Grayson Adkins
Applied Computer Vision
Instructor: Dr. Xiaobai Liu
27 April 2016

Homework 5: Object Tracking

I. Overview

The goal of this project is to perform object tracking via feature matching. The task is accomplished using a simplified version of the Kanade-Lucas-Tomasi (KLT) feature tracker. The algorithm, as implmented, has the following six steps:

1. Crop images of objects
2. Detect interest points
3. Extract feature descriptors for every interest point and find correspondences between two cropped images
4. Estimate translation displacement
5. Calculate average traking error
6. Generate video sequence demonstrating tracking

The object being tracked was a coffee mug sitting on a desk which appeared in two separate video sequences. In particular, the focus of this project was on steps 4-5 where two different methods for calculating translation displacement were explored, simple least squares and random sample consensus (RANSAC). I will discuss these in great detail in the following sections.

II. Method

A. Crop images of objects

The object was cropped using a simple bounding box approach represented as a vector [x_min, y_min, width, height]. The object was cropped in one image, then a translation was estimated (Steps 2-4), and a newly calculated bounding box was used to crop the object in the second image. Two different cropping methods were tested. In the first method, the first image (t=0) in the video sequence was always used as a starting point, and the translation was calcuated from t=0 to t=1,2,...,n. In the second method, the previous image at t=t-1 was used as the starting point to find the new bounding box at t. The result of the latter approach was that the estimated new position of the object drifted due to accumulating translation error, resulting in small error beingd compounded in each subsequent crop. This can be seen by the off-centered previously-cropped object in Figure 1. However, in the first method, which always used the first image as reference, the translation estimation was more difficult since the translation distance

was greater as the object moved farther from its original position. As *t* approached *n*, the estimations were very inaccurate. The conclusion drawn from this experiment is that it is best to use the second method and just to try to minimize the error during each iteration (see RANSAC). RANSAC also has a better chance to recover the object since there is randomization of sampling.

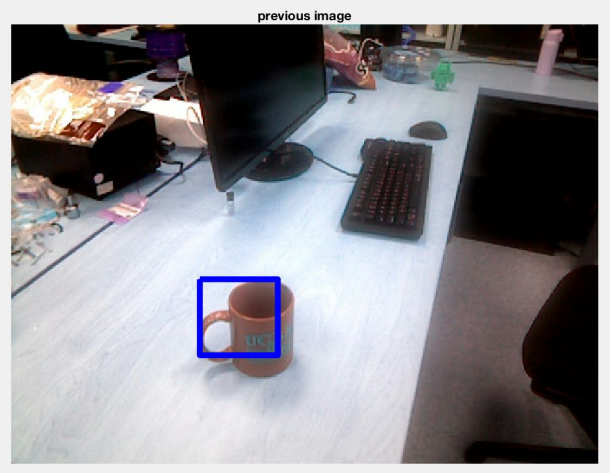

Figure 1: Previous cropped image with misalignment

B. Detect interest points

Interest points were detected using the Matlab Computer Vision Toolbox functions: *detectHarrisFeatures* and *showMatchedFeatures*. This function implements the Harris corner detector that analyzes image gradients in the x and y directions. My own implementation of the Harris corner detector, *get_interest_points.m,* was also used but did not generate points that could reliably be tracked, so the built-in function was used instead for the remainder of the experiments. An example of the matched feature visualization can be seen in Figure 2.

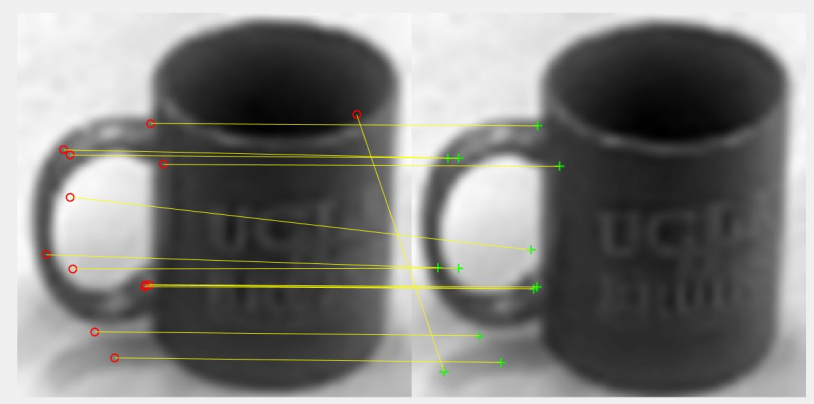

Figure 2. Matching points with some error

C. Extract feature descriptors for every interest point and find correspondences between two cropped images

Feature descriptors were calculated using the Matlab Computer Vision Toolbox function: *extractFeatures*.

D. Estimate translation displacement

The translation was solved using two different methods: simple least squares and RANSAC. Both methods attempt to estimate displacement by fitting a line to the matched sample pairs. Least squares is simple to use and implement and is guided by the following principles. When considering displacement between a single at time *t*=A and t=B, $(x_i\ y_i)^A$ and $(x_i\ y_i)^B$, we can isolate the displacement into two parts: change in x-direction and change in y-direction. We can write this relationship in matrix form as

$$\begin{bmatrix} x_i^B \\ y_i^B \end{bmatrix} = \begin{bmatrix} x_i^A \\ y_i^A \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

where t_x and t_y are the translation distances in x and y-directions, respectively. In general, least squares attempts to minimize the following equation:

$$\min E = \sum_{i=1}^{n} (y_i - mx_i - b)^2$$

where *x* and *y* represent a single data point (x,y) and the slope *m* and intercept *b* describe it's verticle distance from a line that fits all data points. We can solve this equation for all points (*x,y*) analytically by formulating the problem as a linear combination of matrices of the form Ax = b. This equation can be written in matrix so that all matched pairs can be included:

$$E = \sum_{i=1}^{n} \left(\begin{bmatrix} x_i & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix} - y_i \right)^2 = \left\| \begin{bmatrix} x_1 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix} - \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} \right\|^2 = \left\| \mathbf{Ap} - \mathbf{y} \right\|^2$$

Then we can simply solve for **p** directly as follows:

$$\mathbf{p} = \left(\mathbf{A}^T \mathbf{A} \right)^{-1} \mathbf{A}^T \mathbf{y}$$

Thus, our final equation Ax = b is:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ \vdots & \vdots \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} t_x \\ t_y \end{bmatrix} = \begin{bmatrix} x_1^B - x_1^A \\ y_1^B - y_1^A \\ \vdots \\ x_n^B - x_n^A \\ y_n^B - y_n^A \end{bmatrix}$$

We can solve for the translation $[t_x\ \ t_y]$ just as we did for **p**.

Although the least squares method is widely used, it suffers from some limitations; namely, it is very vulnerable to outliers. RANSAC was developed to account for this discrepancy. In the RANSAC equation, a subset of points are chosen and a model is fitted to that subset, rejecting points outside the subset. This is done iteratively by selecting pairs of points at random and a line is fit to them. An error is calculated, and, given some pre-defined threshold, the points are defined as inlier or outlier. This process continues (often decaying logarithmically as in my implementation -- see *ransac.m*) until the best model is chosen based on which line produced the most inliers. To calculate the displacement between consecutive image frames, a modified least squares operation is performed on the chosen subset, and then all other points are compared to the subset's translation.

There are several free parameters in the RANSAC algorithm including:

- Number of iterations, *N*
- Number of points *s* in the subset chosen at random
- Inlier/outlier threshold *t* measured in pixels
- Consensus set size *d*

I chose to implement and adaptive approach for determining *N* instead of fixed, given by the following equation and chart:

$$N = \log(1-p) / \log\left(1-(1-e)^s\right)$$

	proportion of outliers e						
s	5%	10%	20%	25%	30%	40%	50%
2	2	3	5	6	7	11	17
3	3	4	7	9	11	19	35
4	3	5	9	13	17	34	72
5	4	6	12	17	26	57	146
6	4	7	16	24	37	97	293
7	4	8	20	33	54	163	588
8	5	9	26	44	78	272	1177

The Matlab code for the RANSAC implementation is show below.

```
function x = ransac(matchedPts1, matchedPts2)

% Repeat N times:
%   Draw s points uniformly at random
%   Fit line to these s points
%   Find inliers to this line among the remaining points
%       (i.e., points whose distance from the line is less than t)
%   If there are d or more inliers, accept the line and refit using all inliers

if (length(matchedPts1)~=length(matchedPts2))
    error('Lists of matching points must be the same size');
end

p = 0.99;
e = 0.05;
```

```
s = ceil(length(matchedPts1)/2);
N = Inf;
t = 3;
% t = sqrt(3.84*std(std([matchedPts1,matchedPts2])));
x = zeros(size(matchedPts1,1),2);
N_inliers = zeros(size(matchedPts1,1),1);
i = 1;

while N>i
   randPt = randi([1 length(matchedPts1)],s,1);
   displacementx = matchedPts1(randPt,1)-matchedPts2(randPt,1);
   displacementy = matchedPts1(randPt,2)-matchedPts2(randPt,2);
   b=[displacementx; displacementy];
   A1 = repmat([1 0],s,1);
   A2 = repmat([0 1],s,1);
   A = [reshape(A1',[],1),reshape(A2',[],1)];
   x(i,:) = A \ b;
   remainder = setdiff(1:length(matchedPts1), randPt);
   translation1 = matchedPts1(remainder,:) - repmat(x(i,:),length(remainder),1);
   inliers = find(pdist2(translation1,matchedPts2(remainder,:)) < t);
   N_inliers(i) = length(inliers);

   % Adaptively determine number of samples
   e = 1 - length(inliers)/size(matchedPts1,1);
   N = log(1-p)/log(1-(1-e).^s);
   i = i + 1;
end

[~,idx] = max(N_inliers);
x = x(idx);
```

E. Calculate average traking error

The tracking error is the difference between the actual location of the object as defined by the ground truth bounding box and the estimate new location. Tracking error can be calculated as simply the pairwise distance between x_min and y_min; thus a the convenient matlab function *pdist2* was used.

F. Generate video sequence demonstrating tracking

The ground truth and estimated bounding boxes were overlaid on the images and compiled into video sequences *seq1_results.avi* and *seq2_results.avi*. I have uploaded my results to YouTube. They are available at the following links:

Simple least squares method at: https://youtu.be/ZmroAzcQmP4
RANSAC method at: https://youtu.be/3WdYCjesobc

III. Results

The tracking algorithm performed fairly well using both methods of calculating translation. The implementation of RANSAC uses a pseudo-random number generator to select a subset of points during each iteration and a non-fixed, logarithmically decreasing value of *N*; together, these factors caused the results to be less consistent than simple least squares which was deterministic. Thus, in some situations, RANSAC outperformed simple least squares as expected, but in some cases it did not. A comparison of the two methods and their average error rates can be seen in Tables 1 and 2. As shown, for both approaches, Sequence 2 was more difficult t track than Sequence 1. One possible reason for this is that because the camera is closer to the object in Sequence 2, it occupies a larger field of view; the harris corner detector, consequentially had less success finding matching points, especially as the cropped object error increased and the object drifted farther out of view.

Table 1: Simple Least Squares Error

Simple Least Squares	Average Error	Duration (s)
Sequence 1	14.71	65.997
Sequence 2	29.45	76.676

Table 2: RANSAC Error

RANSAC	Threshold *t*	Average Error Range	Average Duration (s)
Sequence 1	3	16.41 - 23.53	50.022
	5	13.08 - 15.23	37.816
	10	17.24 - 22.92	40.929
Sequence 2	3	93.72 -107.37	70.290
	5	88.12 -142.08	62.891
	10	77.43 - 82.31	45.263

From observations of the video, the RANSAC approach was able to re-center the object more effectively then simple least squares. The reason for this is probably because of the pseudo-random component of the algorithm that only selects a few random subsets, and thus, can “jump” out of the error drift. In some cases, RANSAC also appeared to hold a tighter lock on the object. This is likely due to the inaccurate matches acting as outliers and RANSAC’s ability to diminish the influence these outliers.

RANSAC also appear to have much larger variability in the amount of time it took to complete the algorithm. Although, simple least squares took longer an average, RANSAC was often unpredictable in its duration. This was due to *N* being non-fixed and is a key disadvantage of RANSAC, that is, it has no upper bound on computation time. The longer the algorithm runs, the better the results should be, but it are not guaranteed to beat simple least squares if it is not given sufficient iterations.