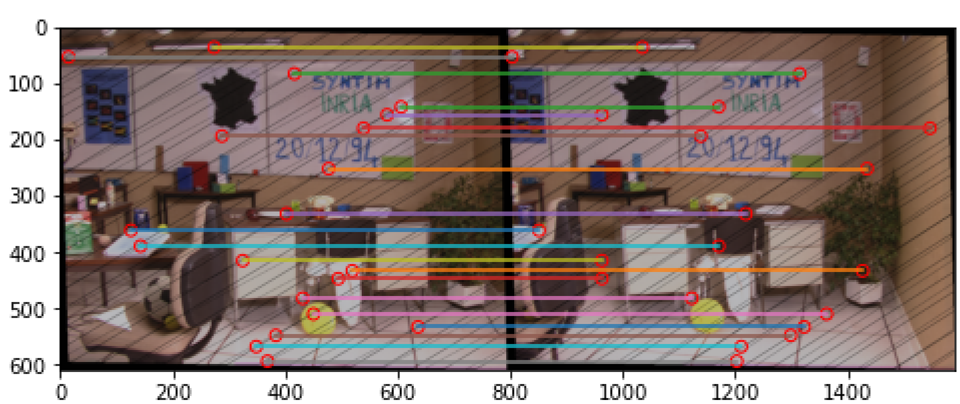
```
        bx = 0
        by = 0
        for i in range(R, w2-R-1,R):
            score = ncc_match(rgb2gray(img1), rgb2gray(img2), np.array([x,y],np.int64), np
            if score > bscore:
                bscore = score
                bx = i
                by = y
        if bscore < NCCth:
            matching.append(([x,y],[bx,by]))

    return matching
```

In [45]:

```
rectified_im1_final, rectified_im2_final = completelyBoundedRectification(src1, src2,
                                                                          H1_bounded, H2_bo
                                                                          min_x1_bounded,
                                                                          max_x1_bounded,
                                                                          min_y1_bounded,
                                                                          max_y1_bounded)
```

In [135]:

```
# replace black pixels with white pixels
_black_idxs = (rectified_im1_final[:, :, 0] == 0) & (rectified_im1_final[:, :, 1] == 0) &
rectified_im1_final[:, :][_black_idxs] = [1.0, 1.0, 1.0]
_black_idxs = (rectified_im2_final[:, :, 0] == 0) & (rectified_im2_final[:, :, 1] == 0) &
rectified_im2_final[:, :][_black_idxs] = [1.0, 1.0, 1.0]

nCorners = 20
# Choose your threshold and NCC matching window radius
smoothSTD = 1
windowSize = 7
NCCth = 0.9
R = 8
# detect corners using corner detector here, store in corners1
corners1 = corner_detect(rgb2gray(rectified_im1_final), nCorners, smoothSTD, windowSize)
corrs = correspondence_matching_epipole(rectified_im1_final, rectified_im2_final, corners1
display_correspondence(rectified_im1_final, rectified_im2_final, corrs)
```



In [ ]: