1 CSE 252A Computer Vision I Fall 2017

1.1 Assignment 3

This assignment contains theoretical and programming exercises. If you plan to submit hand written answers for theoretical exercises, please be sure your writing is readable and merge those in order with the final pdf you create out of this notebook.

1.2 Problem 1: Epipolar Geometry [3 pts]

Consider two cameras whose image planes are the z=1 plane, and whose focal points are at (-25, 0, 0) and (25, 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 (-25, 0, 1) in world coordinates, while if (u, v) = (0, 0) this is the point (25, 0, 1).

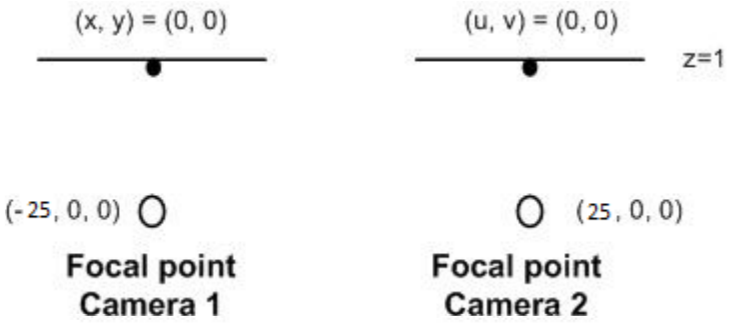


Figure 1

a) Suppose the points (x, y) = (8, 8) is matched with disparity of 7 to the point (u, v) = (1, 8). What is the 3D location of this point?

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

1.3 Problem 2: Epipolar Rectification [4 pts]

In stereo vision, image rectification is a common preprocessing step to simplify the problem of finding matching points between images. The goal is to warp image views such that the epipolar lines are horizontal scan lines of the input images. Suppose that we have captured two images I_A and I_B from identical calibrated cameras separated by a rigid transformation

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

Without loss of generality assume that camera A's optical center is positioned at the origin and that its optical axis is in the direction of the z-axis.

From the lecture, a rectifying transform for each image should map the epipole to the point infinitely far away in the horizontal direction $H_A e_A = H_B e_B = [1, 0, 0]^T$. Consider the following special cases:

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

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

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

d) Pure rotation $t = [0, 0, 0]^T$, R is an arbitrary rotation matrix

For each of these cases, determine whether or not epipolar rectification is possible. Include the following information for each case

- The epipoles e_A and e_B
- The equation of the epipolar line l_B in I_B corresponding to the point $[x_A, y_A, 1]^T$ in I_A (if one exists)
- A plausible solution to the rectifying transforms H_A and H_B (if one exists) that attempts to minimize distortion (is as close as possible to a 2D rigid transformation). Note that the above 4 cases are special cases; a simple solution should become apparent by looking at the epipolar lines.

One or more of the above rigid transformations may be a degenerate case where rectification is not possible or epipolar geometry does not apply. If so, explain why.

1.4 Problem 3: NCC [3 pts]

Show that maximizing the NCC $\sum_{i,j} \tilde{W}_1(i,j) \cdot \tilde{W}_2(i,j)$ is equivalent to minimizing the NSSD $\sum_{i,j} \left|\tilde{W}_1(i,j) - \tilde{W}_2(i,j)\right|^2$.

Hint: Express c_{NCC} and c_{NSSD} in terms of the vectors $\tilde{w_1} = \tilde{W}_1(:)$ and $\tilde{w_2} = \tilde{W}_2(:)$. Thus, $c_{NCC} = \tilde{w_1}^T \tilde{w_2}$ and $c_{NSSD} = (\tilde{w_1} - \tilde{w_2})^T (\tilde{w_1} - \tilde{w_2})$,

$\tilde{W} = \frac{W - \overline{W}}{\sqrt{\sum_{k,l} (W(k,l) - \overline{W})^2}}$ is a mean-shifted and normalized version of the window. N refers to the number of pixels in each window.

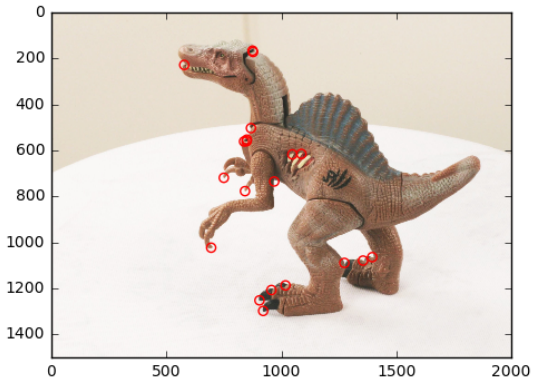
1.5 Problem 4: Sparse Stereo Matching [18 pts]

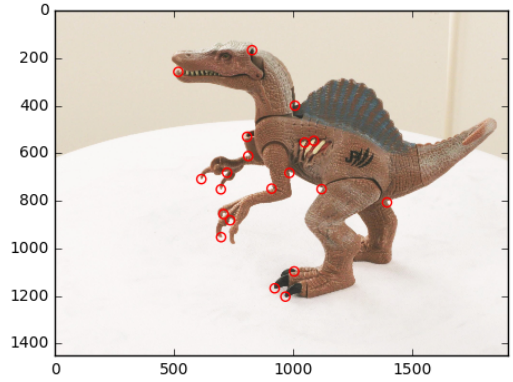
In this problem we will play around with sparse stereo matching methods. You will work on two image pairs, a warrior figure and a figure from the Matrix movies. These files both contain two images, two camera matrices, and set sets of corresponding points (extracted by manually clicking the images). For illustration, I have run my code on a third image pair (dino1.png, dino2.png). This data is also provided for you to debug your code, but you should only report results on warrior and matrix. In other words, where I include one (or a pair) of images in the assignment below, you will provide the same thing but for BOTH matrix and warrior. Note that the matrix image pair is harder, in the sense that the matching algorithms we are implementing will not work quite as well. You should expect good results, however, on warrior.

1.5.1 Corner Detection [3 pts]

The first thing we need to do is to build a corner detector. This should be done according to http://cseweb.ucsd.edu/classes/fa17/cse252A-a/lec11.pdf (http://cseweb.ucsd.edu/classes/fa17/cse252A-a/lec11.pdf). Your should fill in the function `corner_detect` below, and take as `input corner_detect(image, nCorners, smoothSTD, windowSize)` where smoothSTD is the standard deviation of the smoothing kernel and windowSize is the window size for corner detector and non maximum suppression. In the lecture the corner detector was implemented using a hard threshold. Do not do that but instead return the nCorners strongest corners after non-maximum suppression. This way you can control exactly how many corners are returned. Run your code on all four images (with nCorners = 20) and show outputs as in Figure 2. You may find scipy.ndimage.filters.gaussian_filter easy to use for smoothing. In this problem try different parameters and then comment on results.

1. windowSize = 3, 5, 9, 17
2. smoothSTD = 0.5, 1, 2, 4





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

In [ ]:
```
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 c
        nCorners: Total number of corners to be extracted.
        smoothSTD: Standard deviation of the Gaussian smoot
        windowSize: Window size for corner detector and non

    Returns:
        Detected corners (in image coordinate) in a numpy a

    """

    """
    Your code here:
    """
    corners = np.zeros((nCorners, 2))
    return corners
```

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

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

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

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, e

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

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

1.5.2 SSD Matching [1 pts]

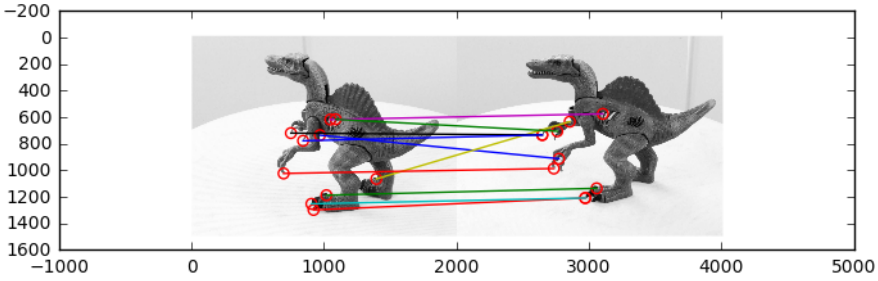
Write a function ssd_match that implements the SSD matching algorithm for two input windows.

In [ ]:
```
def ssd_match(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 i
        c2: Center (in image coordinate) of the window in i
        R: R is the radius of the patch, 2 * R + 1 is the w

    Returns:
        SSD matching score for two input windows.

    """

    """
    Your code here:
    """
    matching_score = 0
    return matching_score
```

In [ ]:
```
# test SSD 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 ssd_match(img1, img2, np.array([1, 1]), np.array([1,
# should print 20
print ssd_match(img1, img2, np.array([2, 1]), np.array([2,
# should print 20
print ssd_match(img1, img2, np.array([1, 1]), np.array([2,
# should print 46
```

1.5.3 Naive Matching [3 pts]

Equipped with the corner detector and the SSD matching function, we are ready to start finding correspondances. One naive strategy is to try and find the best match between the two sets of corner points. Write a script that does this, namely, for each corner in image1, find the best match from the detected corners in image2 (or, if the SSD match score is too low, then return no match for that point). You will have to figure out a good threshold (SSDth) value by experimentation. Write a function naiveCorrespondanceMatching.m and call it as below. Examine your results for 10, 20, and 30 detected corners in each image. Choose a number of detected corners to the maximize the number of correct matching pairs. naive_matching will call your SSD mathching code.



In [ ]:

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

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

    Returns:
        SSD matching result a list of tuple (c1, c2),
        c1 is the 1x2 corner location in image 1,
        c2 is the 1x2 corner location in image 2.

    """

    """
    Your code here:
    """
    matching = []
    return matching
```

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

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

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

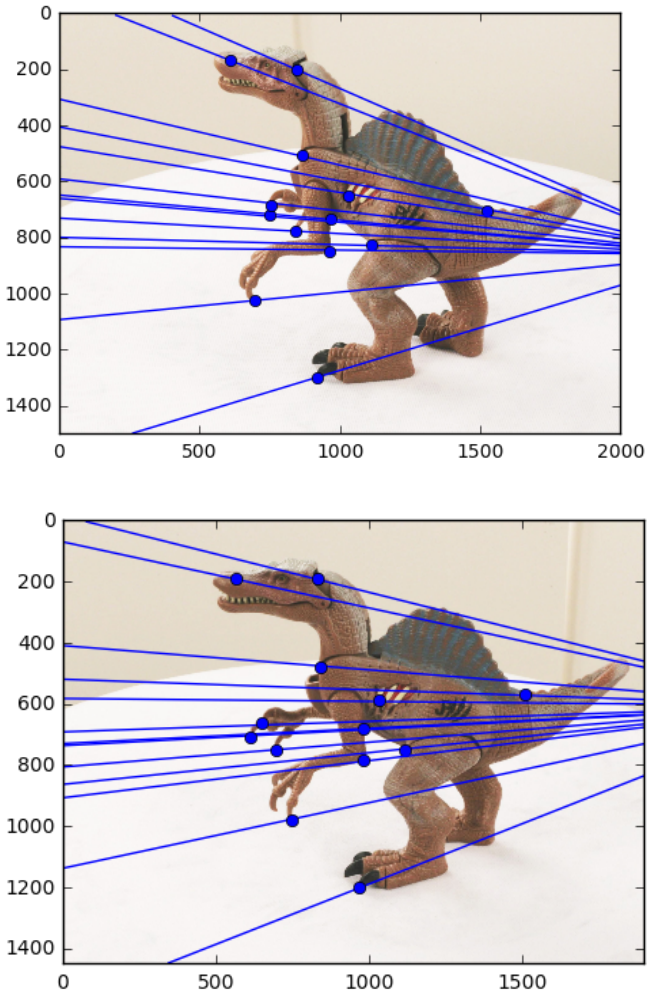
In [ ]:
```
# match corners
R = 15
SSDth = 100
matching_mat = naive_matching(imgs_mat[0]/255, imgs_mat[1]/
matching_war = naive_matching(imgs_war[0]/255, imgs_war[1]/
```

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
    for p1, p2 in matching:
        plt.scatter(p1[0], p1[1], s=35, edgecolors='r', fac
        plt.scatter(p2[0] + img1.shape[1], p2[1], s=35, edg
        plt.plot([p1[0], p2[0] + img1.shape[1]], [p1[1], p2
    plt.savefig('dino_matching.png')
    plt.show()

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

1.5.4 Epipolar Geometry [3 pts]

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

Using this fundamental matrix, plot the epipolar lines in both image pairs across all images. For this part you may want to complete the function plot_epipolar_lines. Shown your result for matrix and warrior as the figure below.



In [ ]:

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

def compute_fundamental(x1,x2):
    """    Computes the fundamental matrix from correspondi
        (x1,x2 3*n arrays) using the 8 point algorithm.
        Each row in the A matrix below is constructed as
        [x'*x, x'*y, x', y'*x, y'*y, y', x, y, 1]
    """

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

    # build matrix for equations
    A = np.zeros((n,9))
    for i in range(n):
        A[i] = [x1[0,i]*x2[0,i], x1[0,i]*x2[1,i], x1[0,i]*x
                x1[1,i]*x2[0,i], x1[1,i]*x2[1,i], x1[1,i]*x
                x1[2,i]*x2[0,i], x1[2,i]*x2[1,i], x1[2,i]*x

    # compute linear least square solution
    U,S,V = np.linalg.svd(A)
    F = V[-1].reshape(3,3)

    # constrain F
    # make rank 2 by zeroing out last singular value
    U,S,V = np.linalg.svd(F)
    S[2] = 0
    F = np.dot(U,np.dot(np.diag(S),V))

    return F/F[2,2]


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

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

In [ ]:
```
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 im
        cor2: Corners in homogeneous image coordinate in im

    """

    """
    Your code here:
    """
```

In [ ]:
```
# replace images and corners with those of matrix and warri
I1 = imread("./p4/dino/dino0.png")
I2 = imread("./p4/dino/dino1.png")

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

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

1.5.5 Matching Using Epipolar Geometry [4 pts]

We will now use the epipolar geometry to build a better matching algorithm. First, detect 10 corners in Image1. Then, for each corner, do a linesearch along the corresponding epipolar line in Image2. Evaluate the SSD score for each point along this line and return the best match (or no match if all scores are below the SSDth). R is the radius (size) of the SSD patch in the code below. You do not have to run this in both directions. Show your result as in the naive matching part.

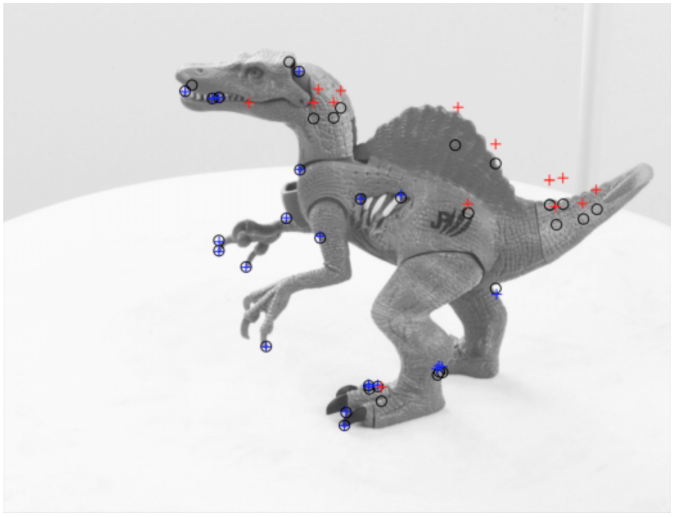
In [ ]:
```
def display_correspondence(img1, img2, corrs):
    """Plot matching result on image pair given images and 

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

    """

    """
    Your code here.
    You may refer to the show_matching_result function
    """

def correspondence_matching_epipole(img1, img2, corners1, F
    """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
        R: SSD matching window radius.
        SSDth: SSD matching threshold.


    Returns:
        Matching result to be used in display_correspondenc

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

```

In [ ]:
```
# replace images and corners with those of matrix and warri
I1 = rgb2gray(imread("./p4/dino/dino0.png"))
I2 = rgb2gray(imread("./p4/dino/dino1.png"))

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

F = fundamental_matrix(cor1, cor2)

nCorners = 10
# detect corners using corner detector here, store in corne
corners1 = corner_detect(I1, nCorners, smoothSTD, windowSiz
corrs = correspondence_matching_epipole(I1, I2, corners1, F
display_correspondence(img1, img2, corrs)
```

1.5.6 Triangulation [4 pts]

Now that you have found correspondences between the pairs of images we can triangulate the corresponding 3D points. Since we do not enforce the ordering constraint the correspondences you have found are likely to be noisy and to contain a fair amount of outliers. Using the provided camera matrices you will triangulate a 3D point for each corresponding pair of points. Then by reprojecting the 3D points you will be able to find most of the outliers. You should implement the linear triangulation method described in lecture (For additional reference, see section 12.2 of ”Multiple View Geometry” by Hartley and Zisserman. The book is available electronically from any UCSD IP address.). P1 and P2 below are the camera matrices. Also write a function, find_outliers, that reprojects the world points to Image2, and then determines which points are outliers and inliers respectively. For this purpose, we will call a point an outlier if the distance between the true location, and the reprojected point location is more than 20 pixels.

Display your results by showing, for image 2, the original points, the inliers, and the outliers in different markers. Compare this outlierplot with your epipolar line matching result. Does the detected outliers correspond to false matches?



.

In [ ]:
```
def triangulate(corsSSD, P1, P2):
    """Find corner correspondence along epipolar line.

    Args:
        corsSSD: Corner correspondence
        P1: Projection matrix 1
        P2: Projection matrix 2

    Returns:
        3D points from triangulation

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

def find_outliers(points3D, P2, outlierTH, corsSSD):
    """Find and plot outliers on image 2

    Args:
        points3D: 3D point from triangulation
        P2: Projection matrix in for image 2
        outlierTH: outlier pixel distance threshold
        corsSSD: Corner correspondence

    """
    """
    Your code here
    """
```

In [ ]:
```
# replace images and corners with those of matrix and warri
I1 = rgb2gray(imread("./p4/dino/dino0.png"))
I2 = rgb2gray(imread("./p4/dino/dino1.png"))

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

P1 = np.load("./p4/dino/proj1.npy")
P2 = np.load("./p4/dino/proj2.npy")

outlierTH = 20
F = fundamental_matrix(cor1, cor2)
ncorners = 50
corners1 = corner_detect(I1, ncorners, smoothSTD, windowSiz
corsSSD = correspondence_matching_epipole(I1, I2, corners1,
points3D = triangulate(corsSSD, P1, P2)
inlier, outlier = find_outliers(points3D, P2, outlierTH, co
```