Wei-Yuan Wen, Yen-Ting Chen, Shun-Jen Lee
*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:**

**Problem** 2. Epipolar Geometry [4 pt]

**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. The outputs should look like Figure 1. Describe how well you think this will 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 $\frac{p \cap g}{p \cup g}$. 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.

**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 $\theta$ 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:**

**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:**

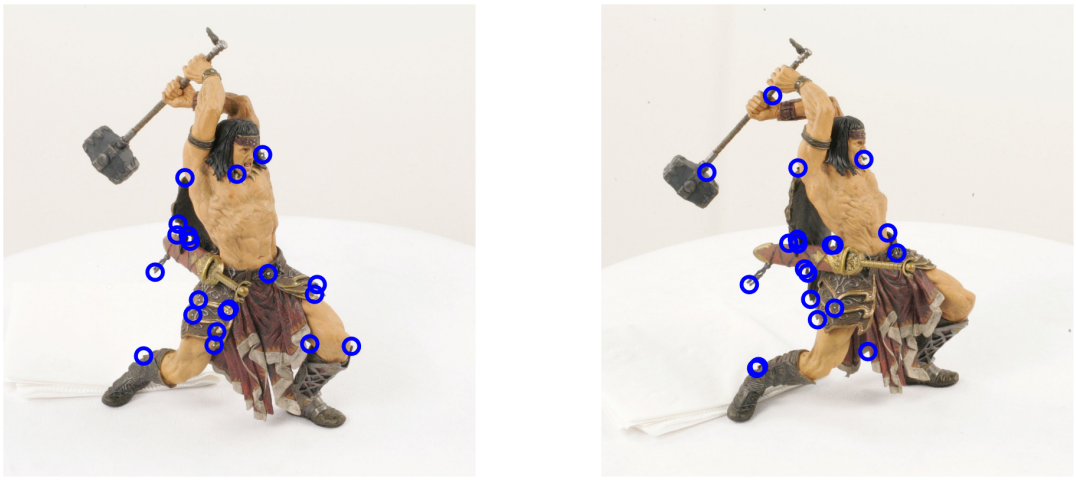

Figure 1: Corner Detection: Warrior

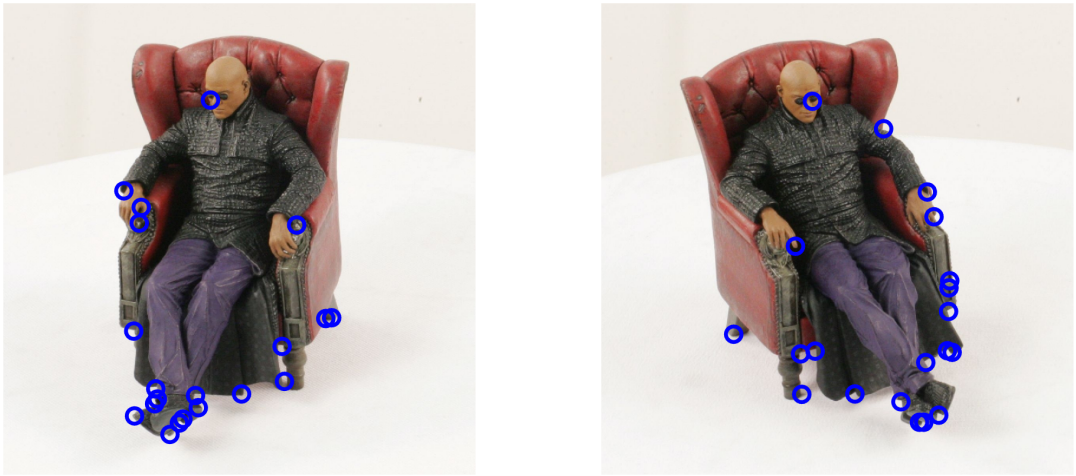

Figure 2: Corner Detection: Matrix

**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:**

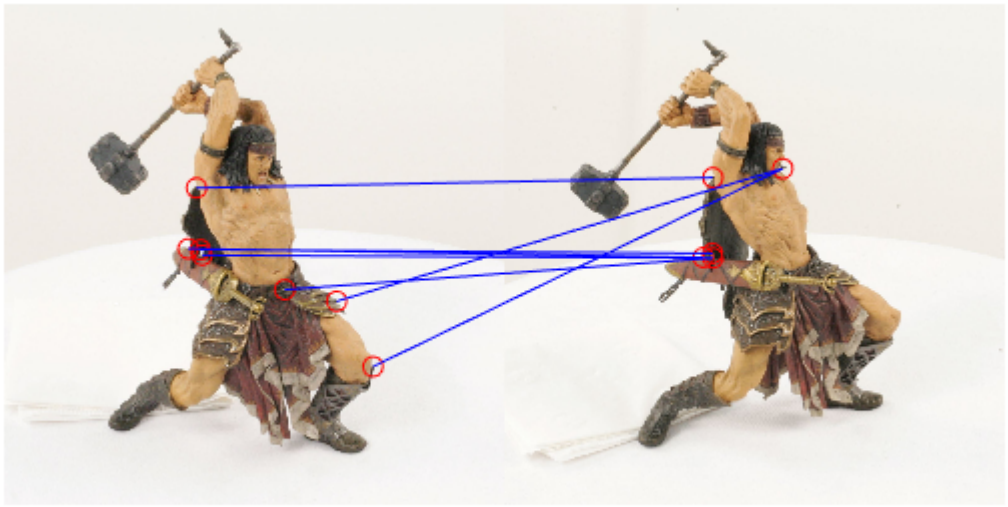

Figure 3: Naive Matching: Warrior

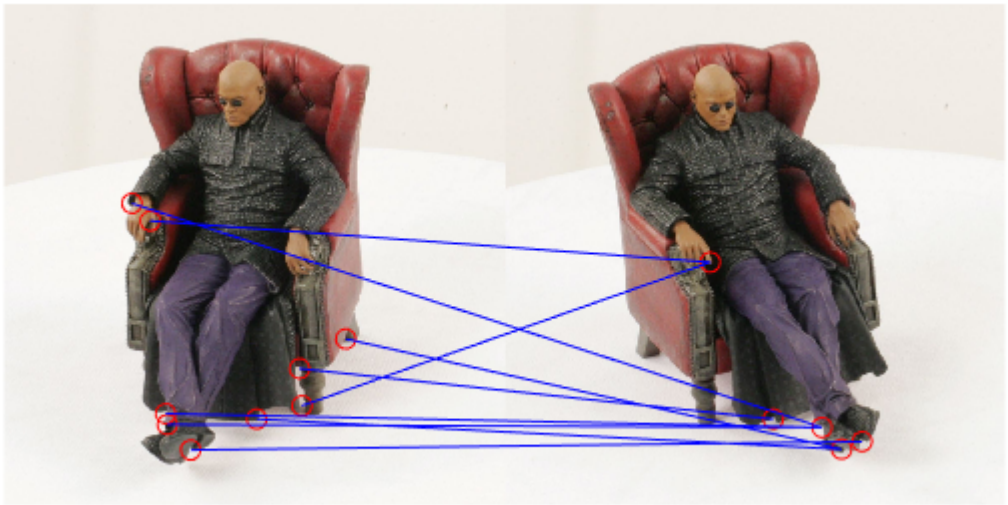

Figure 4: Naive Matching: Matrix

**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:**

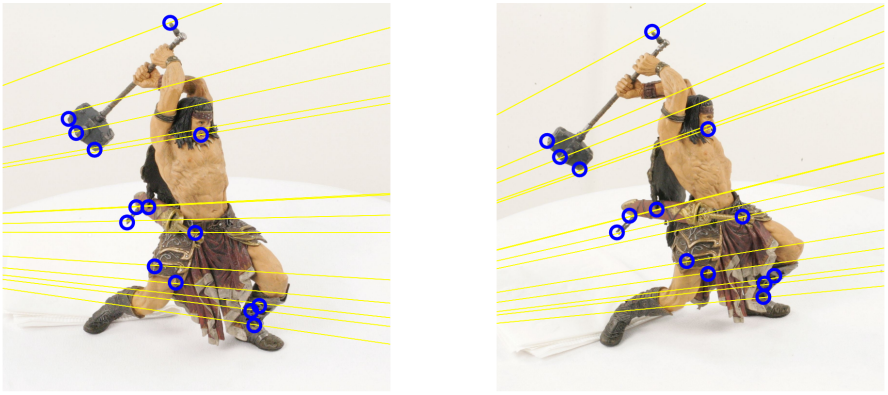

Figure 5: Epipolar Lines: Warrior

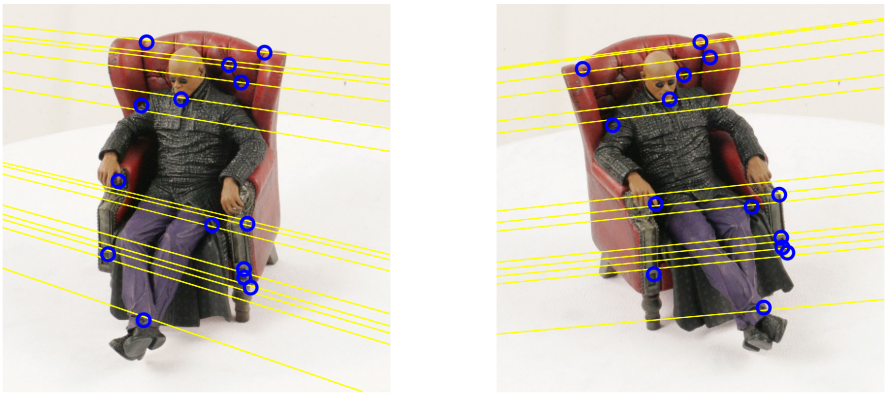

Figure 6: Epipolar Lines: Matrix

**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:**

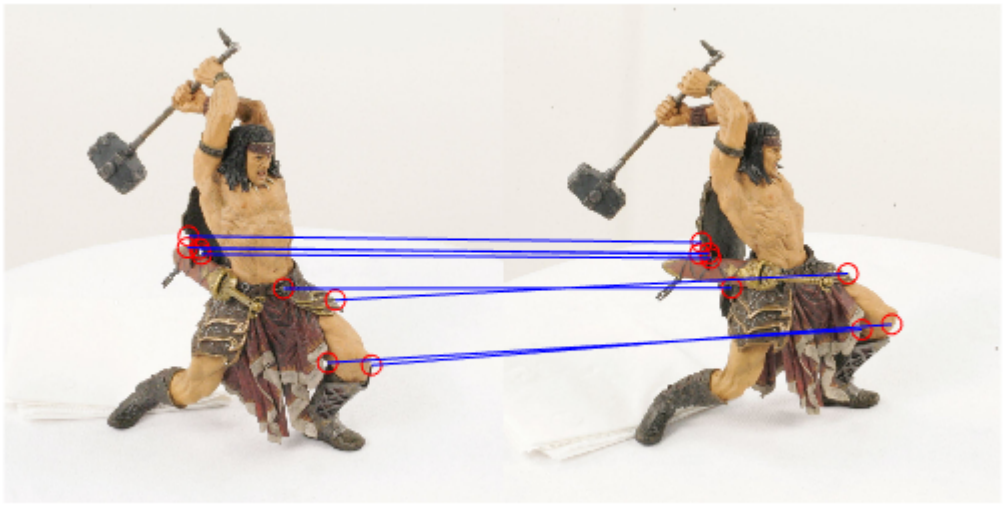

Figure 7: Epipolar Geometry Based Matching: Warrior

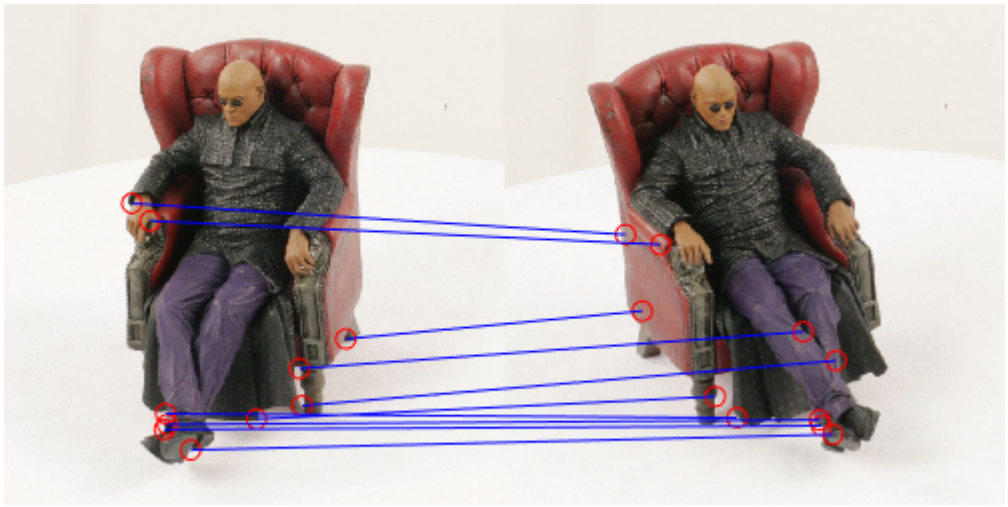

Figure 8: Epipolar Geometry Based Matching: Matrix

**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:**

## Appendix

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