HW3

November 14, 2020

1 CSE 252A (Computer Vision I) · Fall 2020 · Assignment 3

1.0.1 Instructor: David Kriegman

1.0.2 Assignment published on Sunday, November 15, 2020

1.0.3 Due on Sunday, November 29, 2020 at 11:59 pm Pacific Time

---

1.1 Instructions

- Review the academic integrity and collaboration policies on the course website.
- This assignment must be completed individually.
- This assignment contains theoretical and programming exercises. If you plan to submit handwritten answers for the theoretical exercises, please be sure that your writing is readable (illegible answers will not be given the benefit of the doubt!) and merge your handwritten solutions in problem order with the PDF that you create from this notebook. You can also write the answers within the notebook itself by creating Markdown cells.
- Programming aspects of this assignment must be completed using Python in this notebook.
- If you want to modify the skeleton code, you can do so. The existing code is merely meant to provide you with a framework for your solution.
- You may use Python packages for basic linear algebra (you can use NumPy or SciPy for basic operations), but you may not use packages that directly solve the problem.
- If you are unsure about using a specific package or function, then ask the instructor and teaching assistants for clarification.
- You must submit to Gradescope:
  - (1) This notebook exported as a `.pdf` file (including any handwritten solutions scanned and merged into the PDF, if applicable).
  - (2) This notebook as an `.ipynb` file.
- You must select the pages associated with each problem on Gradescope (for your PDF submission).
- **Late policy:** Assignments submitted late will receive a 10% grade reduction for each day late (e.g. an assignment submitted an hour after the due date will receive a 10% penalty, an assignment submitted 10 hours after the due date will receive a 10% penalty, and an assignment submitted 28 hours after the due date will receive a 20% penalty). Assignments will not be accepted 72 hours after the due date. If you require an extension (for personal reasons only), you must request one as far in advance as possible. Extensions requested close to or after the due date will only be granted for clear emergencies or clearly unforeseeable circumstances.

1.2 Problem 1: Edge Detection (11 points)

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

- **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 5x5 Gaussian kernel filter with $\sigma = 1.4$ to smooth the images.
- **Gradient Computation [2 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 $G_\theta = \tan^{-1}\left(\frac{G_y}{G_x}\right)$.
- **Non-Maximum Suppression [4 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.
- **Hysteresis Thresholding [4 pts]:** If we use NMS output as a mask to detect edges i.e. `nms_edges = nms > 0; plt.imshow(nms_edges)`, we see that too many edges (including many stray edges) are discovered. To circumvent this, choose suitable values of thresholds and apply the hysteresis thresholding approach described in lecture. This will remove the edges caused by noise and color variations. The following algorithm might be helpful:
  - 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 from these starting points).

Compute the images after each step and select suitable thresholds that retain most of the true edges. As depicted in the example below, show each of the intermediate steps and label your images accordingly.

In total, there should be five output images (original, smoothed, gradient magnitude, NMS edges, final edges).

For this question, use the image `sio_pier.jpg`.

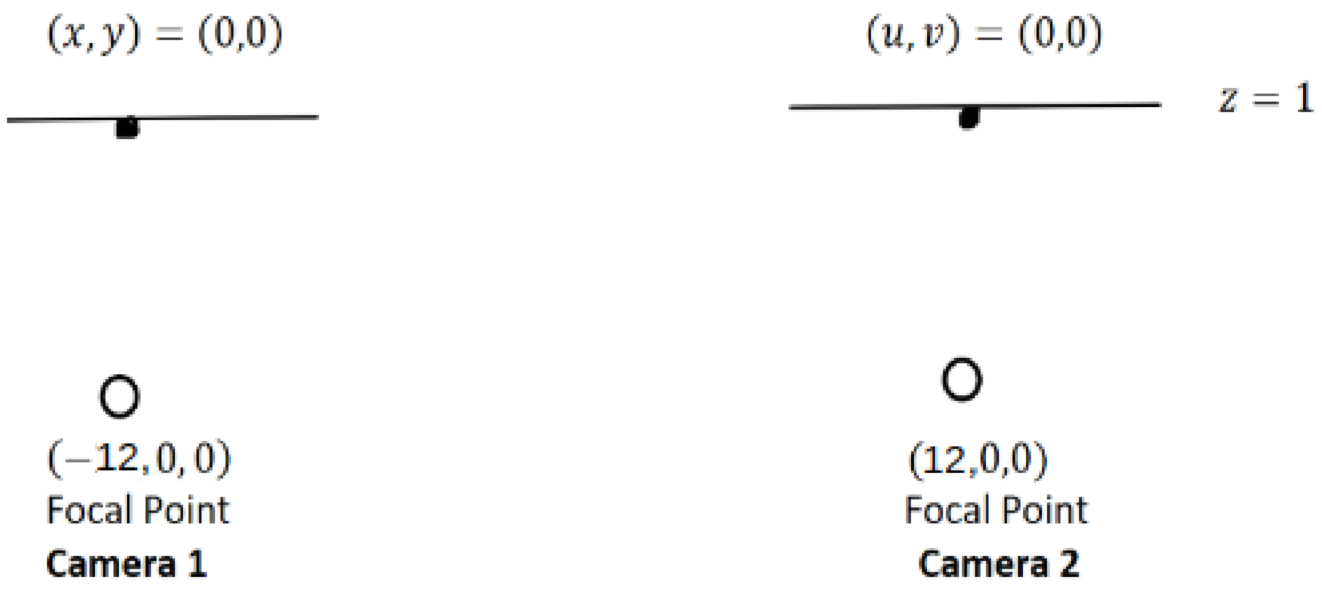
In the figure above: (a) Smoothed image , (b) Gradient magnitude , (c) NMS edges , (d) Final edges after thresholding

```
In [ ]: 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]

def gaussian2d(filter_size=5, sig=1.0):
    """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)

def smooth(image):
    """ ==========
    YOUR CODE HERE
    ========== """
    return image

def gradient(image):
    """ ==========
    YOUR CODE HERE
    ========== """
    g_mag = np.zeros_like(image)
    g_theta = np.zeros_like(image)
    return g_mag, g_theta

def nms(g_mag, g_theta):
    """ ==========
    YOUR CODE HERE
    ========== """
    nms_response = np.zeros_like(g_mag)
    return nms_response

def hysteresis_threshold(image, g_theta):
    """ ==========
    YOUR CODE HERE
    ========== """
    return image

def edge_detect(image):
    """Perform edge detection on the image."""
    smoothed = smooth(image)
    g_mag, g_theta = gradient(smoothed)
    nms_image = nms(g_mag, g_theta)
    thresholded = hysteresis_threshold(nms_image, g_theta)
    return smoothed, g_mag, nms_image, thresholded

# 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 versions'

# Perform edge detection
smoothed, g_mag, nms_image, thresholded = edge_detect(image)

print('Original:')
plt.imshow(image, cmap=cm.gray)
plt.show()

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

print('Gradient magnitude:')
plt.imshow(g_mag, cmap=cm.gray)
plt.show()

print('NMS:')
plt.imshow(nms_image, cmap=cm.gray)
plt.show()

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

1.3 Problem 2: Epipolar Geometry (6 points)

Consider two cameras whose image planes are the z=1 plane, and whose focal points are at (-12, 0, 0) and (12, 0, 0). We'll call a point in the first camera (x, y), and a point in the second camera (u, v). Points in each camera are relative to the camera center. So, for example if (x, y) = (0, 0), this is really the point (-12, 0, 1) in world coordinates, while if (u, v) = (0, 0) this is the point (12, 0, 1).



a) Suppose the point (x, y) = (15, 2) is matched to the point (u, v) = (1, 2). What is the 3D location of this point?

b) Consider points that lie on the line x + z = 0, y = 0. Use the same stereo set up as before. Write an analytic expression giving the disparity of a point on this line after it projects onto the two images, as a function of its position in the right image. So your expression should only involve the variables u and d (for disparity). Your expression only needs to be valid for points on the line that are in front of the cameras, i.e. with z > 1.

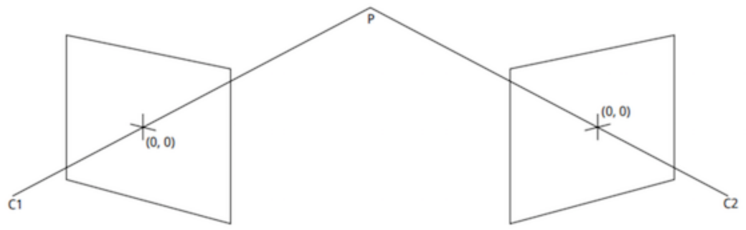


Figure 1: fig1

1.4 Problem 3: The Epipolar Constraint (4 points)

Suppose two cameras fixate on a point P in space such that their principal axes intersect at that point. (See the figure below.) Show that if the image coordinates are normalized so that the coordinate origin (0, 0) coincides with the principal point, then the F_{33} element of the fundamental matrix is zero.

In the figure, $C1$ and $C2$ are the optical centers. The principal axes intersect at point P.

1.5 Problem 4: Sparse Stereo Matching (32 points)

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. Each file contains two images, two camera matrices, and two sets of corresponding points (extracted by manually clicking the images). For illustration, we have run our code on a third image pair (`dino1.png`, `dino2.png`). This data is also provided for you to debug your code, but **you should only report results for `warrior` and `matrix`**. In other words, where we include one (or a pair) of images in the assignment below, you will provide the same thing but for BOTH `warrior` and `matrix`. Note that the `matrix` image pair is harder, in the sense that the matching algorithms we are implementing will not work quite as well. You should expect good results, however, on `warrior`.

1.5.1 Corner Detection (8 points)

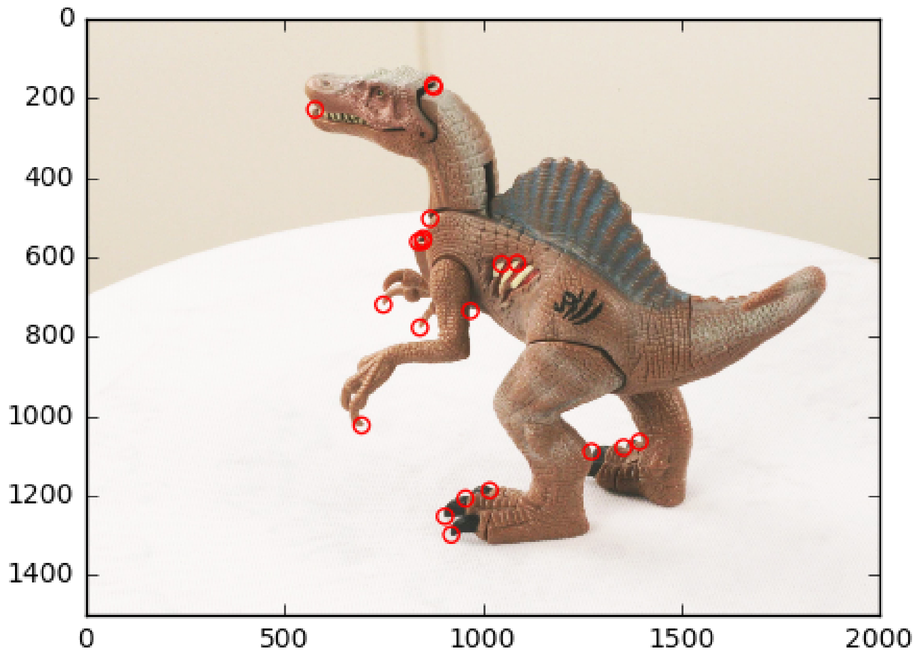
The first thing we need to do is to build a corner detector. This should be done according to the lecture slides. You should fill in the function corner_detect below, which takes as input image, nCorners, smoothSTD, windowSize – where `smoothSTD` is the standard deviation of the smoothing kernel and `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.

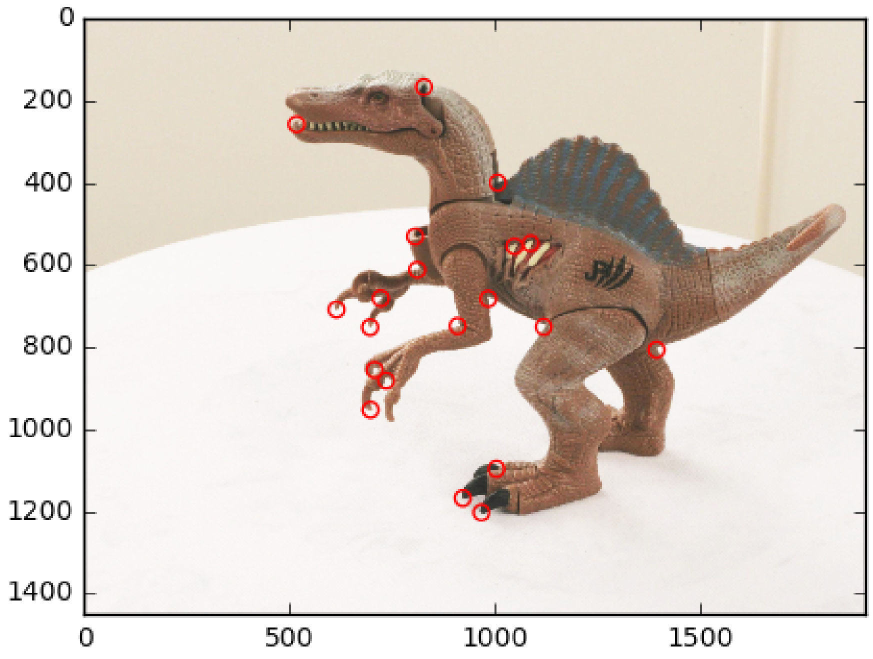
Note 1: careful with your padding! Be aware that zero padding can affect whether corners are detected around the borders of the image, i.e. you might end up with false corners because of large discontinuities between these artificial black regions and the actual image content. To avoid this happening, feel free to ignore `windowSize // 2` pixels on all sides of the image when considering candidate locations for corners, or to try using symmetric padding.

Note 2: you are allowed to use `scipy.signal.convolve`, `scipy.ndimage.maximum_filter`, `np.pad`, and `scipy.ndimage.filters.gaussian_filter` for this problem.

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.





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

```
In [ ]: 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 [ ]: 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
            ========== """
            corners = np.zeros((nCorners, 2))
            return corners
```

```
In [ ]: 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('p4/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('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))

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

1.5.2 NCC (Normalized Cross-Correlation) Matching (2 points)

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 [ ]: 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
        return matching_score
```

```
In [ ]: # 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
```

1.5.3 Naive Matching (4 points)

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

```
In [ ]: 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 = []
            return matching
```

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

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

        # 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 [ ]: # match corners
        R = 120
        NCCth = 0.6  # put your threshold here
        matching_mat = naive_matching(imgs_mat[0]/255, imgs_mat[1]/255, crns_mat[0], crns_mat[1], R, NCC
        matching_war = naive_matching(imgs_war[0]/255, imgs_war[1]/255, crns_war[0], crns_war[1], R, NCC

In [ ]: # plot matching result
        def show_matching_result(img1, img2, matching):
            fig = plt.figure(figsize=(8, 8))
            plt.imshow(np.hstack((img1, img2)), cmap='gray') # two dino images are of different sizes, 
            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()

        print("Number of Corners:", nCorners)
        show_matching_result(imgs_mat[0], imgs_mat[1], matching_mat)
        show_matching_result(imgs_war[0], imgs_war[1], matching_war)

In [ ]: # =====================================================
        # ALSO SHOW RESULTS FOR DIFFERENT # OF DETECTED CORNERS
        # =====================================================

        for nCorners in (10, 20, 30):
            # 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))

# match corners
R = 120 if nCorners != 30 else 80
NCCth = 0.6
matching_mat = naive_matching(imgs_mat[0]/255, imgs_mat[1]/255, crns_mat[0], crns_mat[1], R
matching_war = naive_matching(imgs_war[0]/255, imgs_war[1]/255, crns_war[0], crns_war[1], R

print('nCorners: %r' % nCorners)
show_matching_result(imgs_mat[0], imgs_mat[1], matching_mat)
show_matching_result(imgs_war[0], imgs_war[1], matching_war)
```

1.5.4 Epipolar Geometry (4 points)

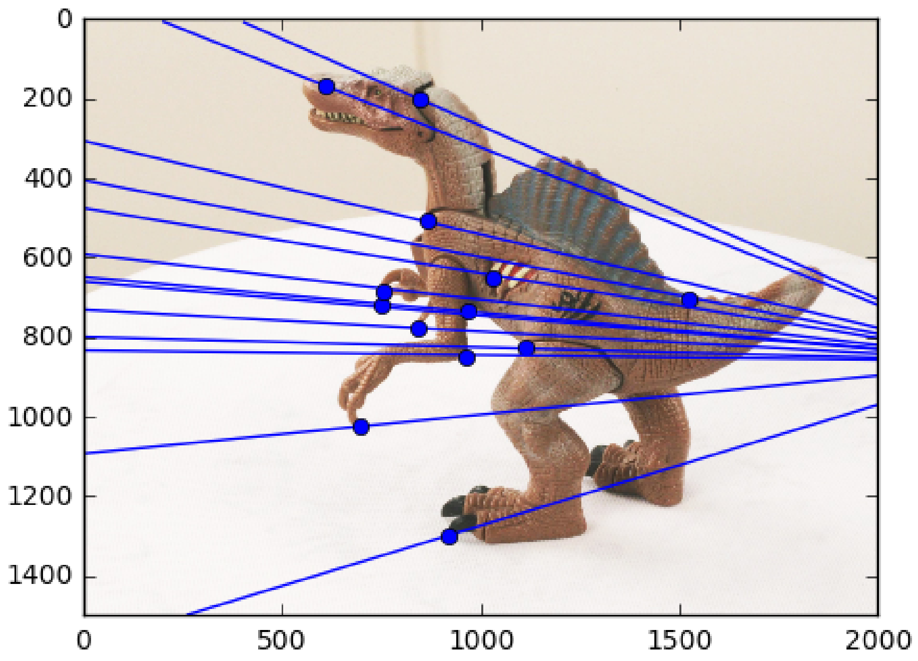
Complete the compute_fundamental function below using the 8-point algorithm described in lecture. Using the fundamental_matrix function and the corresponding points provided in `cor1.npy` and `cor2.npy`, calculate the fundamental matrix for the `matrix` and `warrior` image sets. Note that the normalization of the corner points is handled in the fundamental_matrix function.

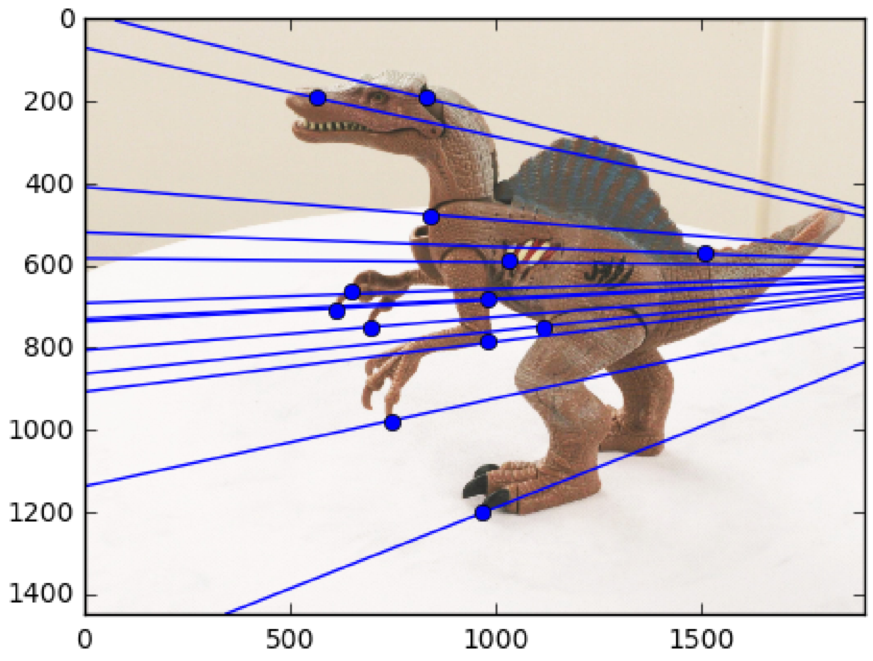
```
In [ ]: import numpy as np
        from imageio import imread
        import matplotlib.pyplot as plt
        from scipy.io import loadmat
        from numpy.linalg import svd, eig

        def compute_fundamental(x1, x2):
            """ Computes the fundamental matrix from corresponding points
                (x1,x2 3*n arrays) using the 8 point algorithm.

                Construct the A matrix according to lecture
                and solve the system of equations for the entries of the fundamental matrix.

                Returns:
                Fundamental Matrix (3x3)
            """

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

```
    # return your F matrix
    pass


def fundamental_matrix(x1,x2):
    # Normalization of the corner points is handled here
    n = x1.shape[1]
    if x2.shape[1] != n:
        raise ValueError("Number of points don't match.")

    # normalize image coordinates
    x1 = x1 / x1[2]
    mean_1 = np.mean(x1[:2],axis=1)
    S1 = np.sqrt(2) / np.std(x1[:2])
    T1 = np.array([[S1,0,-S1*mean_1[0]],[0,S1,-S1*mean_1[1]],[0,0,1]])
    x1 = np.dot(T1,x1)

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

    # compute F with the normalized coordinates
    F = compute_fundamental(x1,x2)

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

    return F/F[2,2]
```

1.5.5 Plot Epipolar Lines (5 points)

Using this fundamental matrix, plot the epipolar lines in both images for each image pair. For this part, you will want to complete the function plot_epipolar_lines. Show your result for matrix and warrior as exemplified by the figure below.





Also complete the function to calculate the epipoles for a given fundamental matrix.

```
In [ ]: def compute_epipole(F):
            """
            This function computes the epipoles for a given fundamental matrix.

            input:
              F --> fundamental matrix
            output:
              e1 --> corresponding epipole in image 1
              e2 --> epipole in image2
```

```
        """

        """ ==========
        YOUR CODE HERE
        ========== """
        return e1, e2


    def plot_epipolar_lines(img1, img2, cor1, cor2):
        """Plot epipolar lines on image given image, corners

        Args:
            img1: Image 1.
            img2: Image 2.
            cor1: Corners in homogeneous image coordinate in image 1 (3xn)
            cor2: Corners in homogeneous image coordinate in image 2 (3xn)

        """

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

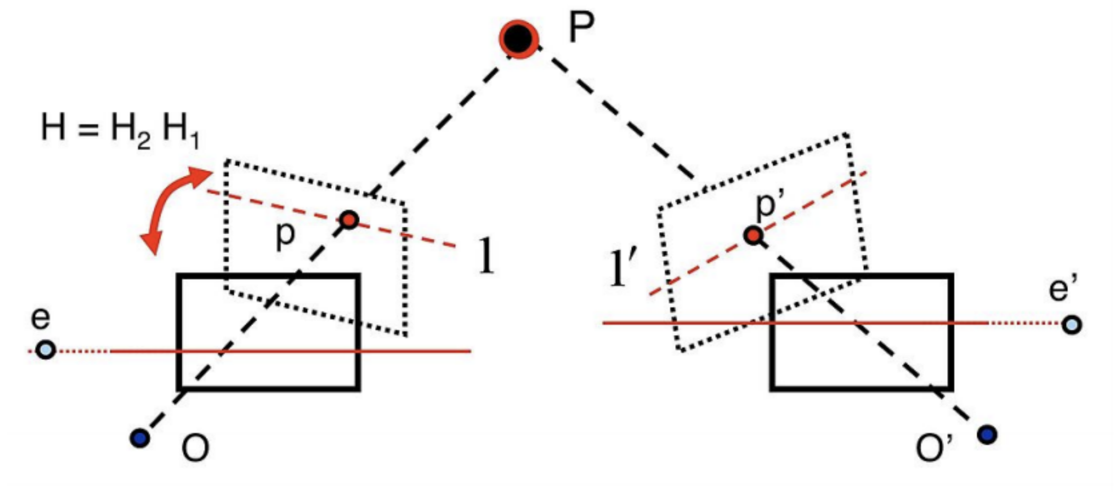
```
In [ ]: # replace images and corners with those of matrix and warrior

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

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

1.5.6 Image Rectification (5 points)

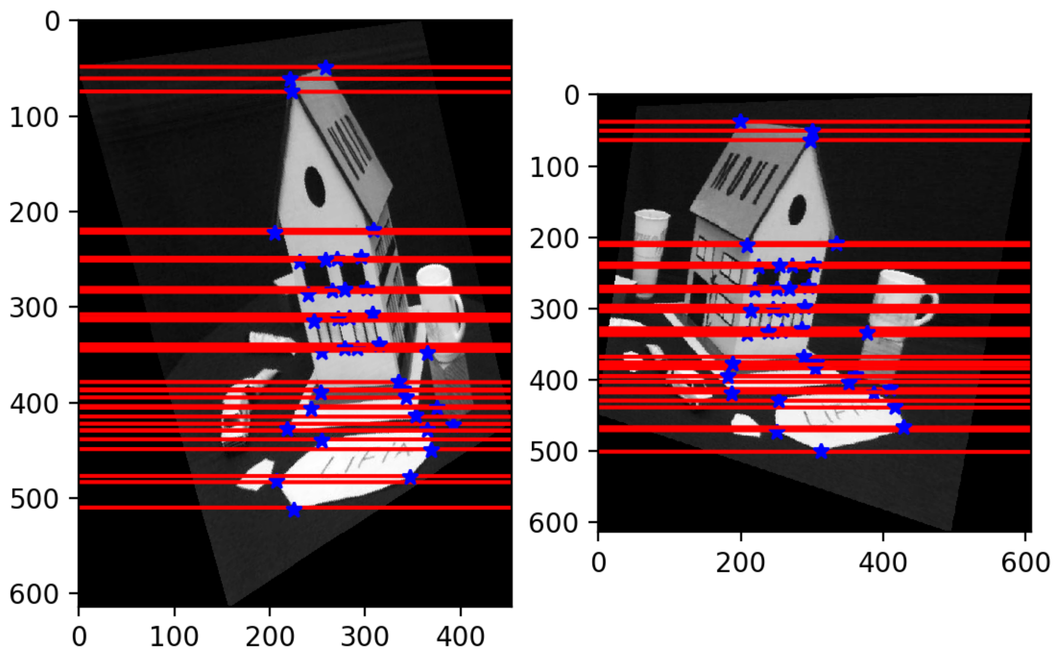
An interesting case for epipolar geometry occurs when two images are parallel to each other. In this case, there is no rotation component involved between the two images and the essential matrix is $\mathtt{E} = [\boldsymbol{T_x}]\boldsymbol{R} = [\boldsymbol{T_x}]$. Also if you observe the epipolar lines $\boldsymbol{l}$ and $\boldsymbol{l}^{'}$ for parallel images, they are horizontal and consequently, the corresponding epipolar lines share the same vertical coordinate. Therefore the process of making images parallel becomes useful while discerning the relationships between corresponding points in images.

Rectifying a pair of images can also be done for uncalibrated camera images (i.e. we do not require the camera matrix of intrinsic parameters). Using the fundamental matrix we can find the pair of epipolar lines $\boldsymbol{l_i}$ and $\boldsymbol{l_i^{'}}$ for each of the correspondences. The intersection of these lines will give us the respective epipoles $\boldsymbol{e}$ and $\boldsymbol{e}^{'}$. Now to make the epipolar lines to be parallel we need to map the epipoles to infinity. Hence, we need to find a homography that maps the epipoles to infinity.

The method to find the homography has already 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_epipole` 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. Below is an example of the output for this part:



```
In [ ]: def compute_matching_homographies(e2, F, im2, points1, points2):
            """This function computes the homographies to get the rectified images.

            input:
              e2 --> epipole in image 2
              F --> the fundamental matrix (think about what you should be passing: F or F.T!)
              im2 --> image2
              points1 --> corner points in image1
              points2 --> corresponding corner points in image2
```

```
output:
  H1 --> homography for image 1
  H2 --> homography for image 2
"""
# calculate H2
width = im2.shape[1]
height = im2.shape[0]

T = np.identity(3)
T[0][2] = -1.0 * width / 2
T[1][2] = -1.0 * height / 2

e = T.dot(e2)
e1_prime = e[0]
e2_prime = e[1]
if e1_prime >= 0:
    alpha = 1.0
else:
    alpha = -1.0

R = np.identity(3)
R[0][0] = alpha * e1_prime / np.sqrt(e1_prime**2 + e2_prime**2)
R[0][1] = alpha * e2_prime / np.sqrt(e1_prime**2 + e2_prime**2)
R[1][0] = - alpha * e2_prime / np.sqrt(e1_prime**2 + e2_prime**2)
R[1][1] = alpha * e1_prime / np.sqrt(e1_prime**2 + e2_prime**2)

f = R.dot(e)[0]
G = np.identity(3)
G[2][0] = - 1.0 / f

H2 = np.linalg.inv(T).dot(G.dot(R.dot(T)))

# calculate H1
e_prime = np.zeros((3, 3))
e_prime[0][1] = -e2[2]
e_prime[0][2] = e2[1]
e_prime[1][0] = e2[2]
e_prime[1][2] = -e2[0]
e_prime[2][0] = -e2[1]
e_prime[2][1] = e2[0]

v = np.array([1, 1, 1])
M = e_prime.dot(F) + np.outer(e2, v)

points1_hat = H2.dot(M.dot(points1.T)).T
points2_hat = H2.dot(points2.T).T

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

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

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


def image_rectification(im1, im2, points1, points2):
    """This function provides the rectified images along with the new corner points as outputs
    images with corner correspondences
    input:
    im1--> image1
    im2--> image2
    points1--> corner points in image1
    points2--> corner points in image2
    outpu:
    rectified_im1-->rectified image 1
    rectified_im2-->rectified image 2
    new_cor1--> new corners in the rectified image 1
    new_cor2--> new corners in the rectified image 2
    """

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

    return rectified_im1, rectified_im2, new_cor1, new_cor2


# Plot the parallel epipolar lines using plot_epipolar_lines
for subj in ('matrix', 'warrior'):
    I1 = imread("./p4/%s/%s0.png" % (subj, subj))
    I2 = imread("./p4/%s/%s1.png" % (subj, subj))

    cor1 = np.load("./p4/%s/cor1.npy" % subj)
    cor2 = np.load("./p4/%s/cor2.npy" % subj)

    rectified_im1, rectified_im2, new_cor1, new_cor2 = image_rectification(I1, I2, cor1, cor2)
    plot_epipolar_lines(rectified_im1, rectified_im2, new_cor1, new_cor2)
```

1.5.7 Matching Using Epipolar Geometry (4 points)

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 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. Execute this process for the `warrior` and `matrix` image sets. **In total, you should have two output images for this part.**

```
In [ ]: 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.
            ========== """

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

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

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

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

```
In [ ]: I1 = imageio.imread("./p4/matrix/matrix0.png")
        I2 = imageio.imread("./p4/matrix/matrix1.png")
        cor1 = np.load("./p4/matrix/cor1_alt.npy")
        cor2 = np.load("./p4/matrix/cor2_alt.npy")
        I3 = imageio.imread("./p4/warrior/warrior0.png")
        I4 = imageio.imread("./p4/warrior/warrior1.png")
        cor3 = np.load("./p4/warrior/cor1.npy")
        cor4 = np.load("./p4/warrior/cor2.npy")

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

        # replace black pixels with white pixels
        _black_idxs = (rectified_im1[:, :, 0] == 0) & (rectified_im1[:, :, 1] == 0) & (rectified_im1[:,
        rectified_im1[:, :][_black_idxs] = [1.0, 1.0, 1.0]
        _black_idxs = (rectified_im2[:, :, 0] == 0) & (rectified_im2[:, :, 1] == 0) & (rectified_im2[:,
        rectified_im2[:, :][_black_idxs] = [1.0, 1.0, 1.0]

        nCorners = 10

        # Choose your threshold and NCC matching window radius
        NCCth = 0.6
        R = 75

        # detect corners using corner detector here, store in corners1
        corners1 = corner_detect(rgb2gray(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)

# For warrior
rectified_im3, rectified_im4, new_cor3, new_cor4 = image_rectification(I3, I4, cor3, cor4)
F_new2 = fundamental_matrix(new_cor3, new_cor4)

# replace black pixels with white pixels
_black_idxs = (rectified_im3[:, :, 0] == 0) & (rectified_im3[:, :, 1] == 0) & (rectified_im3[:,
rectified_im3[:, :][_black_idxs] = [1.0, 1.0, 1.0]
_black_idxs = (rectified_im4[:, :, 0] == 0) & (rectified_im4[:, :, 1] == 0) & (rectified_im4[:,
rectified_im4[:, :][_black_idxs] = [1.0, 1.0, 1.0]

# You may wish to change your NCCth and R for warrior here.
NCCth = 0.6
R = 150

corners2 = corner_detect(rgb2gray(rectified_im3), nCorners, smoothSTD, windowSize)
corrs = correspondence_matching_epipole(rectified_im3, rectified_im4, corners2, F_new2, R, NCCt
display_correspondence(rectified_im3, rectified_im4, corrs)
```

1.5.8 Submission Instructions

Remember to submit a PDF version of this notebook to Gradescope. You can create a PDF via **File > Download as > PDF via LaTeX** (preferred, if possible), or by downloading as an HTML page and then "printing" the HTML page to a PDF (by opening the print dialog and then choosing the "Save as PDF" option).

```
In [ ]:
```