Wei-Yuan Wen, Yen-Ting Chen, Shun-Jen Lee
A53202335, A53214417, A53209671
University of California, San Diego

November 2, 2016

HOMEWORK 2

Problem 1. Epipolar Geometry Theory [6 pt]

Description:

Suppose a camera calibration gives a transformation (R, T) such that a point in the world maps to the camera by ${}^{C}P = R^{W}P + T$.

1. Given calibrations of two cameras (a stereo pair) to a common external coordinate system, represented by R_1, T_1, R_2, T_2, provide an expression that will map points expressed in the coordinate system of camera 1 to that of camera 2.
2. What is the length of the baseline of the stereo pair.
3. Give an expression for the Essential Matrix in terms of R_1, T_1, R_2, T_2

Solution: 1. Assume a point in world coordinate ${}^{W}P, and the mapping equation form world to the camera as are : And now we map the point from the camera 1 coordinate system to the world one and then the camera 2 one : 2. Let {}^{C_1}P = O_1$, we have:

$$ {}^{C_2}P = -R_2R_1^TT_1 + T_2l = |-R_2R_1^TT_1 + T_2| $$

3. Assume the origins of two camera O_1, O_2 and a point P projected to two image plane p_1, p_2, we have:

$$ p_2(O_2O_1xp_1) = 0p_2((-R_2R_1^TT_1 + T_2)x(R_2R_1^Tp_1)) = 0\varepsilon = [-R_2R_1^TT_1 + T_2]_xR_2R_1^T $$

Problem 2. Epipolar Geometry [4 pt] 1.We can derive two ray parametric equations from the description in the problem:

$$x = -12 + 8t, y = 7t, z = t, t \geq 1 x = 12 + 2s, y = 7s, z = s, s \geq 1$$

solving the equations above, we have the intersection of two rays:

$$t = s = 4 (x, y, z) = (20, 28, 4)$$

2.We express the line in the form:

$$X = t, Y = 0, Z = 2 - t$$

and the projected line on the two planes are:

$$x_1 = \frac{t + 12}{2 - t}, y_1 = 0 x_2 = \frac{t - 12}{2 - t}, y_2 = 0$$

u denotes the distance between projected point and (0,0) in image 1, and u+d denoteds the one in image 2. We can derive:

$$\frac{t + 12}{2 - t} = u \frac{t - 12}{2 - t} = u + d$$

We can derive that:

$$t = \frac{2u - 12}{1 + u} d = \frac{-12}{7} (1 + u)$$

Description:

Consider two cameras whose image planes are the $z = 1$ plane, and whose focal points are at (12, 0, 0) and (12, 0, 0). Well 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).

1. Suppose the points $(x, y) = (8, 7)$ is matched with disparity of 6 to the point $(u, v) = (2, 7)$. What is the 3D location of this point?

2. Consider points that lie on the 3-D line $X + Z = 2, Y = 0$. 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 $Z > 1$.

Solution:

Problem 3. Reconstruction Accuracy [3pt]

Description:

Characterize the accuracy of the 2D stereo system below. Assume the only source of noise is the localization of corresponding points in the two images (in other words, the only source of error is the disparity). Discuss the dependence of the error in depth estimation ($\Delta Z'$) as a function of baseline width (b), focal length (f), and depth (Z').

Solution:

As mentioned in lecture 8, we assume that the localization positions of corresponding points in the two images are X_L and X_R. By using similar triangles, we have:

$$X' = \frac{bX_L}{(X_L - X_R)}, Z' = \frac{bf}{(X_L - X_R)}, \text{ where } (X_L - X_R) \text{ is the disparity.}$$

To observe the error in depth estimation with respect to the noise of disparity, we differentiate Z' to the disparity. We get:

$$\frac{dZ'}{d(X_L - X_R)} = \frac{-bf}{(X_L - X_R)^2} \rightarrow \Delta Z' = \frac{-bf}{(X_L - X_R)^2}\Delta(X_L - X_R)$$

By substituting Z' into the equation, the result becomes:

$$\Delta Z' = \frac{-Z'^2}{bf}\Delta(X_L - X_R)$$

Hence, we can observe that ($\Delta Z'$) is in proportional to the square of estimated depth (Z'^2) and inversely proportional to the product of baseline width (b) and focal length (f).

Problem 4. Filters as Templates [16pt]

Description:

In this problem we will play with convolution filters. Filters, when convolved with an image, will fire strongest on locations of an image that look like the filter. This allows us to use filters as object templates in order to identify specific objects within an image. In the case of this assignment, we will be finding cars within an image by convolving a car template onto that image. Although this is not a very good way to do object detection, this problem will show you some of the steps necessary to create a good object detector. The goal of this problem will be to teach some pre-processing steps to make vision algorithms be successful and some strengths and weaknesses of filters. Each problem will ask you to analyze and explain your results. If you do not provide an explanation of why or why not something happened, then you will not get full credit. Provide your code in the appendix.

4.1 Warmup [3pt]

First you will convolve a filter to a synthetic image. The filter or template is `filter.jpg` and the synthetic image is `toy.png`. These files are available on the course webpage. You may want to modify the filter image and original slightly. I suggest `filter_img = filter_img - mean(filter_img(:))`. To convolve the filter image with the toy example, in Matlab you will want to use `conv2`. The output of the convolution will create an intensity image. Provide this image in the report. In the original image (not the image with its mean subtracted), draw a bounding box of the same size as the filter image around the top 3 intensity value locations in the convolved image. Describe how well you think this technique will work on more realistic images? Do you foresee any problems for this algorithm on more realistic images?

4.2 Detection Error [3pt]

We have now created an algorithm that produces a bounding box around a detected object. However we have no way to know if the bounding box is good or bad. In the example images shown above, the bounding boxes look reasonable, but not perfect. Given a ground truth bounding box (g) and a predicted bounding box (p), a commonly used measurement for bounding box quality is $\frac{p \cap q}{p \cup q}$. More intuitively, this is the number of overlapping pixels between the bounding boxes divided by the total number of unique pixels of the two bounding boxes combined. Assuming that all bounding boxes will be axis-aligned rectangles, implement this error function and try it on the toy example in the previous section. Choose 3 different ground truth bounding box sizes around one of the Mickey silhouettes. In general, if the overlap is 50% or more, you may consider that the detection did a good job.

4.3 More Realistic Images [8pt]

Now that you have created an algorithm for matching templates and a function to determine the quality of the match, it is time to try some more realistic images. The file, `cartemplate.jpg`, will be the filter to convolve on each of the 5 other car images (`car1.jpg, car2.jpg, car3.jpg, car4.jpg, car5.jpg`). Each image will have an associated text files that contains 2 x, y coordinates (one pair per line). These coordinates will be the ground truth bounding box for each image. For each car image, provide the following:

- A heat map image.
- A bounding box drawn on the original image.

- The bounding box overlap percent.
- A description of what pre-processing steps you needed to do to achieve this overlap.
- An explanation of why you felt these steps made sense.

4.4 Invariance [2pt]
In computer vision there is often a desire for features or algorithms to be invariant to X. One example briefly described in class was illumination invariance. The detection algorithm that was implemented for this problem may have seemed a bit brittle. Can you describe a few things that this algorithm was not invariant to? For example, this algorithm was not scale-invariant, meaning the size of the filter with respect to the size of the object being detected mattered. One filter size should not have worked on everything.

Solution:

4.1 Warmup:
First, I pre-processed the filter by using `filter_img = filter_img - mean(filter_img(:))`. Then I used `conv2` to convolve the filter to the toy image and got an intensity image. The intensity image is shown as Fig. 1(a).
To draw bounding boxes around the top 3 intensity value, I first found the top 3 intensity value and got the responding coordinates. Then I used `rectangle` to draw bounding boxes of the same size as the filter image around these coordinates in the original image. The result is shown as Fig. 1(b).
I think this technique won't work well on more realistic images since the background of the input image is pretty simple. Also, the filter is pretty as the same size of the object we want to find, so we won't need to do much pre-processing steps on the filter image. If we apply this algorithm on more realistic images, the maximum of the intensity image may not be locate at the object we want to find, since the background will be complicated and distracting. This will mislead us to draw the bounding box on different location in the image. On the other hand, we may not use only a simple filter image to find various kind of object in the same category, or we need to do some more pre-processing steps to get an accurate bounding box.

The codes are attached as Listing 1 in Appendix.

4.2 Detection Error:
I wrote a function named `detectError`, which takes ground truth box and predicted bounding box as two arguments. Since I have the length and width and at least 2 coordinates of these rectangles, I can easily compute the number of overlapping pixels of the two bounding boxes between the bounding boxes and then divide it by the total number of unique pixels of the two bounding boxes. I then constructed 3 ground truth bounding boxes around one of the Mickey silhouettes and found the overlap is more than 50%.

The codes are attached as Listing 3 in Appendix.

4.3 More Realistic Images:
For `car1` image:

- Bounding box overlap percent: 94.72%.
- Pre-processing steps:
  a) Subtracted both filter image and `car1` image by their means.
  b) Crop the filter.
  c) Rescale the filter image to the size of ground truth bounding box.
  d) Flip the filter image in x and y dimensions before convolution.
- Explanation:
  a) I want the object in `car1` that looks like the filter image has maximums of intensity values, so I subtracted both filter image and `car1` image by their means.
  b) I first cropped the filter image to get rid of the blank area, because I don't want the blank to affect my result of convolution.
  c) Then I rescaled the filter image to the size of ground truth bounding box, this way the intensity image will be more accurate and make the maximum locate at the position of `car1`.
  d) To do the convolution, we need to make sure the filter image is upside down and reversed from left to right, so I flipped it in x and y dimensions right before convolution.
- The result is shown as Fig. 2(a) and Fig. 2(b).

For `car2` image:

- Bounding box overlap percent: 89.59%.
- Pre-processing steps:
  a) Subtracted both filter image and `car2` image by their means.
  b) Crop the filter.
  c) Rescale the filter image to the size of ground truth bounding box.
  d) Reverse the filter image from left to right.
  e) Flip the filter image in x and y dimensions before convolution.
- Explanation:
  a) I want the object in `car2` that looks like the filter image has maximums of intensity values, so I subtracted both filter image and `car2` image by their means.
  b) I cropped the filter to get rid of the blank area, because I don't want the blank to affect my result of convolution.
  c) Then I rescaled the filter image to the size of ground truth bounding box, this way the intensity image will be more accurate and make the maximum locate at the position of `car2`.
  d) I reversed the filter image from left to right, since the head of the car in the `car2` image points toward right.
  e) To do the convolution, we need to make sure the filter image is upside down and reversed from left to right, so I flipped it in x and y dimensions right before convolution.
- The result is shown as Fig. 3(a) and Fig. 3(b).

For `car3` image:

- Bounding box overlap percent: 50.80%.

- Pre-processing steps:
  a) Subtracted both filter image and `car3` image by their means.
  b) Crop the filter.
  c) Rescale the filter image to the size of the rear part of the car in `car3` image.
  d) Flip the filter image in x and y dimensions before convolution.
- Explanation:
  a) I want the object in `car3` that looks like the filter image has maximums of intensity values, so I subtracted both filter image and `car3` image by their means.
  b) I cropped the filter to get rid of the blank area, because I don't want the blank to affect my result of convolution.
  c) Then I rescaled the filter image to the size of the rear part of the car in `car3` image, because I thought somehow that rear part looks more alike with the filter image than the side part. This way the intensity image will be more accurate and make the maximum locate at the position of `car3`.
  d) To do the convolution, we need to make sure the filter image is upside down and reversed from left to right, so I flipped it in x and y dimensions right before convolution.
- The result is shown as Fig. 4(a) and Fig. 4(b).

For `car4` image:

- Bounding box overlap percent: 65.35%.
- Pre-processing steps:
  a) Subtracted both filter image and `car2` image by their means.
  b) Crop the filter.
  c) Add 0.1 on each pixel in the filter image.
  d) Rescale the filter image to the size of ground truth bounding box.
  e) Reverse the filter image from left to right.
  f) Flip the filter image in x and y dimensions before convolution.
- Explanation:
  a) I want the object in `car4` that looks like the filter image has maximums of intensity values, so I subtracted both filter image and `car4` image by their means.
  b) I cropped the filter to get rid of the blank area, because I don't want the blank to affect my result of convolution.
  c) I found the top-left area in `car4` image has similar brightness with the filter image, which means the intensity is likely to locate at that area. Thus I add 0.1 to every pixels in the filter image to make it brighter.
  d) Then I rescaled the filter image to the size of ground truth bounding box, this way the intensity image will be more accurate and make the maximum locate at the position of `car4`.
  e) I reversed the filter image from left to right, since the head of the car in the `car2` image points toward right.
  f) To do the convolution, we need to make sure the filter image is upside down and reversed from left to right, so I flipped it in x and y dimensions right before convolution.

- The result is shown as Fig. 5(a) and Fig. 5(b).

For `car5` image:

- Bounding box overlap percent: 74.40%.
- Pre-processing steps:
  a) Subtracted both filter image and `car5` image by their means.
  b) Crop the filter.
  c) Add 0.1 on each pixel in the filter image.
  d) Rescale the filter image to the size of ground truth bounding box.
  e) Reverse the filter image from left to right.
  f) Flip the filter image in x and y dimensions before convolution.
- Explanation:
  a) I want the object in `car5` that looks like the filter image has maximums of intensity values, so I subtracted both filter image and `car5` image by their means.
  b) I cropped the filter to get rid of the blank area, because I don't want the blank to affect my result of convolution.
  c) I found that rhino in `car5` image and the right part of the forest has similar brightness with the filter image, which means the intensity is likely to locate at that area. Thus I add 0.1 to every pixels in the filter image to make it brighter.
  d) Then I rescaled the filter image to the size of ground truth bounding box, this way the intensity image will be more accurate and make the maximum locate at the position of `car5`.
  e) I reversed the filter image from left to right, because I thought the trunk of truck in the `car5` image somehow seems like the head of the filter image.
  f) To do the convolution, we need to make sure the filter image is upside down and reversed from left to right, so I flipped it in x and y dimensions right before convolution.
- The result is shown as Fig. 6(a) and Fig. 6(b).

The codes are attached as Listing 1 to Listing 5 in Appendix.

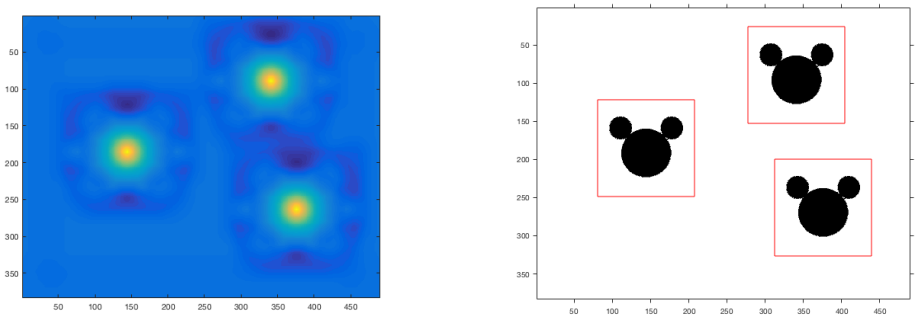
4.4 Invariance:

a) Not color-invariant, the color/brightness of the filter with respect to the color/brightness of the object being detected mattered. We may need to adjust the brightness of the filter to find the correct object in the image. For example, in the `car4` and `car5` images, the darker area like smoke, forest and rhinos will mislead us to detect wrong objects.

b) Not shape-invariant, the silhouette of the filter with respect to the silhouette of the object being detected mattered. For example, in the `car3` image, we can see that the silhouette of a car may not appear at the correct location we expect.

Result:

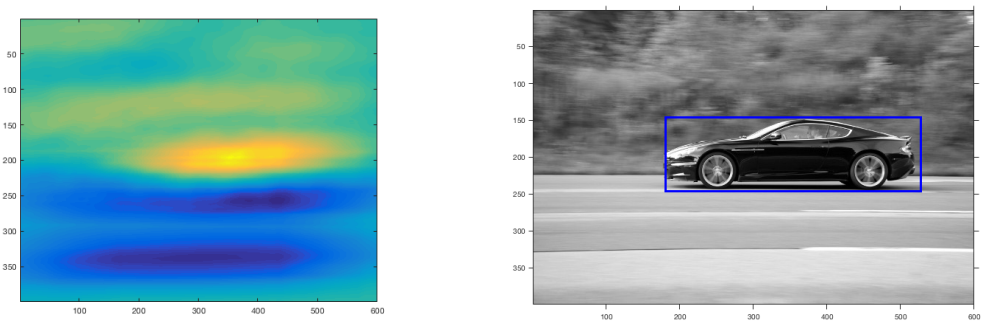
4.1 Warmup:



(a) (b)

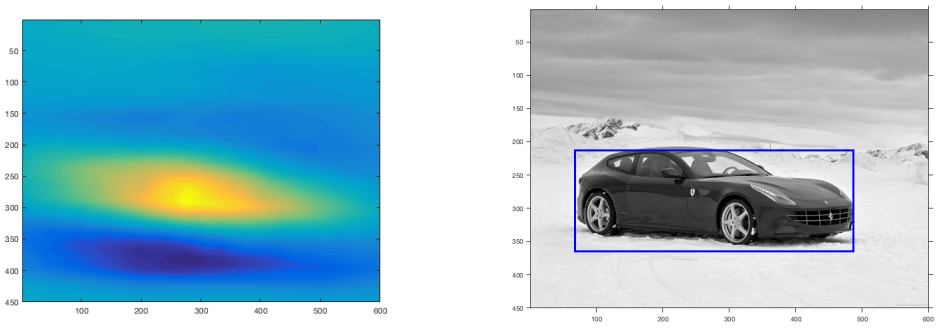
Figure 1: (a) Intensity image, (b) Bounding boxes.

4.3 More Realistic Images:



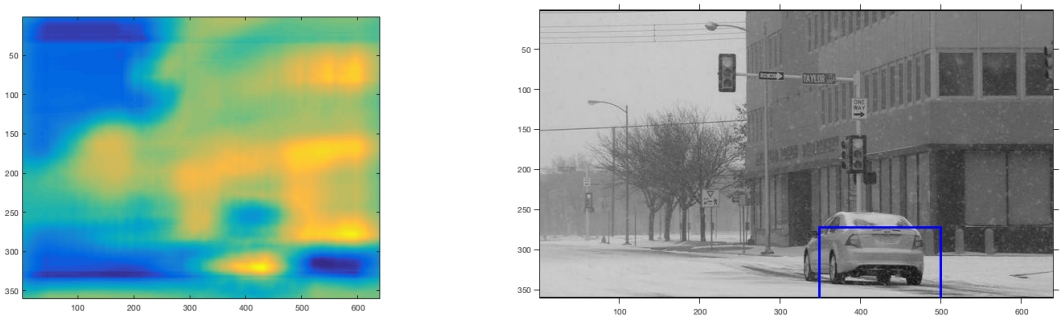
(a) (b)

Figure 2: Outputs for `car1`. (a) Heat map, (b) Bounding boxes.



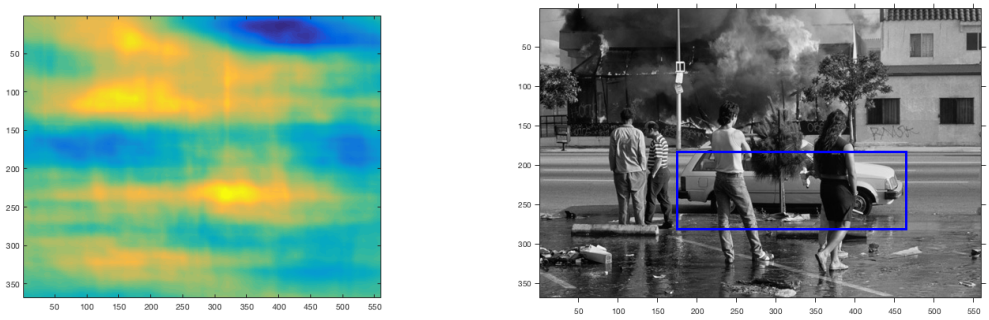
(a) (b)

Figure 3: Outputs for `car2`. (a) Heat map, (b) Bounding boxes.



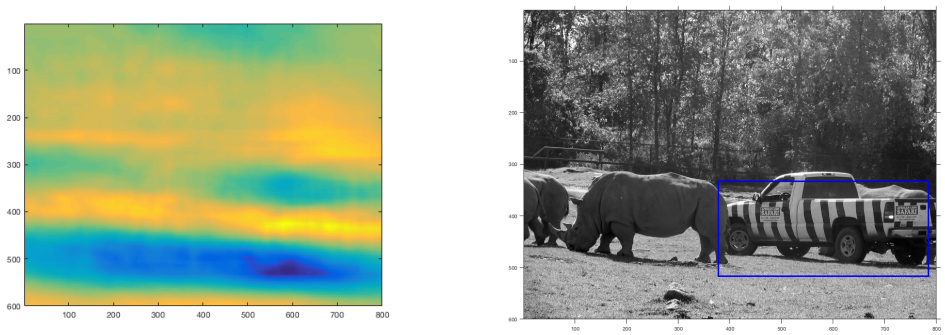
(a) (b)

Figure 4: Outputs for `car3`. (a) Heat map, (b) Bounding boxes.



(a) (b)

Figure 5: Outputs for `car4`. (a) Heat map, (b) Bounding boxes.



(a) (b)

Figure 6: Outputs for `car5`. (a) Heat map, (b) Bounding boxes.

Problem 5. Canny Edge Detection [9pt]

Description:

In this problem, you have to write a function to do CANNY EDGE DETECTION. The following steps need to be implemented.

- **Smoothing [1pt]:** First, we need to smooth the images to remove noise from being considered as edges. For this assignment, use a 5×5 Gaussian kernel filter 1 with $= 1.4$ to smooth the images.

$$K = \frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 12 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} \tag{1}$$

- **Gradient Computation [2pt]:** After you have finished smoothing, find the image gradient in the horizontal and vertical directions. You can use Sobel operators 2 as your filter kernel to calculate G_x and G_y. G_x and G_y are the gradients along the x and y axis respectively. s_x and s_y are the corresponding kernels. Compute the gradient magnitude image as $|G| = \sqrt{G_x^2 + G_y^2}$. The edge direction as each pixel is given as $G_\theta = \tan^{-1}(\frac{G_x}{G_y})$.

$$s_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, s_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}. \tag{2}$$

- **Non Maximum Suppression [3pt]:** Our desired edges need to be sharp, not thick like the ones in gradient image. Use non maximum suppression to preserve all local maximas and discard the rest. You can use the following method to do so:
  For each pixel do:
  • Round the gradient direction θ to the nearest multiple of 45° in a 8-connected neighbourhood.
  • Compare the edge strength at the current pixel to the pixels along the $+ve$ and ve gradient direction in the 8-connected neighbourhood.
  • Preserve the values of only those pixels which have maximum gradient magnitudes in the neighbourhood along the $+ve$ and ve gradient direction.

- **Hysteresis Thresholding [3pt]:** Choose optimum values of thresholds and use the thresholding approach given in lecture 7. This will remove the edges caused due to noise and colour variations.

Compute the images after each step and select suitable thresholds that retains most of the true edges. For this question use the image `geisel.jpeg` in the data folder.

Solution:

For smoothing, I used `conv2.m` and applied the Gaussian filter with $\sigma = 1.4$ to smooth the image. The result is shown as Fig. 8(a).

For gradient computation, I use s_x and s_y to calculate G_x and G_y respectively. Then I computed the gradient magnitude image using $|G| = \sqrt{G_x^2 + G_y^2}$ and get the edge direction at each pixel by computing $G_\theta = \tan^{-1}(\frac{G_x}{G_y})$. The result is shown as Fig. 8(b).

For non maximum suppression, I first rounded every gradient direction at each pixel to the nearest multiple of 45°. And for each pixel, I checked if the gradient magnitude of two pixels along the $+ve$ and ve gradient direction in the 8-connected neighbourhood is smaller than that of current pixel. If both magnitudes are smaller, I added the current pixel to the edge candidate. The result is shown as Fig. 8(c).

For hysteresis thresholding, I first added all the pixels, which have values higher than $\tau_h igh$, from edge candidate into an dynamic array. For each pixel in the array, I added it into a result map and added pixels within 8-connected neighbourhood of it into the array if those pixels' gradient magnitude are higher than $\tau_l ow$ and are not already in the result map. The result is shown as Fig. 8(d).

The codes are attached as Listing 6 to Listing 7 in Appendix.

Result:

Figure 7: Original Image

(a) (b)(c) (d)

Figure 8: (a) Smoothed Image using Gaussian filter with $\sigma = 1.4$, (b) Gradient Magnitude, (c) Non Maximum Suppression edges, (d) Final Edge after Hysteresis Thresholding.

Problem 6. Sparse Stereo Matching [27pt]

Description:

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 (`warrior2.mat` and `matrix2.mat`). 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 (`dino2.mat`). This data is also provided on the webpage 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. To make the TAs extra happy, make the line width and marker sizes bigger than the default sizes.

6.1 Corner Detection [8pt]

The first thing we need to do is to build a corner detector. This should be done according to `http://cseweb.ucsd.edu/classes/fa16/cse252A-a/lec7.pdf`. Your file should be named `CornerDetect.m`, and take as input

```
corners = CornerDetect(Image, nCorners, smoothSTD, windowSize)
```

where `smoothSTD` is the standard deviation of the smoothing kernel, and `windowSize` the size of the smoothing window. 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.

Solution:

To detect the corners, we follow the method in lecture 7.

1. Filter image with a Gaussian.
2. Compute the gradient in both x and y directions for every pixel.
3. Move window over image, and for each window location:

(1) Construct the matrix C over the window.

$$C(x,y) = \begin{pmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{pmatrix}$$

(2) Find the eigenvalue for C using eig() in matlab.

(3) Store the smaller eigenvalue for every pixel in a matrix.

4. Perform non-maximum suppression by comparing the stored eigenvalue in every pixel with its neighbors(a window). If the eigenvalue is the maximum in the window, we take this pixel as a candidate of a corner.
5. We sort all the candidates by their eigenvalues, and choose the largest nCorners pixels as our result.

Result:

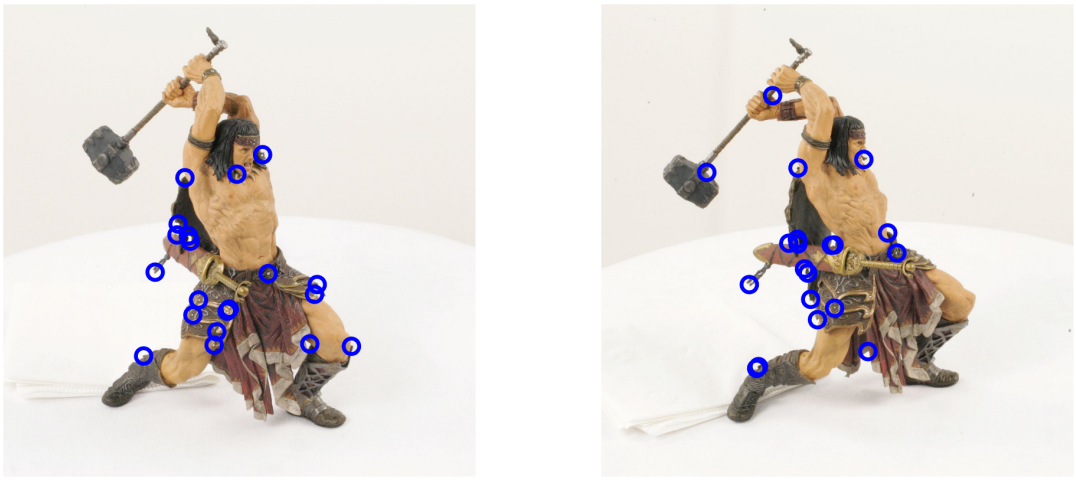


Figure 9: Corner Detection: Warrior

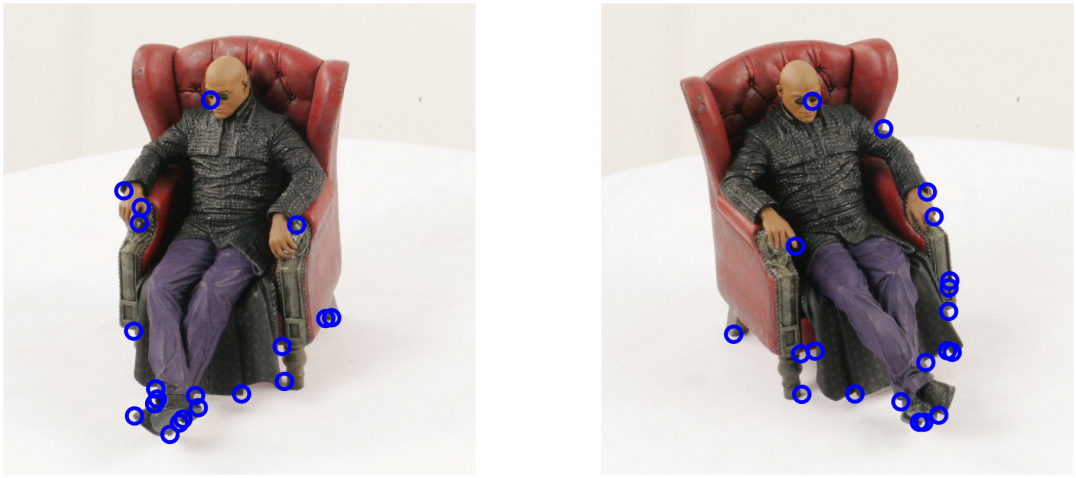


Figure 10: Corner Detection: Matrix

6.2 SSD Matching [2pt]
Write a function `SSDmatch.m` that implements the SSD matching algorithm for two input windows. Include this code in your report (in appendix as usual).
Solution:
To implement the SSD matching algorithm, we simply vectorizes the two input windows and then take the 2-norm of their difference.

6.3 Naive Matching [4pt]
Equipped with the corner detector and the SSD matching code, 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. In your report, only include your results for 10 corners, so that the figure will not be cluttered. `naiveCorrespondanceMatching.m` will call your SSD mathch- ing code. The parameter `R` below, is the radius of the patch used for matching.

```
ncorners = 10;
corners1 = CornerDetect(I1, ncorners, smoothSTD, windowSize));
corners2 = CornerDetect(I2, ncorners, smoothSTD, windowSize));
[I, corsSSD] = naiveCorrespondanceMatching(I1, I2, corners1, corners2, R, SSDth);
```

Solution:
We simply follow the instruction above to search the best match in image2 for every corner in image1. We adjust the SSD threshold so that it can have the highest percentage of correct matches. In the end, we got 4 correct matches among 7 matches in Warrior images and 3 correct matches among 9 matches in Matrix images.

Result:

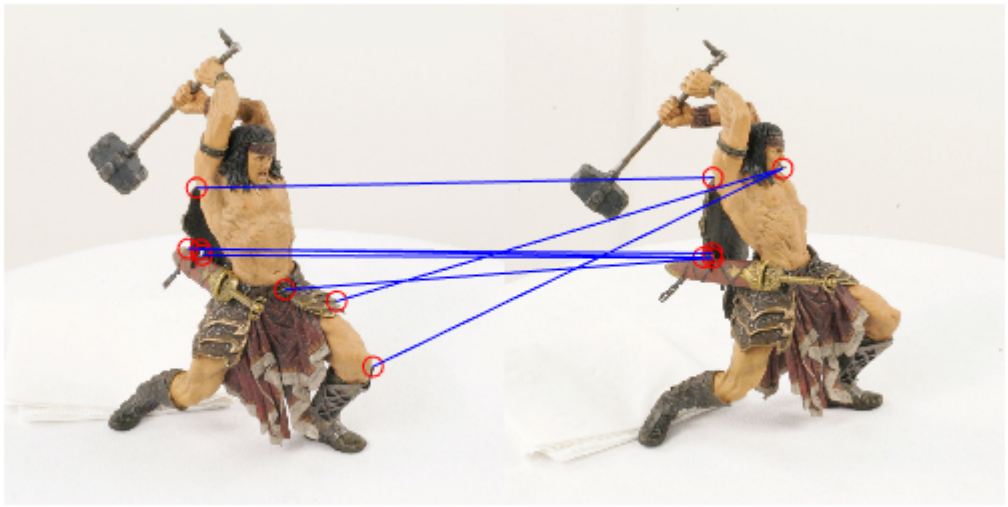


Figure 11: Naive Matching: Warrior

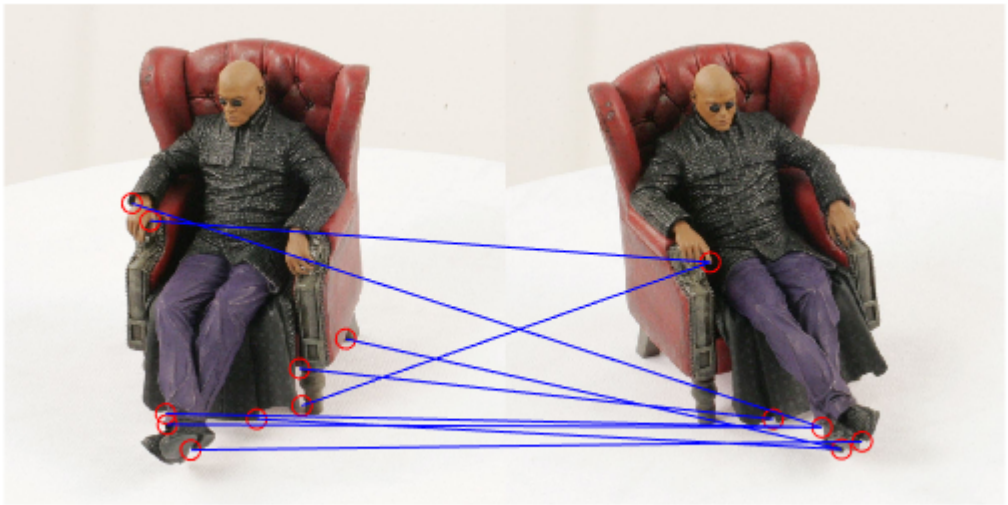


Figure 12: Naive Matching: Matrix

6.4 Epipolar Geometry [3pt]
Using the provided `fund.m` together with the provided points `cor1`, and `cor2`, calculate the fundamental matrix, and then plot the epipolar lines in both images pairs as shown below. Plot the points and make sure the epipolar lines go through them. You might find the supplied `linePts.m` function useful when you are working with epipolar lines.
Solution:
To compute the epipolar line in image1 for the the corner in image2, we use $l_1 = \mathbf{F}^T p_2$
To compute the epipolar line in image2 for the the corner in image1, we use $l_2 = \mathbf{F} p_1$
The **F** above is the fundimental matrix.
Result:

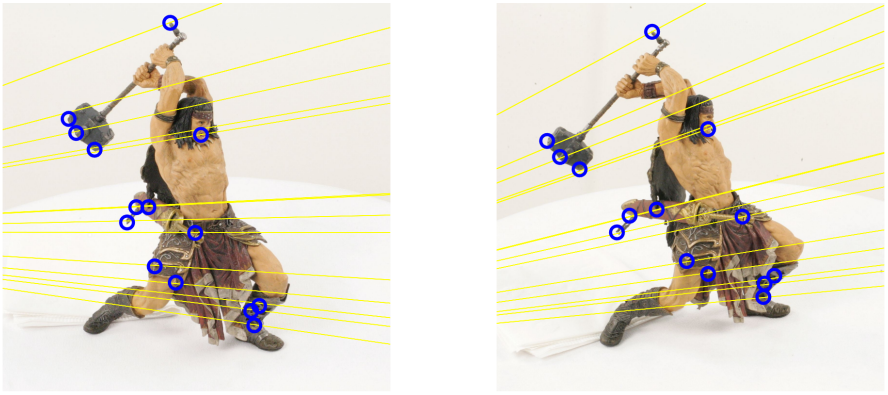


Figure 13: Epipolar Lines: Warrior

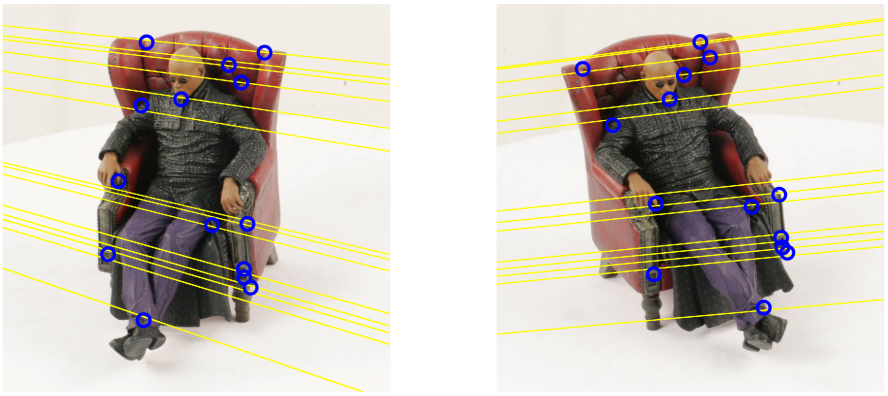


Figure 14: Epipolar Lines: Matrix

6.5 Epipolar Geometry Based Matching [5pt]
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, but only as indicated in the code below.

```
ncorners = 10;
F = fund(cor1, cor2);
corners1 = CornerDetect(I1, ncorners, smoothSTD, windowSize));
corsSSD = correspondanceMatchingLine( I1, I2, corners1, F, R, SSDth);
```

Solution:
To do the linesearch, we have to interpolate the pixels from the line. We interpolate the pixels by first using the supplied `linePts.m` to find the two end points of the line. Then for every x and y in between these end points, we compute a corresponding y and x by the slope of the line.
The other processes are as same as the processes in 6.3.

Result:

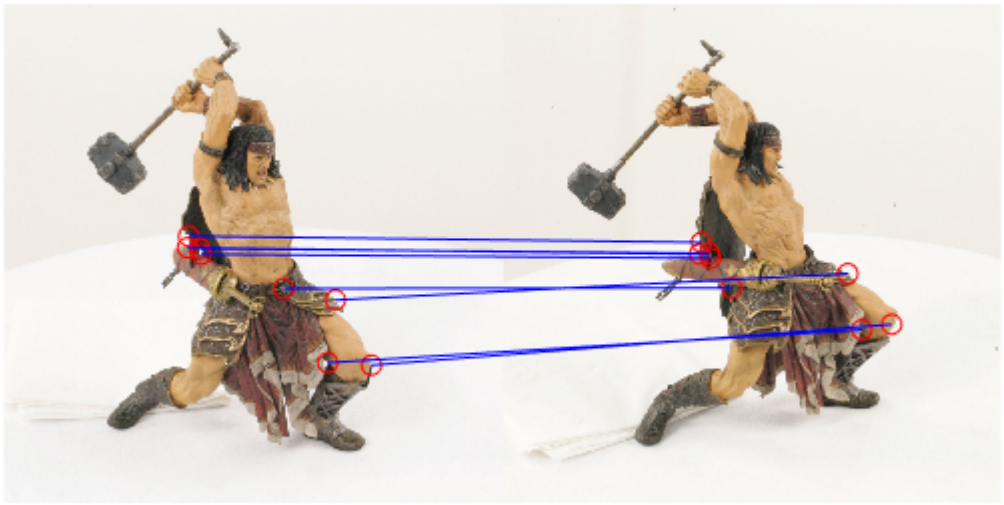


Figure 15: Epipolar Geometry Based Matching: Warrior

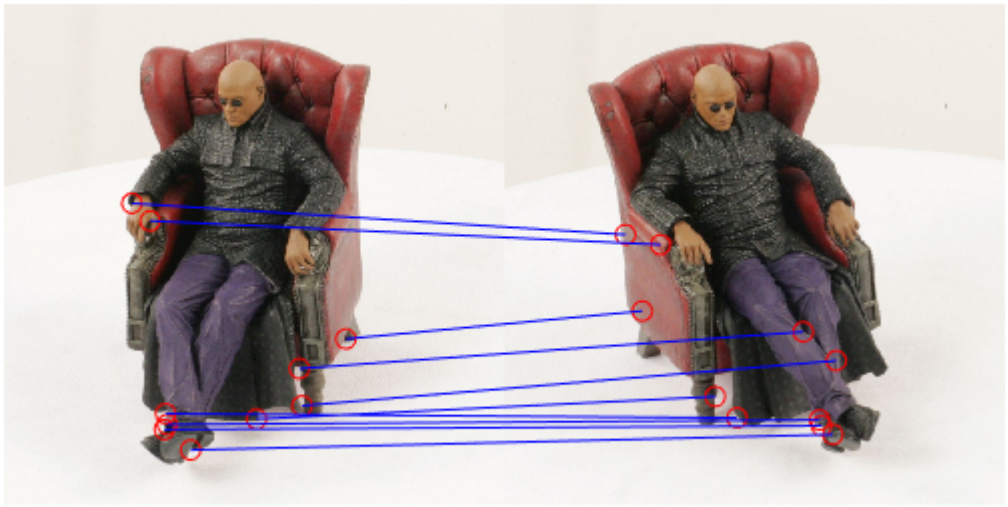


Figure 16: Epipolar Geometry Based Matching: Matrix

6.6 Triangulation [5pt]
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. `P1` and `P2` below are the camera matrices. Also write a function, `findOutliers.m`, 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.

```
outlierTH = 20;
F = fund(cor1, cor2);
ncorners = 50;
corners1 = CornerDetect(I1, ncorners, smoothSTD, windowSize));
corsSSD = correspondanceMatchingLine( I1, I2, corners1, F, R, SSDth);
points3D = triangulate(corsSSD, P1, P2);
[ inlier, outlier ] = findOutliers(points3D, P2, outlierTH, corsSSD);
```

Display your results by showing, for Image2, the original points in black circles, the inliers as blue plus signs, and the outliers as red plus signs, as shown in Figure 7. Compare this outlierplot with Figure 6. Does the detected outliers correspond to false matches?

Solution: To implement the triangulation, we use the linear triangulation method described in lecture. First, for a pair of corresponding corners in image1 and image2, we each construct two lines that passes the corner. We choose the lines simply by

$$\begin{aligned} x + y - (x_1 + y_1) &= 0 \\ x - y - (x_1 - y_1) &= 0 \\ x + y - (x_2 + y_2) &= 0 \\ x - y - (x_2 - y_2) &= 0 \end{aligned}$$

, where$(x_1, y_1), (x_2, y_2)$ are the corners in image1 and image2.
Follow the method mentioned in the lecture, we construct the matrix $\mathbf{A}$, where

$$\mathbf{A} = \begin{pmatrix} 1 & 1 & -(x_1 + y_1) \\ 1 & -1 & -(x_1 - y_1) \\ 1 & 1 & -(x_2 + y_2) \\ 1 & -1 & -(x_2 - y_2) \end{pmatrix}$$

We solve the null space for $\mathbf{AX} = 0$ by using singular value decomposition and we take the last column of matrix $\mathbf{V}$ as our answer.
To detect outliers, we project the 3D points obtained by triangulation back to image2 and see if the distance between the projection location and the original point is greater than `outlierTH`. We project the 3D point to image2 by $\mathbf{x}_2(reproject) = \mathbf{P}_2\mathbf{X}$, where $\mathbf{P}_2$ is the camera matrix for image2.

Result:

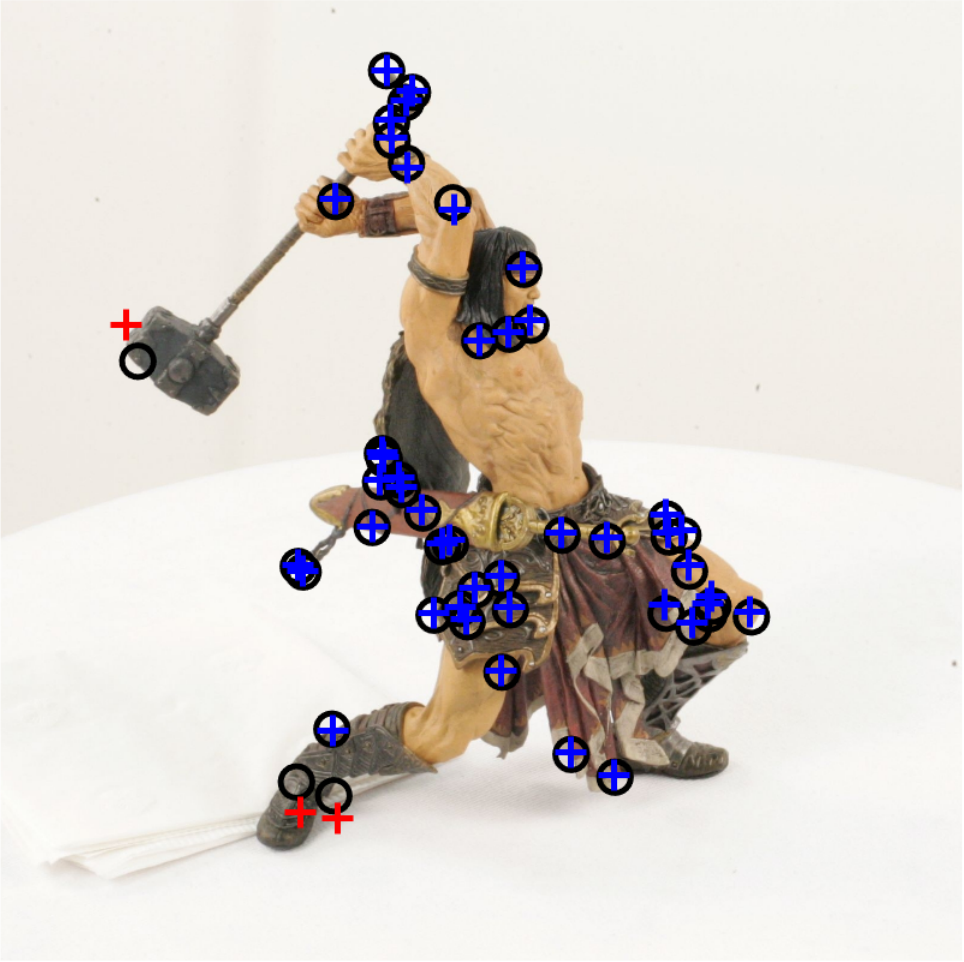


Figure 17: Result of Reprojection: Warrior

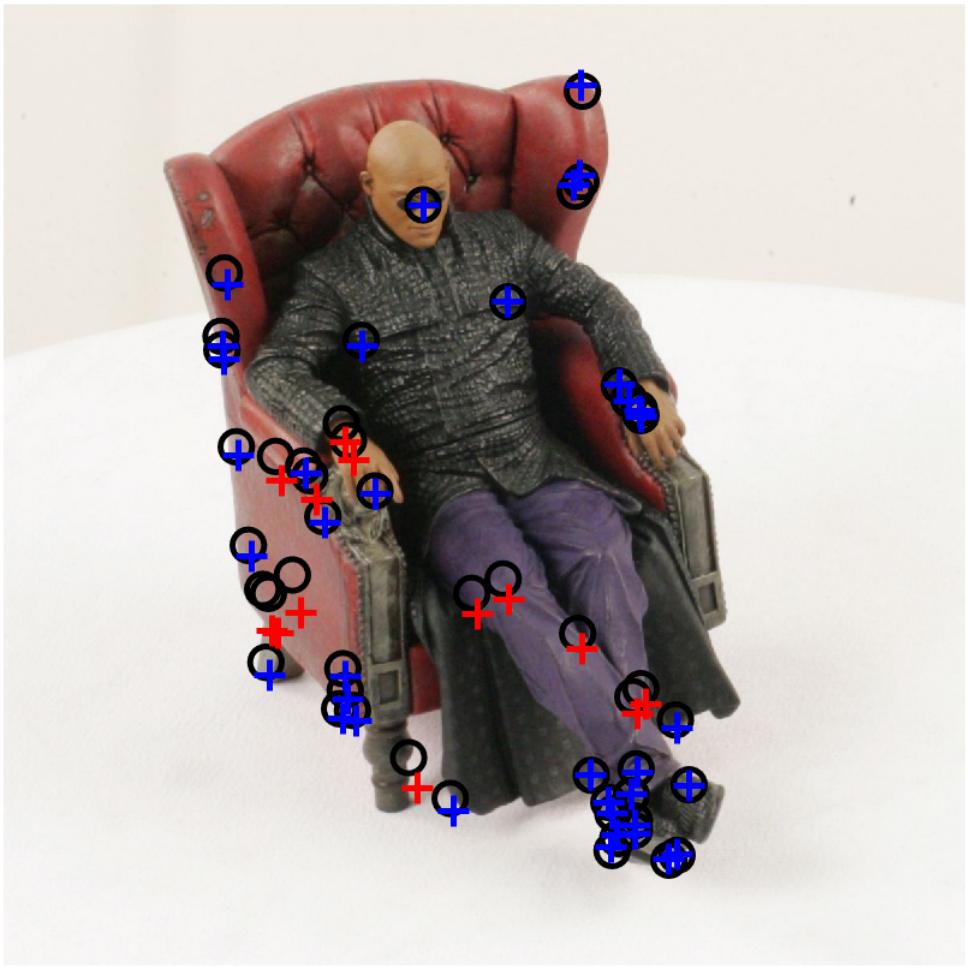


Figure 18: Result of Reprojection: Matrix

Appendix

Listing 1: Script of Problem 4

```
clear
clc
%% Problem 4.1 Warmup
filter_img = loadAndPreprocess('filter.jpg');

toy_ori_img = imread('toy.png');
toy_img = loadAndPreprocess('toy.png');
toy_conv = conv2(toy_img, filter_img, 'same');
figure;
imagesc(toy_conv);

% find maximum intensity
max_intensity = max(toy_conv(:));

max_index = zeros(1,2,3);
count = 1;
for i = 1:size(toy_ori_img, 1)
    for j = 1:size(toy_ori_img, 2)
        if toy_conv(i,j) == max_intensity
            max_index(1,1,count) = i;
            max_index(1,2,count) = j;
            count = count + 1;
        end
    end
end
len = size(filter_img, 1);
width = size(filter_img, 2);
figure;
imshow(toy_ori_img);
hold on;
for i = 1:3
    y = max_index(1,1,i)-len/2;
    x = max_index(1,2,i)-width/2;
    rectangle('Position',[x y width len],'EdgeColor','r');
end
%% Problem 4.2 Detection Error
figure;
imshow(toy_ori_img);
hold on;
ground_truth_box_1 = [322,213,105,105];
ground_truth_box_2 = [325,215,110,110];
ground_truth_box_3 = [320,220,120,120];
rect_1 = [x y width len];
```

```
44 rectangle('Position',rect_1, 'EdgeColor','r');
45 rectangle('Position',ground_truth_box_3, 'EdgeColor','r');
46 overlap = detectError(rect_1, ground_truth_box_3);
47 %% Problem 4.3 More Realistic Images
48 car_template_img = loadAndPreprocess('cartemplate.jpg');
49
50 % car1
51 car1_ori_img = imread('car1.jpg');
52 car1_img = loadAndPreprocess('car1.jpg');
53 car1_in = fopen('car1.txt');
54 car1_rect = textscan(car1_in,'%d %d');
55 box_len = car1_rect{1}(2)- car1_rect{1}(1);
56 box_width = car1_rect{2}(2)- car1_rect{2}(1);
57 car1_box = [car1_rect{1}(1) car1_rect{2}(1) box_len box_width];
58 % car1 filter preprocessing
59 car1_template = imcrop(car_template_img, [90 280 1630 620]);
60 car1_template = imresize(car1_template,[box_width box_len]);
61 % car1 overlap finding process
62 heat_map1 = constructHeatMap(car1_img, car1_template);
63 rect_car1 = constructBoundingBox(car1_ori_img, heat_map1, car1_box);
64 % rectangle('Position',car1_box, 'EdgeColor','b');
65 overlap1 = detectError(rect_car1, car1_box);
66 %% car2
67 car2_ori_img = imread('car2.jpg');
68 car2_img = loadAndPreprocess('car2.jpg');
69 car2_in = fopen('car2.txt');
70 car2_rect = textscan(car2_in,'%d %d');
71 box_len = car2_rect{1}(2)- car2_rect{1}(1);
72 box_width = car2_rect{2}(2)- car2_rect{2}(1);
73 car2_box = [car2_rect{1}(1) car2_rect{2}(1) box_len box_width];
74 % car2 filter preprocessing
75 car2_template = imcrop(car_template_img, [90 280 1630 620]);
76 car2_template = imresize(car2_template, [box_width box_len]);
77 car2_template = flip(car2_template,2);
78 % car1 overlap finding process
79 heat_map2 = constructHeatMap(car2_img, car2_template);
80 rect_car2 = constructBoundingBox(car2_ori_img, heat_map2, car2_box);
81 % rectangle('Position',car2_box, 'EdgeColor','b');
82 overlap2 = detectError(rect_car2, car2_box);
83 %% car3
84 car3_ori_img = imread('car3.jpg');
85 car3_img = loadAndPreprocess('car3.jpg');
86 car3_in = fopen('car3.txt');
87 car3_rect = textscan(car3_in,'%d %d');
88 box_len = car3_rect{1}(2)- car3_rect{1}(1);
89 box_width = car3_rect{2}(2)- car3_rect{2}(1);
```

```
car3_box = [car3_rect{1}(1) car3_rect{2}(1) box_len box_width];
% car3 filter preprocess
car3_template = imcrop(car_template_img, [90 260 1630 640]);
car3_template = imresize(car3_template, [68 95]);
% car3 overlap finding process
heat_map3 = constructHeatMap(car3_img, car3_template);
rect_car3 = constructBoundingBox(car3_ori_img, heat_map3, car3_box);
% rectangle('Position',car3_box, 'EdgeColor','b');
overlap3 = detectError(rect_car3, car3_box);
%% car4
car4_ori_img = imread('car4.jpg');
car4_img = loadAndPreprocess('car4.jpg');
car4_in = fopen('car4.txt');
car4_rect = textscan(car4_in,'%d %d');
box_len = car4_rect{1}(2)- car4_rect{1}(1);
box_width = car4_rect{2}(2)- car4_rect{2}(1);
car4_box = [car4_rect{1}(1) car4_rect{2}(1) box_len box_width];
% car4 filter preprocess
car4_template = imcrop(car_template_img, [90 280 1630 620]);
car4_template = car4_template + 0.1;
car4_template = imresize(car4_template, [box_width box_len]);
car4_template = flip(car4_template,2);
% car4 overlap finding process
heat_map4 = constructHeatMap(car4_img, car4_template);
rect_car4 = constructBoundingBox(car4_ori_img, heat_map4, car4_box);
% rectangle('Position',car4_box, 'EdgeColor','b');
overlap4 = detectError(rect_car4, car4_box);
%% car5
car5_ori_img = imread('car5.jpg');
car5_img = loadAndPreprocess('car5.jpg');
car5_in = fopen('car5.txt');
car5_rect = textscan(car5_in,'%d %d');
box_len = car5_rect{1}(2)- car5_rect{1}(1);
box_width = car5_rect{2}(2)- car5_rect{2}(1);
car5_box = [car5_rect{1}(1) car5_rect{2}(1) box_len box_width];
% car5 filter preprocess
car5_template = imcrop(car_template_img, [90 280 1630 620]);
car5_template = imresize(car5_template,[box_width box_len]);
car5_template = car5_template + 0.1;
car5_template = flip(car5_template,2);
% car5 overlap finding process
heat_map5 = constructHeatMap(car5_img, car5_template);
rect_car5 = constructBoundingBox(car5_ori_img, heat_map5, car5_box);
% rectangle('Position',car5_box, 'EdgeColor','b');
overlap5 = detectError(rect_car5, car5_box);
```

Listing 2: loadAndPreprocess.m used in Script of Problem 4

```
function [ output_img ] = loadAndPreprocess( input_img )
    output_img = imread(input_img);
    output_img = im2double(output_img);
    output_img = output_img - mean(output_img(:));
end
```

Listing 3: detectError.m used in Script of Problem 4

```
function [ overlap ] = detectError( predicted_box, ground_truth_box)
    intersection_area = rectint(predicted_box, ground_truth_box);
    union_area = predicted_box(3)*predicted_box(4) + ...
        ground_truth_box(3)*ground_truth_box(4) - intersection_area;
    overlap = double(intersection_area)/double(union_area);
end
```

Listing 4: constructHeatMap.m used in Script of Problem 4

```
function [ heat_map ] = constructHeatMap( input_img, filter_img )
    filter_img = flip(filter_img, 1);
    filter_img = flip(filter_img, 2);
    heat_map = conv2(input_img, filter_img, 'same');
    figure;
    imagesc(heat_map);
end
```

Listing 5: constructBoundingBox.m used in Script of Problem 4

```
function [ rect ] = constructBoundingBox( ori_img, heat_map, ...
    ground_truth_box )
    % find maximum intensity
    max_intensity = 0;
    max_index = zeros(1,2);
    for i = 1:size(ori_img, 1)
       for j = 1:size(ori_img, 2)
           max_intensity = max(heat_map(i,j), max_intensity);
       end
    end
    for i = 1:size(ori_img, 1)
        for j = 1:size(ori_img, 2)
            if heat_map(i,j) == max_intensity
                max_index(1,1) = i;
                max_index(1,2) = j;
            end
        end
    end
    len = ground_truth_box(4);
```

```
    width = ground_truth_box(3);
%     len = size(filter_img, 1);
%     width = size(filter_img, 2);
    figure;
    imshow(ori_img);
    hold on;
    y = max_index(1,1)-len/2;
    x = max_index(1,2)-width/2;
    rectangle('Position',[x y width len],'EdgeColor','b', 'LineWidth', 3);
    rect = [x y width len];
end
```

Listing 6: Script of Problem 5

```
clc;
%% Problem 5.1 Smoothing
geisel = imread('geisel.jpeg');
geisel = im2double(geisel);
figure;
imshow(geisel);
title('Input Image');
K = 1/159*[2,4,5,4,2;4,9,12,9,4;5,12,15,12,5;4,9,12,9,4;2,4,5,4,2];
gaussian = conv2(geisel, K, 'same');
figure;
imshow(gaussian);
title('Gaussian Filter with sigma = 1.4');
%% Problem 5.2 Gradient Computation
sx = [-1,0,1;-2,0,2;-1,0,1];
sy = [-1,-2,-1;0,0,0;1,2,1];
Gx = conv2(gaussian, sx, 'same');
Gy = conv2(gaussian, sy, 'same');
for i = 1:length(Gx)
    for j = 1:length(Gy)
        G(i,j) = sqrt(Gx(i,j)^2 + Gy(i,j)^2);
        edge_direction(i,j) = atan2(Gy(i,j), Gx(i,j));
        edge_direction(i,j) = rad2deg(edge_direction(i,j));
        if (edge_direction(i,j) < 0)
            edge_direction(i,j) = edge_direction(i,j) + 180;
        end
    end
end
figure;
imshow(G);
title('Gradient Magnitude');
%% Problem 5.3 Non Maximum Suppression
edge_candidate = zeros(size(G));
for i = 2:length(G)-1
```

```
34      for j = 2:length(G)-1
35          round_direction(i,j) = round2nearest45(edge_direction(i,j));
36          switch(round_direction(i,j))
37              case 0 % east and west directions
38                  if G(i,j) > G(i,j-1) && G(i,j) > G(i,j+1)
39                      edge_candidate(i,j) = G(i,j);
40                  end
41              case 45 % north east and south west directions
42                  if G(i,j) > G(i-1,j+1) && G(i,j) > G(i+1,j-1)
43                      edge_candidate(i,j) = G(i,j);
44                  end
45              case 90 % north and south directions
46                  if G(i,j) > G(i-1,j) && G(i,j) > G(i+1,j)
47                      edge_candidate(i,j) = G(i,j);
48                  end
49              case 135 % north west and south east directions
50                  if G(i,j) > G(i-1,j-1) && G(i,j) > G(i+1,j+1)
51                      edge_candidate(i,j) = G(i,j);
52                  end
53          end
54      end
55  end
56  figure;
57  imshow(edge_candidate);
58  title('Non_Maximum_Suppression');
59  %% Problem 5.4 Hysteresis Thresholding
60  high = 0.4;
61  low = 0.1;
62  A = [];
63  for i = 2:length(edge_candidate)-1
64      for j = 2:length(edge_candidate)-1
65          if edge_candidate(i,j) > high
66              A = [A; i j];
67          end
68      end
69  end
70  result = zeros(size(edge_candidate));
71  while ~isempty(A)
72      cur = A(size(A,1),:);
73      A(size(A,1),:) = [];
74      result(cur(1),cur(2)) = 1;
75      if cur(1)>1 && cur(1)<length(edge_candidate)-1 && cur(2)>1 ...
76              && cur(2)<length(edge_candidate)-1
77          if edge_candidate(cur(1)-1,cur(2)-1) > low ...
78                  && result(cur(1)-1,cur(2)-1) == 0
79              A = [A; cur(1)-1 cur(2)-1];
```

```
80          elseif edge_candidate(cur(1)-1,cur(2)) > low ...
81                  && result(cur(1)-1,cur(2)) == 0
82              A = [A; cur(1)-1 cur(2)];
83          elseif edge_candidate(cur(1)-1,cur(2)+1) > low ...
84                  && result(cur(1)-1,cur(2)+1) == 0
85              A = [A; cur(1)-1 cur(2)+1];
86          elseif edge_candidate(cur(1),cur(2)-1) > low ...
87                  && result(cur(1),cur(2)-1) == 0
88              A = [A; cur(1) cur(2)-1];
89          elseif edge_candidate(cur(1),cur(2)+1) > low ...
90                  && result(cur(1),cur(2)+1) == 0
91              A = [A; cur(1) cur(2)+1];
92          elseif edge_candidate(cur(1)+1,cur(2)-1) > low ...
93                  && result(cur(1)+1,cur(2)-1) == 0
94              A = [A; cur(1)+1 cur(2)-1];
95          elseif edge_candidate(cur(1)+1,cur(2)) > low ...
96                  && result(cur(1)+1,cur(2)) == 0
97              A = [A; cur(1)+1 cur(2)];
98          elseif edge_candidate(cur(1)+1,cur(2)+1) > low ...
99                  && result(cur(1)+1,cur(2)+1) == 0
100             A = [A; cur(1)+1 cur(2)+1];
101         end
102     end
103 end
104 figure;
105 imshow(result);
106 title('Hysteresis Thresholding');
```

Listing 7: round2nearest45.m used in Script of Problem 5

```
1  function [ degree_out ] = round2nearest45( degree_in )
2      if 0 < degree_in && degree_in < 22.5
3          degree_out = 0;
4      elseif 22.5 <= degree_in && degree_in < 67.5
5          degree_out = 45;
6      elseif 67.5 <= degree_in && degree_in < 112.5
7          degree_out = 90;
8      elseif 112.5 <= degree_in && degree_in < 157.5
9          degree_out = 135;
10     else
11         degree_out = 0;
12     end
13 end
```

Listing 8: Script of Problem 6

```
1  clc;
```

```
2
3  %% 6.1 dino
4  % clear all;
5  % load('dino2.mat');
6  %
7  % nCorners = 20;
8  % smoothSTD = 1;
9  % windowSize = 10;
10 % corners_1 = CornerDetect(dino01, nCorners, smoothSTD, windowSize);
11 % corners_2 = CornerDetect(dino02, nCorners, smoothSTD, windowSize);
12 %
13 % figure;
14 % showMarker1(dino01, corners_1);
15 % figure;
16 % showMarker1(dino02, corners_2);
17
18 %% 6.1 warrior
19 clear all;
20 load('warrior2.mat');
21
22 nCorners = 20;
23 smoothSTD = 1;
24 windowSize = 10;
25 corners_1 = CornerDetect(warrior01, nCorners, smoothSTD, windowSize);
26 corners_2 = CornerDetect(warrior02, nCorners, smoothSTD, windowSize);
27
28 figure;
29 showMarker1(warrior01, corners_1);
30 figure;
31 showMarker1(warrior02, corners_2);
32
33 %% 6.1 matrix
34 clear all;
35 load('matrix2.mat');
36
37 nCorners = 20;
38 smoothSTD = 3;
39 windowSize = 10;
40 corners_1 = CornerDetect(matrix01, nCorners, smoothSTD, windowSize);
41 corners_2 = CornerDetect(matrix02, nCorners, smoothSTD, windowSize);
42
43 figure;
44 showMarker1(matrix01, corners_1);
45 figure;
46 showMarker1(matrix02, corners_2);
47
```

```
%% 6.2 6.3 dino
% clear all;
% load('dino2.mat');
%
% nCorners = 10;
% smoothSTD = 1;
% windowSize = 10;
% corners_1 = CornerDetect(dino01, nCorners, smoothSTD, windowSize);
% corners_2 = CornerDetect(dino02, nCorners, smoothSTD, windowSize);
%
% R = 20;
% SSDth = 70;
% corsSSD = naiveCorrespondanceMatching(dino01, dino02, corners_1, corners_2, R, SSDth
%
% figure;
% showMatching(dino01,dino02,corners_1,corsSSD);

%% 6.2 6.3 warrior
clear all;
load('warrior2.mat');

nCorners = 10;
smoothSTD = 1;
windowSize = 10;
corners_1 = CornerDetect(warrior01, nCorners, smoothSTD, windowSize);
corners_2 = CornerDetect(warrior02, nCorners, smoothSTD, windowSize);

R = 20;
SSDth = 100;
corsSSD = naiveCorrespondanceMatching(warrior01, warrior02, corners_1, corners_2, R, S

figure;
showMatching(warrior01,warrior02,corners_1,corsSSD);

%% 6.2 6.3 matrix
clear all;
load('matrix2.mat');

nCorners = 10;
smoothSTD = 3;
windowSize = 10;
corners_1 = CornerDetect(matrix01, nCorners, smoothSTD, windowSize);
corners_2 = CornerDetect(matrix02, nCorners, smoothSTD, windowSize);

R = 20;
SSDth = 190;
```

```
94  corsSSD = naiveCorrespondanceMatching(matrix01, matrix02, corners_1, corners_2, R, SSD
95
96  figure;
97  showMatching(matrix01,matrix02,corners_1,corsSSD);
98
99  %% 6.4 dino
100 % clear all;
101 % load('dino2.mat');
102 % FMatrix = fund(cor1, cor2);
103 % epiLine_1 = [];
104 % epiLine_2 = [];
105 % for i = 1:size(cor1, 1)
106 %     line_1 = linePts(FMatrix' *[cor2(i,:),1]', [1, size(dino01, 2)], [1, size(dino01
107 %     epiLine_1 = [epiLine_1; line_1(1,:), line_1(2,:)];
108 %     line_2 = linePts(FMatrix  *[cor1(i,:),1]', [1, size(dino02, 2)], [1, size(dino02
109 %     epiLine_2 = [epiLine_2; line_2(1,:), line_2(2,:)];
110 % end
111 %
112 % figure;
113 % img_1 = insertShape(dino01,'Line',epiLine_1,'LineWidth',5);
114 % showMarker1(img_1, cor1);
115 % figure;
116 % img_2 = insertShape(dino02,'Line',epiLine_2,'LineWidth',5);
117 % showMarker1(img_2, cor2);
118
119 %% 6.4 warrior
120 clear all;
121 load('warrior2.mat');
122 FMatrix = fund(cor1, cor2);
123 epiLine_1 = [];
124 epiLine_2 = [];
125 for i = 1:size(cor1, 1)
126     line_1 = linePts(FMatrix' *[cor2(i,:),1]', [1, size(warrior01, 2)], [1, size(warrio
127     epiLine_1 = [epiLine_1; line_1(1,:), line_1(2,:)];
128     line_2 = linePts(FMatrix  *[cor1(i,:),1]', [1, size(warrior02, 2)], [1, size(warrio
129     epiLine_2 = [epiLine_2; line_2(1,:), line_2(2,:)];
130 end
131
132 figure;
133 img_1 = insertShape(warrior01,'Line',epiLine_1,'LineWidth',5);
134 showMarker1(img_1, cor1);
135 figure;
136 img_2 = insertShape(warrior02,'Line',epiLine_2,'LineWidth',5);
137 showMarker1(img_2, cor2);
138
139 %% 6.4 matrix
```

```
clear all;
load('matrix2.mat');
FMatrix = fund(cor1, cor2);
epiLine_1 = [];
epiLine_2 = [];
for i = 1:size(cor1, 1)
    line_1 = linePts(FMatrix' *[cor2(i,:),1]', [1, size(matrix01, 2)], [1, size(matrix0
    epiLine_1 = [epiLine_1; line_1(1,:), line_1(2,:)];
    line_2 = linePts(FMatrix  *[cor1(i,:),1]', [1, size(matrix02, 2)], [1, size(matrix0
    epiLine_2 = [epiLine_2; line_2(1,:), line_2(2,:)];
end

figure;
img_1 = insertShape(matrix01,'Line',epiLine_1,'LineWidth',5);
showMarker1(img_1, cor1);
figure;
img_2 = insertShape(matrix02,'Line',epiLine_2,'LineWidth',5);
showMarker1(img_2, cor2);

%% 6.5 dino
% clear all;
% load('dino2.mat');
%
% nCorners = 10;
% smoothSTD = 1;
% windowSize = 10;
% corners_1 = CornerDetect(dino01, nCorners, smoothSTD, windowSize);
%
% FMatrix = fund(cor1, cor2);
% R = 20;
% SSDth = 70;
% corsSSD = correspondanceMatchingLine(dino01, dino02, corners_1, FMatrix, R, SSDth);
%
% figure;
% showMatching(dino01,dino02,corners_1,corsSSD);

%% 6.5 warrior
clear all;
load('warrior2.mat');

nCorners = 10;
smoothSTD = 1;
windowSize = 10;
corners_1 = CornerDetect(warrior01, nCorners, smoothSTD, windowSize);

FMatrix = fund(cor1, cor2);
```

```
186 R = 20;
187 SSDth = 90;
188 corsSSD = correspondanceMatchingLine(warrior01, warrior02, corners_1, FMatrix, R, SSDt
189
190 figure;
191 showMatching(warrior01,warrior02,corners_1,corsSSD);
192
193 %% 6.5 matrix
194 clear all;
195 load('matrix2.mat');
196
197 nCorners = 10;
198 smoothSTD = 3;
199 windowSize = 10;
200 corners_1 = CornerDetect(matrix01, nCorners, smoothSTD, windowSize);
201
202 FMatrix = fund(cor1, cor2);
203 R = 20;
204 SSDth = 2000;
205 corsSSD = correspondanceMatchingLine(matrix01, matrix02, corners_1, FMatrix, R, SSDth)
206
207 figure;
208 showMatching(matrix01,matrix02,corners_1,corsSSD);
209
210 %% 6.6 dino
211 % clear all;
212 % load('dino2.mat');
213 %
214 % nCorners = 50;
215 % smoothSTD = 1;
216 % windowSize = 10;
217 % corners_1 = CornerDetect(dino01, nCorners, smoothSTD, windowSize);
218 %
219 % FMatrix = fund(cor1, cor2);
220 % R = 20;
221 % SSDth = 70;
222 % corsSSD = correspondanceMatchingLine(dino01, dino02, corners_1, FMatrix, R, SSDth);
223 %
224 % points3D = Triangulate(corners_1, corsSSD, proj_7, proj_8);
225 %
226 % outlierTH = 20;
227 % [inlier, outlier] = findOutliers(points3D, proj_8, outlierTH, corsSSD);
228 %
229 % figure;
230 % showMarker3(dino02, corsSSD, inlier, outlier);
231
```

```
232 %% 6.6 warrior
233 clear all;
234 load('warrior2.mat');
235
236 nCorners = 50;
237 smoothSTD = 1;
238 windowSize = 10;
239 corners_1 = CornerDetect(warrior01, nCorners, smoothSTD, windowSize);
240
241 FMatrix = fund(cor1, cor2);
242 R = 20;
243 SSDth = 90;
244 corsSSD = correspondanceMatchingLine(warrior01, warrior02, corners_1, FMatrix, R, SSDt
245
246 points3D = Triangulate(corners_1, corsSSD, proj_warrior01, proj_warrior02);
247
248 outlierTH = 20;
249 [inlier, outlier] = findOutliers(points3D, proj_warrior02, outlierTH, corsSSD);
250
251 figure;
252 showMarker3(warrior02, corsSSD, inlier, outlier);
253
254 %% 6.6 matrix
255 clear all;
256 load('matrix2.mat');
257
258 nCorners = 50;
259 smoothSTD = 3;
260 windowSize = 10;
261 corners_1 = CornerDetect(matrix01, nCorners, smoothSTD, windowSize);
262
263 FMatrix = fund(cor1, cor2);
264 R = 20;
265 SSDth = 2000;
266 corsSSD = correspondanceMatchingLine(matrix01, matrix02, corners_1, FMatrix, R, SSDth)
267
268 points3D = Triangulate(corners_1, corsSSD, proj_matrix01, proj_matrix02);
269
270 outlierTH = 20;
271 [inlier, outlier] = findOutliers(points3D, proj_matrix02, outlierTH, corsSSD);
272
273 figure;
274 showMarker3(matrix02, corsSSD, inlier, outlier);
```

Listing 9: CornerDetect.m used in Script of Problem 6

```
1 function corners = CornerDetect(Image, nCorners, smoothSTD, windowSize)
```

```
Image_blur = imgaussfilt(rgb2gray(Image),smoothSTD);
[gradient_Y, gradient_X] =  gradient(Image_blur);
windowDiameter = fix(windowSize/2);
minEigen = zeros(size(Image, 1), size(Image, 2));
for i = (1 + windowDiameter):(size(Image, 1) - windowDiameter)
    for j = (1 + windowDiameter):(size(Image, 2) - windowDiameter)
        window_gX = gradient_X(i - windowDiameter: i + windowDiameter, j - windowDiame
        window_gY = gradient_Y(i - windowDiameter: i + windowDiameter, j - windowDiame
        C = zeros(2, 2);
        C(1, 1) = sum(window_gX(:).^2);
        C(2, 2) = sum(window_gY(:).^2);
        C(1, 2) = sum(window_gX(:).*window_gY(:));
        C(2, 1) = C(1, 2);
        e = eig(C);
        minEigen(i, j) = min(e);
    end
end
minEigen_NMS = [];
for i = (1 + windowDiameter):(size(minEigen, 1) - windowDiameter)
    for j = (1 + windowDiameter):(size(minEigen, 2) - windowDiameter)
        localArea = minEigen(i - windowDiameter:i + windowDiameter, j - windowDiameter
        if minEigen(i, j) == max(localArea(:))
            minEigen_NMS = [minEigen_NMS; minEigen(i, j), j, i];
        end
    end
end
[B, idx] = sort(minEigen_NMS(:, 1), 'descend');
corners = [];
for i = 1:nCorners
    corners = [corners; minEigen_NMS(idx(i), 2:3)];
end

end
```

Listing 10: SSDmatch.m used in Script of Problem 6

```
function SSD = SSDmatch( window_1, window_2 )
SSD = (norm(window_1(:)-window_2(:)))^2;
end
```

Listing 11: naiveCorrespondanceMatching.m used in Script of Problem 6

```
function corsSSD = naiveCorrespondanceMatching(I1, I2, corners1, corners2, R, SSDth)
I1 = rgb2gray(I1);
I2 = rgb2gray(I2);
corsSSD = [];
for i = 1:size(corners1, 1)
```

```
6       minSSD = 10000000;
7       bestMatch = 0;
8       for j = 1:size(corners2, 1)
9           window_1 = I1(corners1(i, 2)-R:corners1(i, 2)+R, corners1(i, 1)-R:corners1(i,
10          window_2 = I2(corners2(j, 2)-R:corners2(j, 2)+R, corners2(j, 1)-R:corners2(j,
11          if SSDmatch(window_1, window_2) < minSSD
12              minSSD = SSDmatch(window_1, window_2);
13              bestMatch = j;
14          end
15      end
16      if minSSD < SSDth
17          corsSSD = [corsSSD; corners2(bestMatch, :)];
18      else
19          corsSSD = [corsSSD; 0, 0];
20      end
21  end
22  end
```

Listing 12: correspondanceMatchingLine.m used in Script of Problem 6

```
1   function corsSSD = correspondanceMatchingLine(I1, I2, corners1, FMatrix, R, SSDth)
2   epiLine_dino02 = [];
3   for i = 1:size(corners1, 1)
4       line_2 = linePts(FMatrix  *[corners1(i,:),1]', [1, size(I2, 2)], [1, size(I2, 1)])
5       epiLine_dino02 = [epiLine_dino02; line_2(1,:), line_2(2,:)];
6   end
7   I1 = rgb2gray(I1);
8   I2 = rgb2gray(I2);
9   corsSSD = [];
10  for i = 1:size(epiLine_dino02, 1)
11      points = interpolateLine(epiLine_dino02(i, :));
12      minSSD = 10000000;
13      bestMatch = 0;
14      window_1 = I1(corners1(i,2)-R:corners1(i,2)+R, corners1(i,1)-R:corners1(i,1)+R);
15      for j = 1:size(points, 1)
16          if points(j, 2)-R > 0 && points(j, 2)+R < size(I2,1) && points(j, 1)-R > 0 &&
17              window_2 = I2(points(j, 2)-R:points(j, 2)+R, points(j, 1)-R:points(j, 1)+R
18              if SSDmatch(window_1, window_2) < minSSD
19                  minSSD = SSDmatch(window_1, window_2);
20                  bestMatch = j;
21              end
22          end
23      end
24      if minSSD < SSDth
25          corsSSD = [corsSSD; points(bestMatch, :)];
26      else
27          corsSSD = [corsSSD; 0, 0];
```

```
28      end
29  end
30  corsSSD = double(corsSSD);
31
32  end
```

Listing 13: Triangulate.m used in Script of Problem 6

```
1   function points3D = Triangulate(corners1, corsSSD, P1, P2)
2   points3D = [];
3   for i = 1:size(corners1, 1)
4       if corsSSD(i, :) == [0, 0]
5           points3D = [points3D; 0, 0, 0, 0];
6       else
7           L_1 = [1, 1, -corners1(i, 1)-corners1(i, 2);
8                  1,-1, -corners1(i, 1)+corners1(i, 2)];
9           L_2 = [1, 1, -corsSSD(i, 1)-corsSSD(i, 2);
10                 1,-1, -corsSSD(i, 1)+corsSSD(i, 2)];
11          A_1 = L_1*P1;
12          A_2 = L_2*P2;
13          [U,S,V] = svd([A_1; A_2]);
14          point3D = V(:, size(V, 2))';
15          point3D = point3D./point3D(4);
16          points3D = [points3D; point3D(1, :)];
17      end
18  end
```

Listing 14: findOutliers.m used in Script of Problem 6

```
1   function [ inlier, outlier ] = findOutliers(points3D, P2, outlierTH, corsSSD)
2   inlier = [];
3   outlier = [];
4   for i = 1:size(corsSSD, 1)
5       if corsSSD(i, :) ~= [0, 0]
6           reproject = P2*(points3D(i, :)');
7           reproject = reproject(1:2,1)./reproject(3,1);
8           if norm(reproject - corsSSD(i, :)') < outlierTH
9               inlier = [inlier; reproject'];
10          else
11              outlier = [outlier; reproject'];
12          end
13      end
14  end
15  end
```

Listing 15: interpolateLine.m used in Script of Problem 6

```
1   function points = interpolateLine( line )
```

```
 2 lineEnd_1 = line(1, 1:2);
 3 lineEnd_2 = line(1, 3:4);
 4 points = [];
 5 deltaY = (lineEnd_2(2) - lineEnd_1(2))/(lineEnd_2(1) - lineEnd_1(1));
 6 deltaX = (lineEnd_2(1) - lineEnd_1(1))/(lineEnd_2(2) - lineEnd_1(2));
 7 for X = lineEnd_1(1):lineEnd_2(1)
 8     Y = round(lineEnd_1(2) + (X-lineEnd_1(1))*deltaY);
 9     points = [points; X, Y];
10 end
11 for Y = lineEnd_1(2):lineEnd_2(2)
12     X = round(lineEnd_1(1) + (Y-lineEnd_1(2))*deltaX);
13     points = [points; X, Y];
14 end
15 points = int32(points);
16 end
```

Listing 16: showMarker1.m used in Script of Problem 6

```
1 function showMarker1(img, pts)
2 imshow(img);
3 hold on;
4 scatter(pts(:, 1)',pts(:, 2)',250,'b','LineWidth',4);
5 hold off;
6 end
```

Listing 17: showMarker3.m used in Script of Problem 6

```
1 function showMarker3(img, corsSSD, inlier, outlier)
2 imshow(img);
3 hold on;
4 scatter(corsSSD(:, 1)',corsSSD(:, 2)',250,'k','LineWidth',3);
5 scatter(inlier(:, 1)',inlier(:, 2)',250,'b','+','LineWidth',3);
6 scatter(outlier(:, 1)',outlier(:, 2)',250,'r','+','LineWidth',3);
7 hold off;
8
9 end
```

Listing 18: showMatching.m used in Script of Problem 6

```
1 function showMatching(I1,I2,corners1,corners2)
2 matchedPoints_1 = [];
3 matchedPoints_2 = [];
4 for i = 1:size(corners1, 1)
5     if corners2(i, :) ~= [0, 0]
6         matchedPoints_1 = [matchedPoints_1; corners1(i, :)];
7         matchedPoints_2 = [matchedPoints_2; corners2(i, :)];
8     end
9 end
```

```
10  showMatchedFeatures(I1,I2,matchedPoints_1,matchedPoints_2,'montage','PlotOptions',{'ro
11
12  end
```

Submitted by Wei-Yuan Wen, Yen-Ting Chen, Shun-Jen Lee on November 2, 2016.