EN1610 Image Understanding

Lab # 4: Corners , Interest Points, Hough Transform

The goal of this fourth lab is to

- Learn how to detect corners, and use them in a tracking application
- Learn how to describe a keypoint, with a region descriptor, explore the most popular SIFT,
- Learn how to use SIFT in an application
- Use the Hough Transform to detect lines and circles

All the images you will need for the lab can be downloaded off the class website, `http://vision.lems.brown.edu/engn161/fall2011/Labs`.

Corner Detection

Problem 1. Implement Corner Detection Algorithm

1. Filter the image $f(x, y)$ with spatial derivatives of a Gaussian $G(x, y, \sigma_1)$, where σ_1 is called the differentiation scale, typically $\sigma_1 = 0.7$ or 1, *i.e.* G_x and G_y are used to estimate the gradient of f:

$$\begin{cases} f_x &= G_x * f \\ f_y &= G_y * f \end{cases}$$

2. Form three spatial maps:

$$\begin{cases} A(x, y) &= (G_x * f)^2 \\ B(x, y) &= (G_x * f)(G_y * f) \\ C(x, y) &= (G_y * f)^2 \end{cases}$$

3. Blur these maps with a Gaussian $G(x, y, \sigma_2)$, where σ_2 is the integration scale which can vary over a range of scales, *e.g.* $\sigma_2 = 2.0$ or 3.0, *etc.*:

$$\begin{cases} \overline{A}(x, y) &= G(x, y, \sigma_2) * A \\ \overline{B}(x, y) &= G(x, y, \sigma_2) * B \\ \overline{C}(x, y) &= G(x, y, \sigma_2) * C \end{cases}$$

4. Compute

$$\begin{aligned} tr(x, y) &= \overline{A}(x, y) + \overline{C}(x, y) \\ det(x, y) &= \overline{AC} - \overline{B}^2 \end{aligned}$$

5. Compute

$$R(x, y) = det(x, y) - \alpha\ tr^2(x, y), \tag{1}$$

where α is a parameter of the system

6. Compute local max of $R(x, y)$ by applying non-maximum suppression over a 3×3 neighborhood of each point.

7. Threshold R_0 is used to prune points with $R(x, y) < R_0$, where $R_0 = 0.01 \times \max_{x,y} R(x, y)$

8. Return all points (x, y) with strength $R(x, y)$.

Typical values to use $\sigma_2 = 2.0$ and $\alpha = 0.04$. For the window size for the Gaussian use $\lceil 6\sigma \rceil$ (Add one if needed to make it odd). You can use a Sobel operator to estimate image gradients.

Display your results on the images below, Figure 1 , again use `imshow` followed by `hold on` , then use the `plot` command.

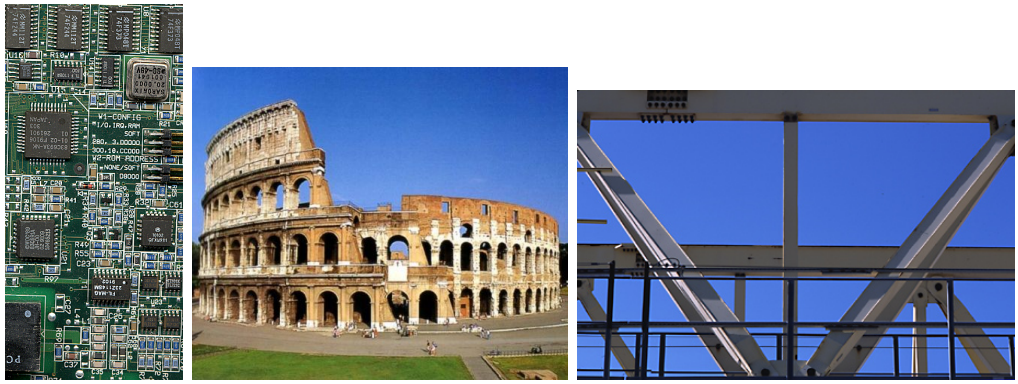


Figure 1: a) Circuit Board b) Coliseum c) Crane

Problem 2. Application of Corners: KLT Tracker In this section we will use the corners you developed in problem 1 to track them across a video sequence. For each corner point (x, y) we will assume that it undergoes a translation from one frame to the next, and we will estimate this displacement, $\overline{v}$, across frames. A corner in the first frame, will be at $(x, y) + \overline{v}$ in the second frame, and we can again estimate $\overline{v}$ to determine its location in the next frame. This iterative procedure continues until we have tracked the corner across all frames of the video sequence. To determine $\overline{v}$ we will be using a very simplified version of the KLT tracker. Pseudo-code of the algorithm you will be implementing, is Algorithm 1.

$$M(p_x, p_y) = \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} \begin{bmatrix} I_x^2(x, y) & I_x(x, y)I_y(x, y) \\ I_x(x, y)I_y(x, y) & I_y^2(x, y) \end{bmatrix} \tag{2}$$

$$\overline{b}(p_x, p_y) = \sum_{x=p_x-w_x}^{p_x+w_x} \sum_{y=p_y-w_y}^{p_y+w_y} \begin{bmatrix} \delta I_t(x, y)I_x(x, y) \\ \delta I_t(x, y)I_y(x, y) \end{bmatrix} \tag{3}$$

Algorithm 1 Simple KLT tracker

1: $I_{f:N}$ Video Sequence of Frames $I_1, I_2, ...I_N$
2: Compute corners, $\overrightarrow{corners}$ (vector of 2D points), on I_1

3: **for** f=1 to N-1 (loop thru frames except last) **do**
4: **for** c=1 to N (loop thru all corners) **do**
5: $(p_x, p_y) \leftarrow \overrightarrow{corners}[c]$
6: $\forall (x, y) \in [p_x - w_x, p_x + w_x]x[p_y - w_y, p_y + w_y]$, $A(x, y) = I_f(x, y)$
7: Compute $M(p_x, p_y)$ on I_f, use Equation 2
8: $\overline{v}^0 \leftarrow [0\ 0]$
9: **for** k=1 to K (or until $\|\overline{\eta}^k\| <$ accuracy threshold) **do**
10: $\forall (x, y) \in [p_x - w_x, p_x + w_x]x[p_y - w_y, p_y + w_y]$, $B^k(x, y) = I_{f+1}(x + \overline{v}_x^{k-1}, y + \overline{v}_y^{k-1})$
11: $\delta I_t^k(x, y) \leftarrow A(x, y) - B^k(x, y)$
12: Compute $\overline{b}^k(p_x, p_y)$, use Equation 3
13: $\overline{\eta}^k = M^{-1}\overline{b}^k$
14: $\overline{v}^k = \overline{v}^{k-1} + \overline{\eta}^k$
15: **end for**
16: **if** $\overline{v}$ converged **then**
17: $\overrightarrow{corners}[c] \leftarrow (p_x, p_y) + \overline{v}^k$
18: **else**
19: $\overrightarrow{corners}[c] \leftarrow \emptyset$
20: **end if**
21: **end for**
22: **end for**

- When considering the number of corners, to track, use a very high threshold, to make sure you are getting strong corners, Try to track 20 to 30 features across the video sequence , again try to track more depending on how long it takes
- Computing 2 and 3 require a window centered around the corner under consideration. The most common window size is 5, again, you will have to play with this parameter to get good results on the video sequences
- **Line 2 in Algorithm 1**, This is the output of problem 1
- **Line 6 and 10 in Algorithm 1**, The window centered around the corner point, will not always be at pixel coordinates , but rather will be at subpixel locations, you will have to interpolate to find these values, use `meshgrid` and `interp2`, when using `interp2` you can use the linear interpolation option, and also set the Extrapval to 0, for values outside the image boundaries
- **Line 9 in Algorithm 1**, Use at most K=15 iterations, and an accuracy threshold of 0.01, the accuracy threshold represents the pixel distance, again you might have to play around with this parameter to get good results

- **Line 16 thru Line 20 Algorithm 1**, We have two cases, that the feature is kept, or the feature is lost. If $\overline{v}$ converges, then we keep the corner, and replace our current position of the corner with our new estimate, and if it doesn't we want to delete this corner position, and not track it further, In practice, you can define a new empty array, to hold all new corners, and then reset the old vector of points to the new set, Also stop tracking the corner if the feature falls outside the image boundaries

There are three different sequences that I have given the class to try. See Figure 2. The rubix cube is more of a way to test, your algorithm, as you should look at the tracks, on the cube to see whether you are implementing it correctly. As for how your results should look, please refer to the examples, on the course webpage.

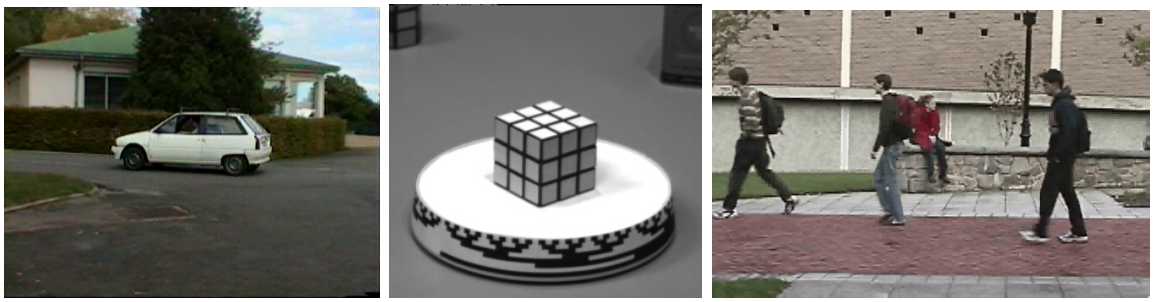


Figure 2: a) Car sequence b) Rubix Cube c) Pedestrian Walking

Challenge - Improving the Tracker If you look at the algorithm I proposed, once features are lost, I do not look for new features. Implement a new scheme where by if a certain number of features are not present you look for new features at that frame. Another improvement is to use a Gaussian pyramid so you can deal with large translation movement.

Challenge - Structure from Motion: Factorization This is a very challenging problem, but one can recover the 3D scene from the tracks you have. The technique is called factorization, and the most famous is the **Tomasi-Kanade factorization**, `http://en.wikipedia.org/wiki/Tomasi-Kanade_factorization`. If you do attempt this, try it on the rubix cube case.

Interest Points/Region Descriptors

Problem 3. Experiment with Region Descriptors In order to find correspondences between interest points, we need to design region descriptors. In this, question we will implement some very simple descriptors for the corners (interest points) in problem 1.

1. Extract the corners (keypoints) using your code from Problem 1
2. We will experiment with two different region descriptors, one is just the pixels around the corner, and the other is a histogram. Write two functions that will compute this

(a) `function D = descriptors_pixels(img,corners,window)`

(b) `function D = descriptors_histogram(img,corners,window)`

3. Now, write a function which takes two sets of region descriptors D_1 and D_2 and tries to find for each descriptor in D_1 the nearest neighbor in D_2, The function will return the index of the nearest neighbor and its distance, We will try different norms depending on the descriptor type , pixel based or histogram based

   (a) `function [Idx,Dist] = findnn_pixels_euclidean(D_1,D_2)`

   (b) `function [Idx,Dist] = findnn_pixels_correlation(D_1,D_2)`

   (c) `function [Idx,Dist] = findnn_histogram_chi_square(D_1,D_2)`

   The first two functions are defined for the pixel descriptor, and the last one for the histogram. For the first function just use the standard Euclidean distance, and for the second function use normalized cross correlation, in matlab `normxcorr2`. Finally, use the χ^2 distance from Lab 4.

4. **Matching** Now we have all the components for a small matching application. We will use two datasets, that exhibit image-plane rotations 3a, and viewpoint changes, see Figure 3b for which we try to find point correspondences. Use the provided function `match_plots` to visualize the N best matches. Visualize your matches for the three cases, `pixels_euclidean`,`pixels_correlation`,`histogram_chi_square`

(a) Plane Rotations (b) ViewPoint Changes

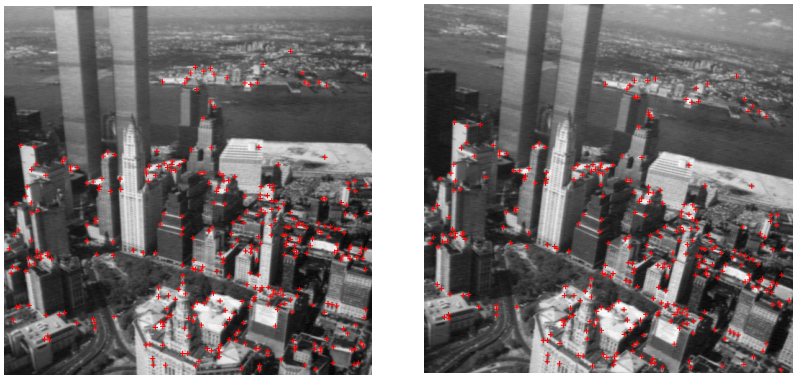
Figure 3: Images to Test Descriptors

5. **How good are your matches?** To determine this we need the relationship of how points in one image are transformed into another. This relationship is called a **Homography**, and is encoded in a matrix. In the case, of the New York City skyline images, it relates every point in one image by a rotation of the plane. You will visually inspect your matches, against the true matches by using this Homography.
   More formally, in 2D a homography is a defined by a 3x3 matrix H, which relates the points p in one image to corresponding points p' in the second image. The homography for the two images sets is given to you.

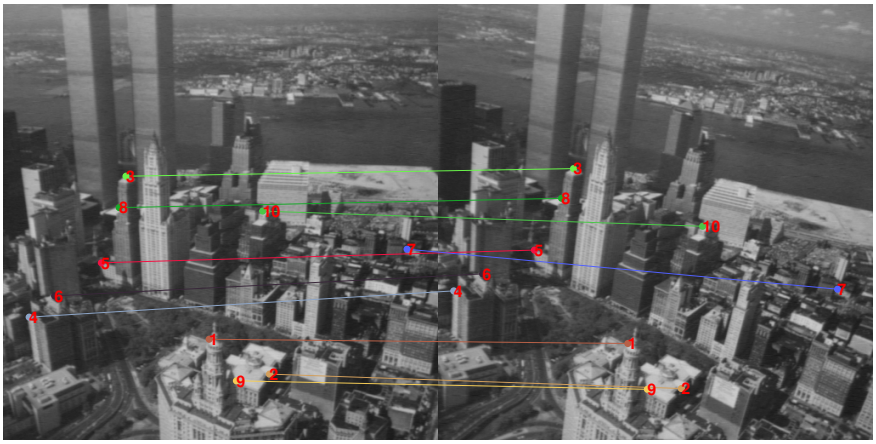
$$p' = Hp \tag{4}$$

This transformation is defined in terms of homogenous coordinates so you will have to convert all your points p, $(x, y) \rightarrow (x, y, 1)$ and the apply the transformation. Lastly, the results of this transformation, will be (wx, wy, w) so divide all points by the scaling factor w and use the first two elements as your points. Overlay your results over your matches,

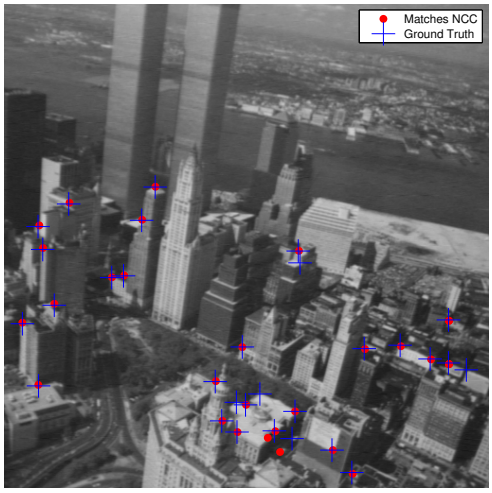
6. An outline of the results for each step can be seen in Figure 4,inital corners 4a,top ten results of matching 4b, overaly with ground truth 4c



(a) Corners Image 1 and Image 2



(b) Top Ten Matches



(c) Thirty Matches with Ground Truth

Figure 4: Outline of Matching System

7. Finally discuss your results, which descriptor performed best? for what image type? Where did the descriptors fail? Would you say these descriptors are invariant to rotation? viewpoint changes?

Hough Transform - Lines/Circles

Problem 4. Implement Hough Transform for Lines In this problem, you will implement the Hough Transform to detect lines, see Figure 5. We will use the normal form of a

line, Equation 5. The general outline of the algorithm is below 2.

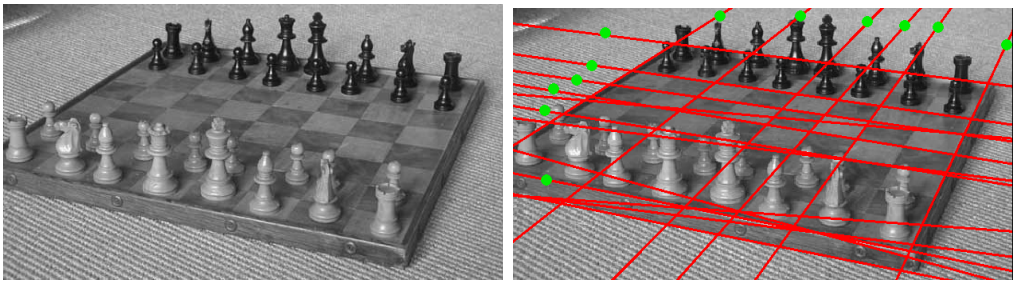


Figure 5: a) Checkerboard Image b) Output of Line Hough Transform

$$\rho = x\cos(\theta) + y\sin(\theta) \tag{5}$$

Algorithm 2 Hough Transform for Lines

1: I, Binary Edge Map size [M x N]
2: $\rho_d[r] \in [0, \sqrt{N^2 + M^2}], \theta_d[t] \in [0, pi]$, Arrays containing the discretized intervals of the parameter space
3: A be the accumulator, matrix initialized to all zeros, size [R x T] where R is the length of the array of $\rho_d[r]$ and T is the length of the array of $\theta_d[t]$

4: **for** $i = 1 \rightarrow M$ **do**
5: **for** $j = 1 \rightarrow N$ **do**
6: **if** $I(i, j) == 1$ **then**
7: **for** $t = 1 \rightarrow T$ **do**
8: $\rho = i\cos(\theta_d[t]) + j\sin(\theta_d[t])$
9: Find index r so that $\rho_d[r]$ is closest to ρ
10: $A[r, t] \leftarrow A[r, t] + 1$
11: **end for**
12: **end if**
13: **end for**
14: **end for**
15: Find all local maxima (r_k, t_k) such that $A[r_k, t_k] > \tau$, where τ is user defined threshold
16: The output is a set of lines described by $(\rho_d[r_k], \theta_d[t_k])$

- **Line 1 in Algorithm 2**, This is the output of a gradient based edge detector. Use matlab's `edge` function, and use the Sobel operator
- **Line 2 in Algorithm 2** When you define these arrays you are discretizing the parameter space of ρ and θ using sampling steps of $\delta\rho$ and $\delta\theta$. Choose appropriate step sizes that yield acceptable and manageable resolution of the parameter space
- **Line 9 in Algorithm 2** The value compute for ρ will not exactly be in your parameter space, so you have to choose either to round, ceil, or floor the value

- **Line 15 in Algorithm 2** When defining your threshold , think about how many edges would have to vote for a line

When writing your report up

1. Display your accumulator space A as an image, this is also called a sinogram
2. Plot all lines found on top of the images,Figure 6, again use `imshow` followed by `hold on` , then use the `plot` command.

Figure 6: Images to try for line detection

3. What does an edge point (x, y) in the image correspond to in the (ρ, θ) parameter space?
4. Comment on the the effect of discretization of the parameter space, and the threshold you used

Problem 5. Implement Hough Transform for Circles The Hough Transform is a very general approach that we can use to detect any shape that has a parametric form! We will use the same approach we used to find lines to detect circles, see Figure 7. A circle with center radius r and center (a, b) can be described with the parametric equations 6. If an image contains many points (x, y), some of which fall on perimeters of circles, then the job of the search program is to find parameter triplets (a, b, r) to describe each circle.

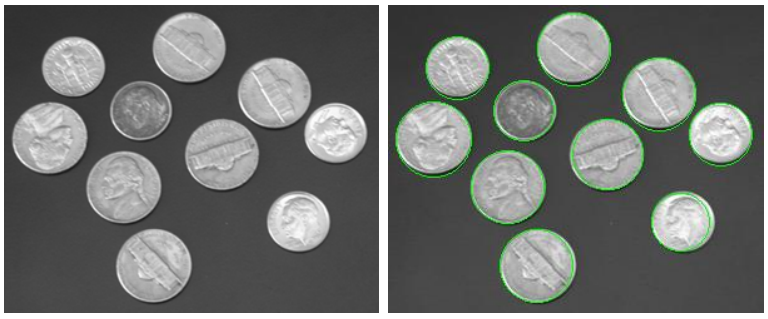


Figure 7: a) Coin Image b) Output of Circular Hough Transform

$$
\begin{aligned}
x &= a + r\cos(\theta) \\
y &= b + r\sin(\theta)
\end{aligned} \tag{6}
$$

We will use the same approach as in the Algorithm 2 for lines.

- **Line 2 in Algorithm 2** Here you will have three arrays that define the discretization of $a_d, b_d,$ andr_d
- **Line 3 in Algorithm 2** Your accumulator with be a 3D matrix of all zeros, again, the size of it will be [length(a_d) x length(b_d) x length(r_d)], the 3D case will be more expensive in terms of both time and memory, so please be judicious in your discretization of the parameter space
- **Line 7 in Algorithm 2** To evaluate Equation 6 you will have two loops instead of one, one loop will be over r_d and the other will be over a discretization of θ where θ ranges from 0 to 2π
- **Line 9 in Algorithm 2** After evaluating Equation 6 you will have two parameters a, b and you will again have to the find closest bin in the 3D accumulator $A[i, j, k]$, keep in mind you are looping over r_d so you already know the index of one parameter
- **Line 15 in Algorithm 2** When defining your threshold , think about how many edges would have to vote for a circle, also keep in mind you have to look around 3D dimensions when finding the local maxima

When writing up your report

1. Plot all circles found on top of the image,Figure 8, again use `imshow` followed by `hold on` , then use the `plot` command. You might have to write a routine to plot a circle

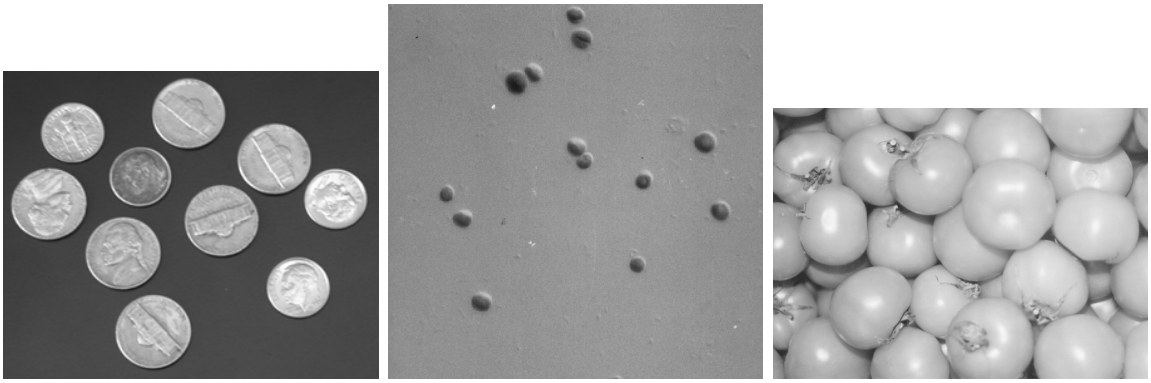


Figure 8: Images to try for circle detection

2. What does an edge point (x, y) in the image correspond to in the (a, b, r) parameter space?

3. Comment on the the effect of discretization of the parameter space, and the threshold you used

Challenge - Accelerated Hough Transform One way of reducing the computation required to perform the Hough transform is to make use of gradient information which is often available as output from an edge detector. Explain in detail how would you use the gradient to accelerate the Hough Transform. **You are free to implement this, but you do not have to**