Problem 1.1
Pseudo code:
Given an image x and threshold Tau:

1. Compute vertical and horizontal gradients for image x
2. Compute the “H” matrix for each pixel in the image, using previously computed vertical and horizontal gradients. The H window is shown in the slides.
3. Compute the eigenvalues for each H window. The method for doing this is also shown in the slides. Each pixel will produce 2 eigenvalues. We we will call these lambda_1 and lambda_2.
4. Create a new image using the smaller lambda (lambda_1). Each pixel in image 1 corresponds to the lambda_1 pixel computed in the previous step.
5. Threshold the pixels of the new image using input parameter Tau. This ensures only the “stronger” corners will be shown.
6. Perform non-maxima suppression over a 5x5 window. This ensures that there is no more than 1 key point in any 5x5 window.
7. For viewing only: bring all key points to maximum brightness. Allows user to see easily.

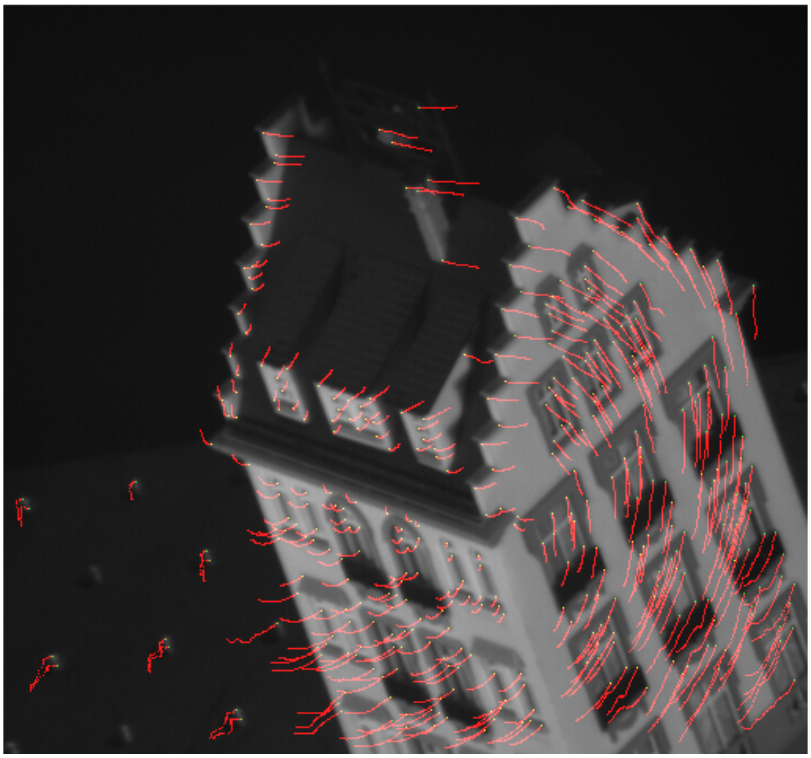
Note: If this pseudo code is not detailed enough, please see KLT_tracker.m, as I have fully implemented a working corner detector and you can see more details there.

Detected corner images:
Points are shown in **green**
Note: lower tau (threshold) = more detections
(In order: Tau = 0.00035, Tau = 0.0001, Tau = 0.00001)

Problem 1.2
Display tracked key points at first and second frames.
For this, it is difficult to see the difference just from frames 1-2. Therefore, I also included frames 1-10 for better visualization.
For all of these frames, I chose Tau = 0.0001. Any key points which left the frame were ignored.

[left image: frames 1-10, right image: frames 1-2]

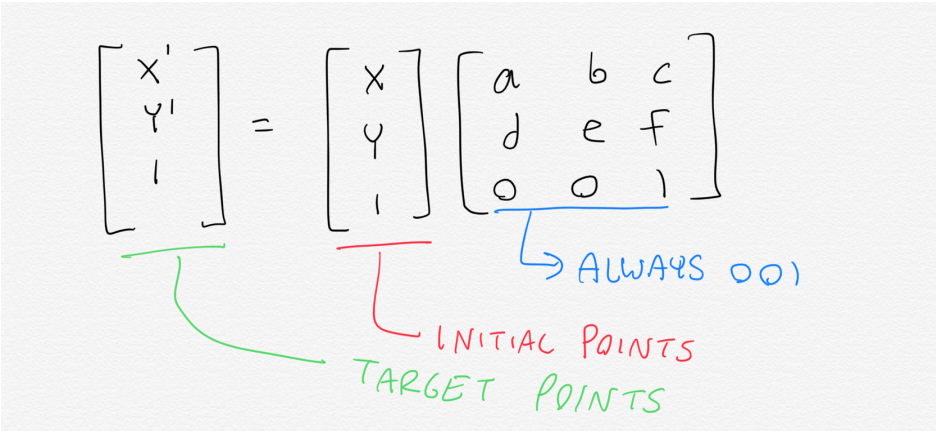
Now I show all 50 frames. I chose Tau = 0.0001 again. There is a lot more than just 20 points in this frame. There is some distortion due to pixel rounding. I am new to matlab and am unsure how to perform write-based interpolation.



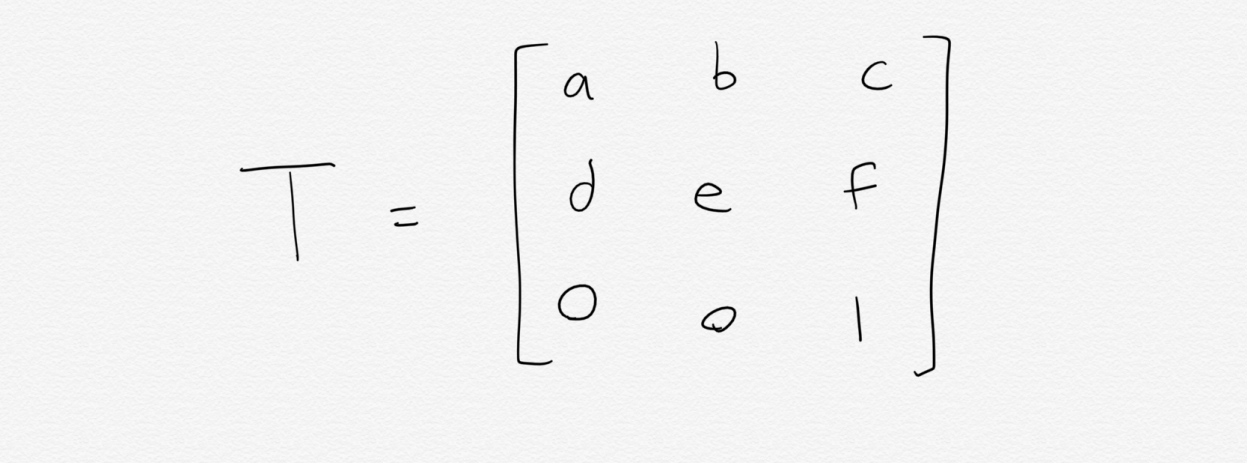
Below we should the points that were lost during tracking. Most are near the bottom where the hotel moves out of frame. These are their initial positions. Look (very) closely, they are marked in red.
Tau of 0.0001 was chosen again here.

Problem 2
For this problem, we choose to implement **gradient descent** to attempt to solve the matrix T. Firstly, we make the following observations:

Given target points x' and y', with initial points x and y, we know that we can represent them as follows:



Therefore, the matrix we are solving for, which we will call matrix T, can be represented as follows:

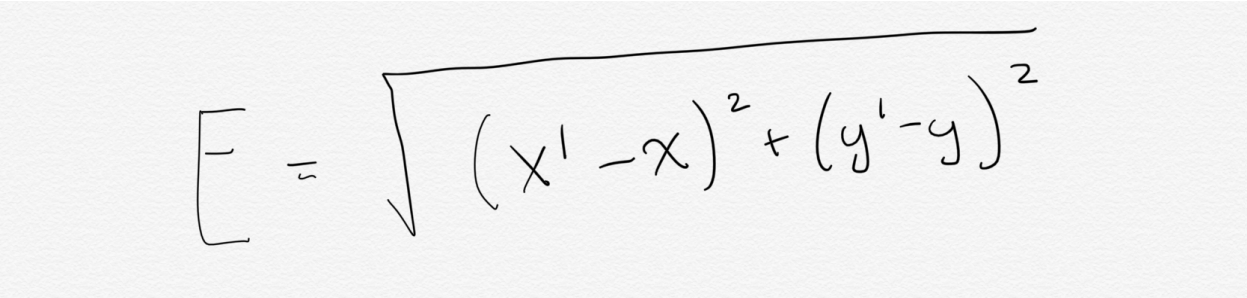


Where our goal is to compute x', y' such that:

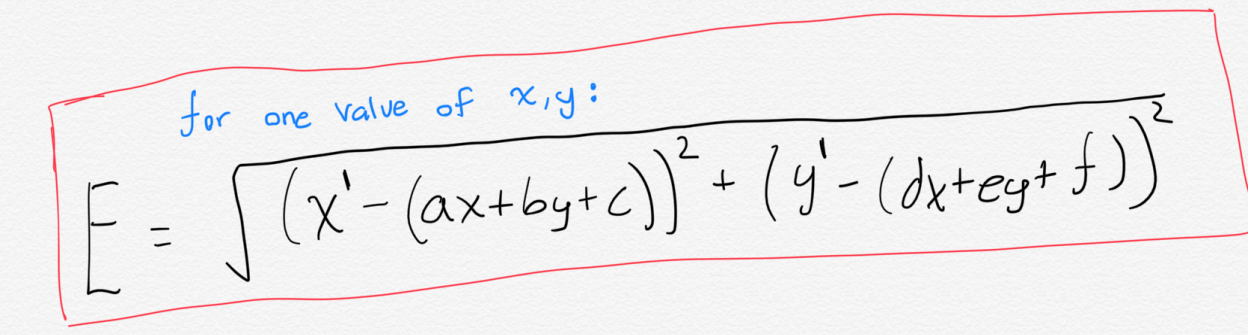
$$x' = ax + by + c$$
$$y' = dx + ey + f$$

We note that this leaves us 6 unknowns to solve for: a, b, c, d, e and f. As is fundamental in gradient descent, we must choose an error function to minimize. For this problem, we wish the distance for every point p in the initial image and every corresponding point p' in the target image to be minimized. That is, we want to minimize the distance between every corresponding p and p'. Of course, we do not know this correspondence, so we can only estimate with closest pairs.

Therefore, we choose our error function to be:



Substituting x' and y', our final error function is



As stated earlier, we wish to compute the 6 unknowns a, b, c, d, e and f. Therefore, we take the partial derivative of each with respect to the error function and get:
(note: these are not in alphabetical order)

$$\frac{\partial E}{\partial b} = \frac{y\,(ax+by+c-x')}{\sqrt{(ax+by+c-x')^2 + (dx+f-y'+ey)^2}}$$

$$\frac{\partial E}{\partial c} = \frac{ax+by+c-x'}{\sqrt{(ax+by+c-x')^2 + (dx+ey+f-y')^2}}$$

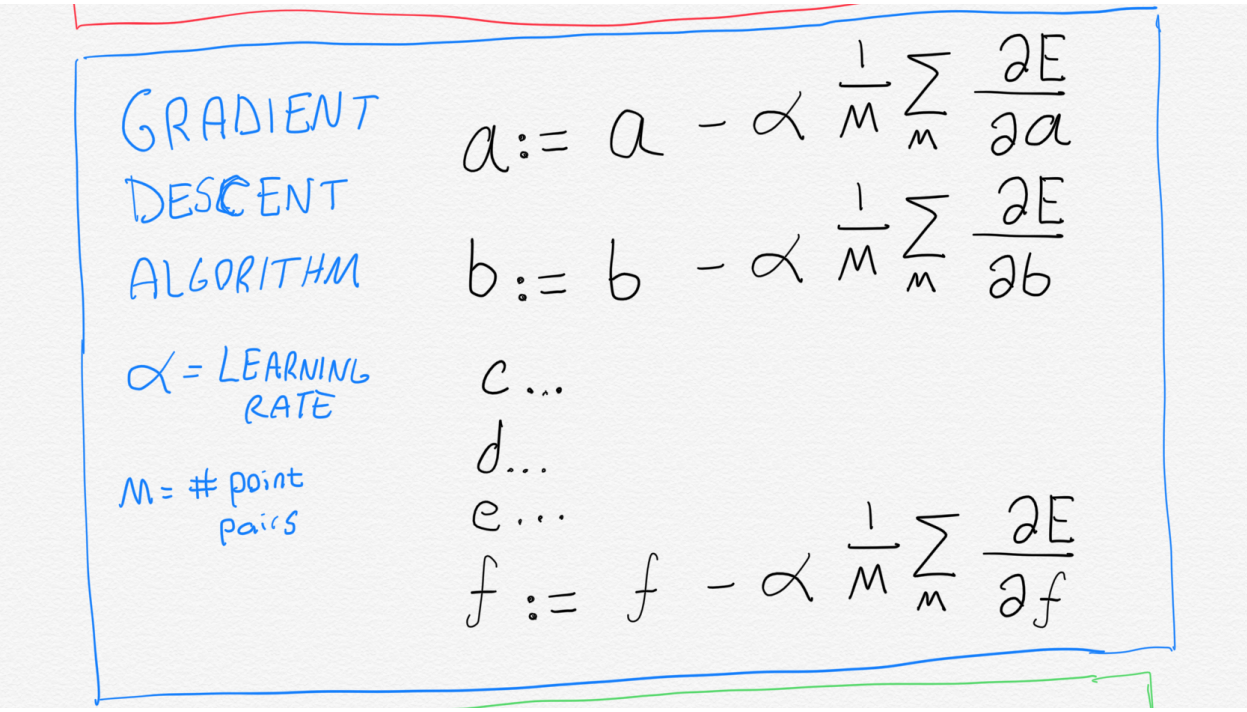
$$\frac{\partial E}{\partial f} = \frac{dx+f-y'+ey}{\sqrt{(ax+by+c-x')^2 + (dx+f-y'+ey)^2}}$$

$$\frac{\partial E}{\partial d} = \frac{x\,(f+dx-y'+ey)}{\sqrt{(ax+by+c-x')^2 + (f+dx-y'+ey)^2}}$$

$$\frac{\partial E}{\partial e} = \frac{y\,(dx+f+ey-y')}{\sqrt{(ax+by+c-x')^2 + (dx+f+ey-y')^2}}$$

$$\frac{\partial E}{\partial a} = \frac{x\,(ax+by+c-x')}{\sqrt{(ax+by+c-x')^2 + (dx+ey+f-y')^2}}$$

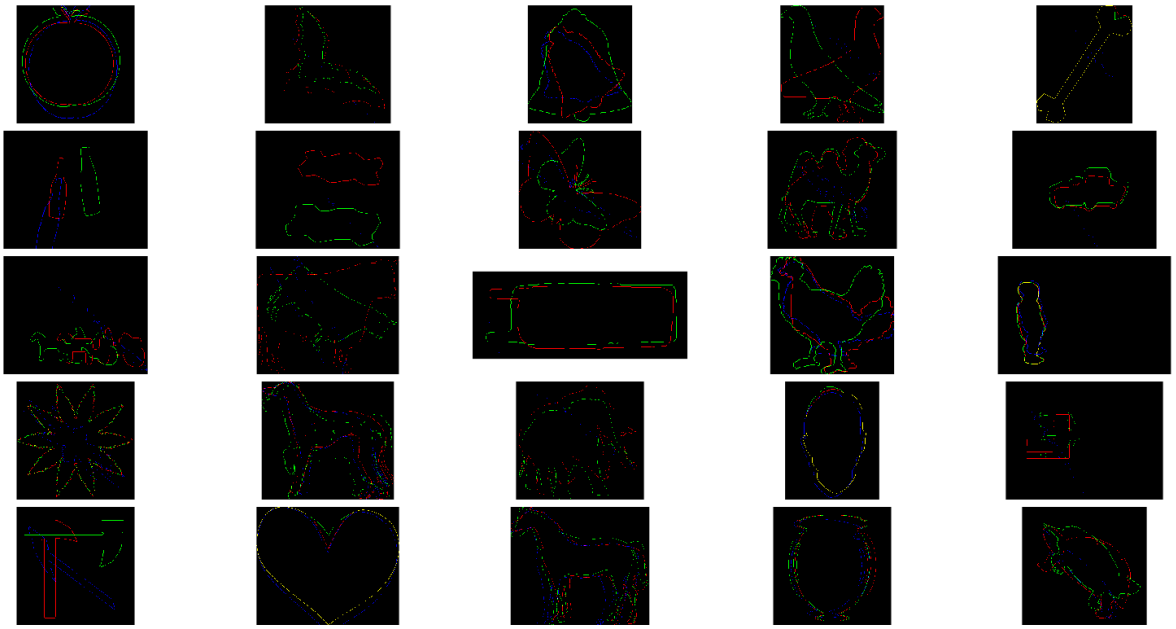
Therefore, we arrive at our iterative gradient descent algorithm as follows:



You might have noticed that we did not provide a summation in initial error function. Here, we take care of this, where we sum over the number of point pairs to compute each derivative. As shown in the illustration, we have a learning rate alpha and a certain number of point pairs M.

Initial Results

We initially try no to use the standard initialization matrix just to see the results. Also, we start with an alpha of 0.001 and iteration count of 200. Unfortunately, they are fairly poor, we an average error of 44.



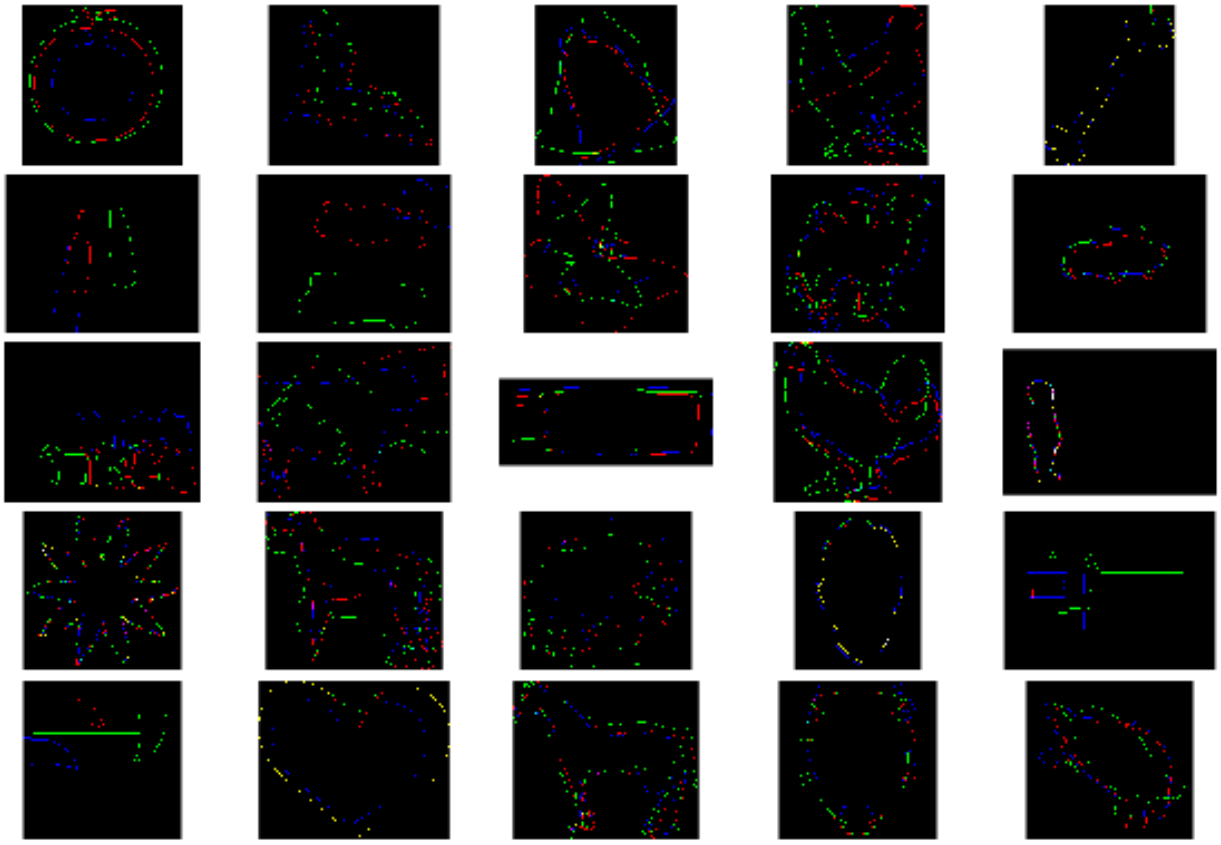
I can think of several reasons for these initially poor results, among them are:

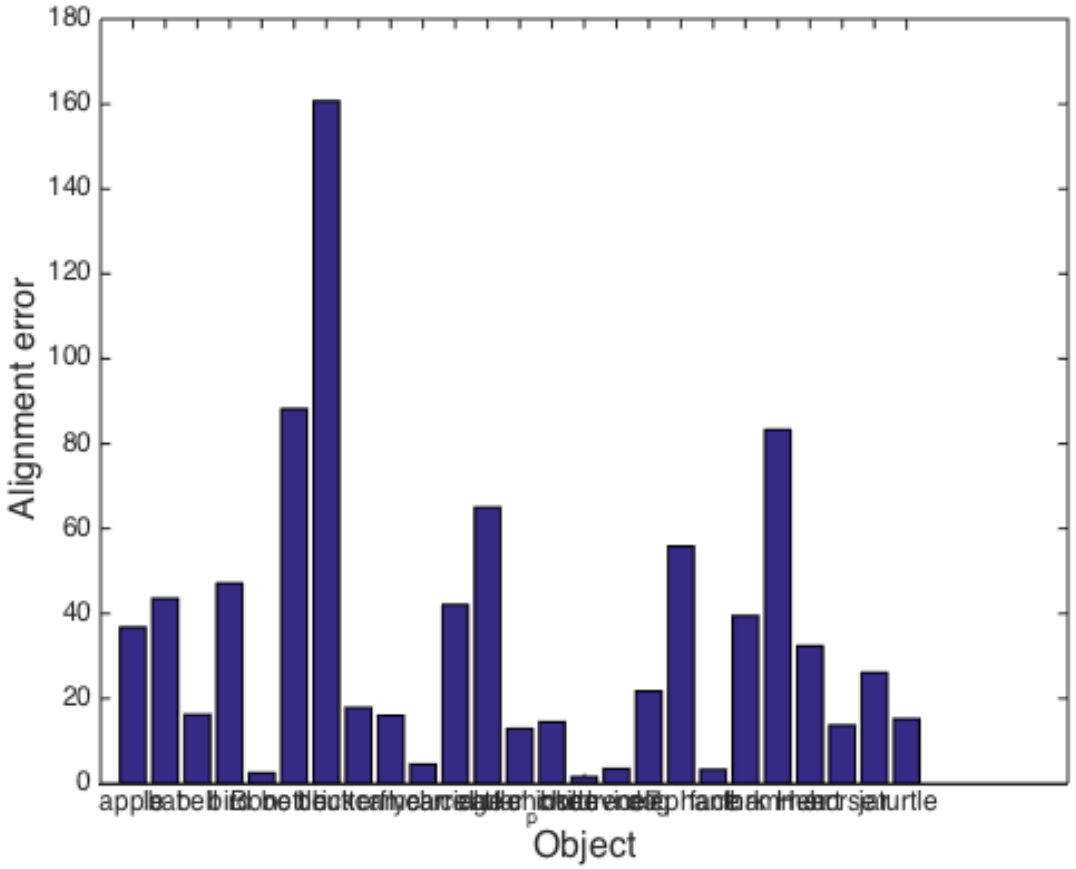
- No initialization
- Bad alpha
- Low iteration count
- Gradient descent tries to compute all 6 values at once.
- Parameters are not normalized for gradient descent. (gradient descent should take in values with similar ranges)
- Error function is not proper. That is, we **do not know** the correct corresponding points between the initial and target images. Therefore, it is very easy for the gradient descent algorithm to find a local minima.

We now try to improve our results with some simple initialization. Namely, we initially resize and relocate the initial image via the variance and mean respectively as suggested in piazza. Thus, we use initialize with matrix:

$$T = \begin{bmatrix} 1 & 0 & \overline{X_2} \\ 0 & 1 & \overline{Y_2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \sigma_2/\sigma_1 & 0 & 0 \\ 0 & \sigma_2/\sigma_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -\overline{X_1} \\ 0 & 1 & -\overline{Y_1} \\ 0 & 0 & 1 \end{bmatrix}$$

We immediately see a reduction in error here at about 38, a significant improve, which certainly helps us stay away from more local minima.





Runtime was asked to be reported. Below I copy pasted the console output to show “runtime” as asked by assignment parameters.

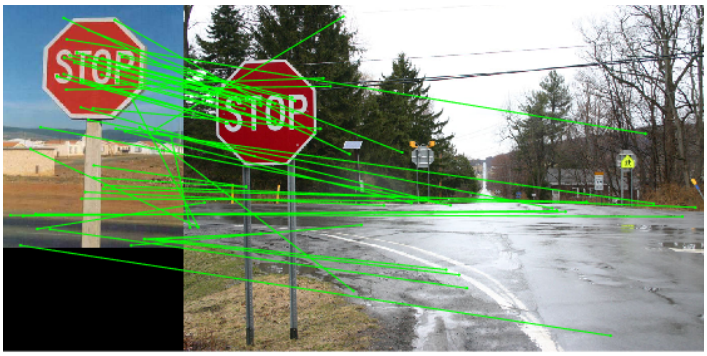
```
Elapsed time is 0.949765 seconds.
Error for aligning "apple": 23.655418
Elapsed time is 6.137880 seconds.
Error for aligning "bat": 57.243755
Elapsed time is 0.612957 seconds.
Error for aligning "bell": 18.073513
Elapsed time is 2.262944 seconds.
Error for aligning "bird": 43.237167
Elapsed time is 2.874812 seconds.
Error for aligning "Bone": 58.441147
Elapsed time is 0.359745 seconds.
Error for aligning "bottle": 89.854324
Elapsed time is 0.619455 seconds.
Error for aligning "brick": 144.248825
Elapsed time is 1.792344 seconds.
Error for aligning "butterfly": 17.919008
Elapsed time is 3.766529 seconds.
Error for aligning "camel": 18.664082
Elapsed time is 0.526300 seconds.
Error for aligning "car": 30.152229
Elapsed time is 0.986276 seconds.
Error for aligning "carriage": 23.687098
Elapsed time is 11.345000 seconds.
Error for aligning "cattle": 57.962730
Elapsed time is 1.774381 seconds.
Error for aligning "cellular_phone": 28.665718
Elapsed time is 0.897944 seconds.
Error for aligning "chicken": 14.053206
Elapsed time is 0.443989 seconds.
Error for aligning "children": 1.358843
Elapsed time is 33.864294 seconds.
Error for aligning "device7": 22.199223
Elapsed time is 13.362881 seconds.
```

```
Error for aligning "dog": 17.823936
Elapsed time is 29.037042 seconds.
Error for aligning "elephant": 34.988213
Elapsed time is 1.061073 seconds.
Error for aligning "face": 2.303580
Elapsed time is 8.025966 seconds.
Error for aligning "fork": 93.154495
Elapsed time is 0.790263 seconds.
Error for aligning "hammer": 99.782326
Elapsed time is 3.617291 seconds.
Error for aligning "Heart": 30.760042
Elapsed time is 27.718484 seconds.
Error for aligning "horse": 13.511835
Elapsed time is 10.207480 seconds.
Error for aligning "jar": 13.523130
Elapsed time is 2.233394 seconds.
Error for aligning "turtle": 18.057838
Averaged alignment error = 38.932867
```

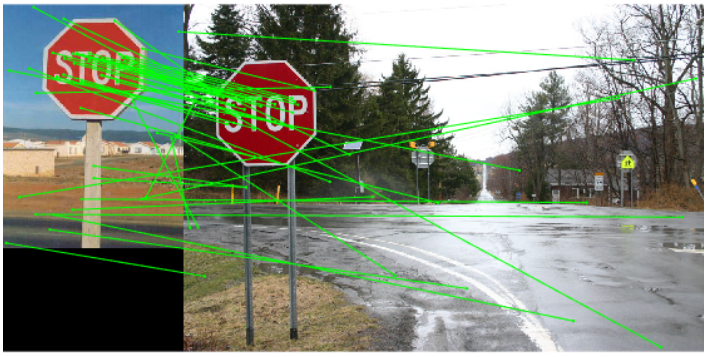
Problem 3

We first show the implementation of both of our results:

Thresholding by distance, (threshold = 450)



Threshold by ratio, (threshold = 0.8)



The ratio is essentially telling us how distinct two similar patches are. Given R = D1/D2, if D1 is small (similar patch) but D2 is large (not similar patch), then it means that a patch has only one other similar patch and is therefore distinct.

The distance is essentially showing us how similar patches are. Therefore:

1) Thresholding by distance: matches similar patches based on highest similarity.
2) Thresholding by ratio: matches similar patches but patches must be distinct.

Graduate Credits

Coarse-to-fine Grain Tracker
Level 1 only [left] Level 2 only [right]

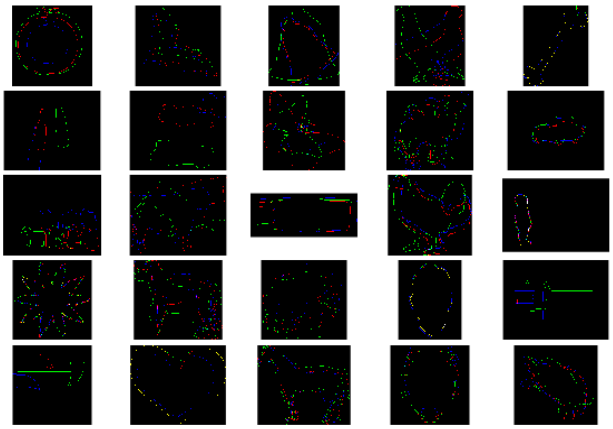
We notice here that the tracking is very bad at the fine level (level 1, on the left). At level 2, it improves significantly. Finally, we should the power of coarse grain tracking with level 3 only in the last picture.

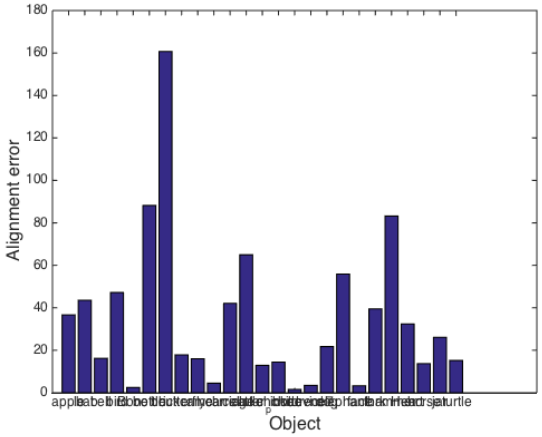
Level 3 tracking. Notice how much better it is.

Optical Flow Generation Attempt:
Please see opticalFlow.m
The pseudo code:
1. Feed ALL pixels into standard KLT tracker (thus tracking every individual pixel).
2. Choose appropriate KLT tracker parameters suck that we are tracking individual pixels.
3. Use the first and last tracked frame to determine the movement of each individual pixel (x and y directions).
4. feed this result into flow matrix (trivial) (HxWx2)

Result: EXTREMELY slow. must track rows * cols number of pixels which is very expensive. I could only run it in reasonable time by reducing the size of the tracked image to ~30x32.
If you have 10,000,000 Ghz computer please try to run it.
30x32 result, followed by 60x64 result:

Improvement of results:
We hypothesize that the gradient descent algorithm trying to calculate a,b,c,d,e,f all at the same time is negatively affecting our results.
We also hypothesized that our alpha and number of iterations may be off.
Let us instead try to ONLY calculate e and f at one time.
When we use only e and f at a time, we get a slight improvement to error of 34.





show how to compute the predicted center position, width, height, and relative orientation of the object in image 2

The relative orientation will be theta_1 - theta_2
The relative scale will be s_r = s_1 / s_2
Width and high can be compute with s_r * [w_2 , h_2] or [w_1/w_2, h_1/h_2]
We can use kepypoint position to determine box center. For example, x2-u2, y2-v2