New York University

Robotics

Professor: Chee Yap

Indoor Line Segment Detection and Recognition

A Robotics Final Project By:
Ryan Cavanaugh,
Evan Johnson,
Eugen Hotaj

May 17, 2016

Abstract

Indoor robotic path planning relies heavily on feature extraction and recognition for map creation. An easy way to extract features is through straight line detection. However, a robot must also be able to re-identify features that are the same across multiple images if the created map is to be robust. In this paper, we show how we can use the Canny Algorithm along with a Hough Space Transform to detect straight lines. Furthermore we implement two different approaches to line re-identification: (1) Distance Comparisons to SIFT points found using Harris & Stephen's Algorithm and (2) building Similarity Maps for lines in order to match similar lines across different images.

1 Background

Our project will make use of Matlab's Canny edge detection algorithm and Hough Transform to find lines. We then use Matlab's Harris & Stephens algorithm implementation (and related methods) to find features that we can associate with the lines.

The Canny edge detection algorithm has 4 major stages: blurring, finding the intensity gradient, edge extraction, and filtering. It first uses a Gaussian blur to remove noise that can cause faulty line detection. On the blurred image, the algorithm then uses an edge detection operator (such as Sobel) to find the first derivative of the gradient in the horizontal and vertical directions (the image is converted to gray scale, so there is only one color channel to determine a gradient). From those gradients, edge gradients ($\mathbf{G}$) and directions (Θ) can be determined:[1]

$$\mathbf{G} = \sqrt{\mathbf{G}_x^2 + \mathbf{G}_y^2} \qquad \Theta = \arctan(\frac{\mathbf{G}_y}{\mathbf{G}_x})$$

Finally, Canny thins these edges to make them just one pixel in width. It also removes low intensity edges (those derived from a small change in gradient) to further remove noise.

The Hough transform takes these edge pixels produced by the Canny algorithm and implements a voting scheme to find straight lines composed of these pixels. It does this by having each edge pixel vote for the set of *all* straight lines that may go through that point. The algorithm considers lines in the following form:[4]

$$r = x \times \cos\Theta + y \times \sin\Theta$$

So the set of all lines that may go through a point (x, y) form a curve on a graph whose axes consist of r and Θ. Locations on this graph where many curves intersect represent many points voting for the same line. Therefore, locations with the highest number of curves intersecting it are considered lines.

The Harris & Stephens algorithm is based on a corner detection algorithm written by Hans Moravec which compares the surroundings of a given pixel to the surroundings of all its neighbors to determine if that pixel is a corner. The algorithm requires large differences to be present along multiple vectors to avoid considering edge pixels as corners. The problem with this is that an edge that doesn't move along the slope of neighboring pixels (horizontal, vertical or 45 degrees) would be considered a corner because it would appear to have large differences across multiple vectors. Harris & Stephens solved this problem by considering the differences with respect to the direction of that difference (instead of using the rigid vectors pointing to the 8 neighboring pixels).

2 Problem Identification

Line detection is a useful step in visual feature extraction and mapping for robotic path planning. However, it's usefulness is quickly deteriorated due to the fact that it is not easy for current vision algorithms to tell if lines found in two different images actually represent the same feature. For example,

suppose a robot takes a picture at time t_0, process it and identifies a feature (i.e line) f. Then at t_1 the robot moves $20cm$ to the right of it's position at t_0, takes another picture and finds feature f'. Using pure line detection, there is no way to decide if $f = f'$ and hence no way for the robot to create a robust, meaningful map of it's environment.

3 Project Aim

The aim of our project is to tackle the above mentioned problem. We plan on doing this by first detecting line segments in images using the Canny Algorithm and Hough Transform, then determining if lines from the different images are actually the same. Refer to the *Theoretical Framework* section below for a technical explanation of our procedure.

4 Research Questions

1. What edge detection algorithms will be helpful to use as a starting point to be used in segment detection?
2. Once we have found points on edges, how can we find the corresponding line segments?
3. Once we have found lines in an image, what is the best method to re-identify lines that have been found in a previous image?

5 Significance of Questions

Edge detection, especially segment detection, can also be used in SLAM algorithms, and mapping algorithms in general. Furthermore, line re-identification serves as a backbone for building a map of an environment. If a robot takes a picture and identifies a line, and then moves and identifies the same line and knows that it is the same line, then the program controlling the robot has an idea of where the robot is in its environment in relation to the found line.

6 Literature Review

[1] A Wikipedia article going over the steps that the Canny algorithm takes.

[2] MathWork's tutorial on using the Computer Vision System Toolbox™ implementation of the Harris-Stephens algorithm.

[3] MathWork's tutorial on using the Image Processing Toolbox™ to detect lines in an image.

[4] A Wikipedia article covering the way Hough represents lines and how it counts the line votes.

[5] A YouTube video describing how the Hough transforms maps line votes on accumulator graph.

[7] A YouTube video describing how the Sobel operator detects edges.

[8] A YouTube video describing how the Canny algorithm detects strong edges

[9] A PowerPoint presentation giving an overview of SLAM using Computer Vision. We did not rely too heavily upon this when implementing our algorithms as this reference deals mostly with points and not lines.

7 Theoretical Framework

As explained, our project aim is twofold: (1) to find line segments in an image and (2) to be able to recognize if lines in two separate images are the same.

Our approach to (1) is a pretty straightforward procedure which has been used by other groups as well to some success. We use the Canny Algorithm [1] to find point proposals (P) for edges in an image. We then feed P to the Hough Transform algorithm which creates line segments (L) from these point proposals based on some previously specified threshold value. Each $l \in L$ is defined by two respective endpoints and the ρ, θ obtained from the Hough Transform.

For (2) we are currently proposing two different approaches. Suppose we have two different images I_1, I_2 and we have found the Hough Lines L_1, L_2 for each image using the procedure described above.

1. For the first approach, which we called Distance Comparison, we use Harris-Stevens Algorithm[2] to find feature points F_1, F_2 in image I_1, I_2 respectively. Then for each $f \in F_i$ we calculate the distance d between f and $l \in L_i$. Then for any two lines $l_\alpha \in L_1, l_\beta \in L_2$, with distances to some feature d_α, d_β, we consider them to be the same line if $|d_\alpha - d_\beta| < \epsilon$ where ϵ is some predetermined threshold. In other words, lines in two different images are considered to be the same if the distance to the same feature in the two images is approximately equal within some ϵ.

2. In the second approach, which we called Similarity Mapping, we define features for each line $l \in L_i$ as follows: first we use the endpoints e_1, e_2; from these, we first obtain two 3×3 matrices S_1, S_2 centered around e_i. These will serve to approximate the locality of each line in an image based on the pixels surrounding the endpoints of that line. We then also find the slope of l and also use the ρ and θ obtained during the Hough Transformation. We these store the representation of l in our map using these features.

   Then for any two lines, $l_1 \in L_1, l_2 \in L_2$ we decide if the lines are the same as follows:

```
if abs(l1.slope−l2.slope) < 3 and
   abs(abs(l1.rho)−abs(l2.rho))<50 and
   abs(abs(l1.theta)−abs(l2.theta))<10 and
   sum(sum(abs(l1.S1 − l2.S1))) < 9 * 20 and
   sum(sum(abs(l1.S2 − l2.S2))) < 9 * 20
then
    return true
```

   To build the map, for each image I_i we iterate through the lines and for each $l \in L_i$ we first check if it matches any line already stored in our map. If l is distinct, we add it to the map and choose some random color to represent it. Therefore, across different images, lines with the same color correspond to the same line store in the map.

   Since Hough features are a big part of our map building, this algorithm is not very robust in the sense that for it to correctly identify lines across two different images, the images must not be rotated and have to have the same dimensions and resolution.

8 Research Methods

We watched YouTube videos on the Canny edge detection algorithm and Hough transform [8, 5]. We also read the Wikipedia articles on these topics as well. This provided a background understanding of the problems. Finally we read some papers about feature re-identification [9] but as previously mentioned most of them dealt with point re-identification which was somewhat unhelpful in re-identifying lines.

MathWorks provided several online tutorials[3, 2] which were helpful in understanding how we would utilize the Image Processing Toolbox™ and Computer Vision System Toolbox™[6], in particular the Canny,

Hough, and Harris-Stephens algorithm implementations. The code found on these tutorials serves as the backbone to our MATLAB function.

9 Results

Refer to the *Theoretical Framework* for an explanation of the two different approaches we took to re-identify lines. In all figures, lines of the same color are considered to be the same.

1. We created two variants of the Distance Comparison approach: one which measures the distances between all lines and all features and another which only measures the distances between the longest line and all features.

   The first variant matches lines from the second image to lines from the first image based on the closest match of distances from the lines to all matched features. If the closest match is below a set threshold, specified by the matchThresh argument (suggested value of .8-.98), then the match will not be included. When good thresholds were provided this variant produced some great results. (see *Figure 3* & *Figure 4*) Since this algorithm is only considering the sum of all distances from the line to every feature, and not taking into account the direction of the distance, it produces some unexpected results with some of our test images (see *Figure 5*).

   Our second variant only compares the longest lines detected by Canny and Hough and therefore relies on these algorithms finding the same line to produce meaningful results (see *Figure 7*). Additionally, this variant relies on the Harris & Stephens algorithm having perfect feature matching across the two images (see *Figure 2*). It compares the most dissimilar distances to avoid being fooled by coincidentally equidistant features. The result of all this is that the second variant rarely returns a match for two longest lines detected in an image, but when it does, it does it with a high level of certainty.

   Since these methods rely on the lines having the same distance to features, it will not work on images (even identical ones) with different scales, as changing the scale will change all of the distances. However, it can be assumed that in the real world application of processing frames of video footage the difference in scale between two consecutive frames will never be very great and so with a reasonable threshold this method of line re-identification is still a viable solution.

2. The second approach, Similarity Mapping, is not as robust as our first approach in terms of invariance to rotation as it does not make use of SIFT points (like Harris-Stevens). *Figure 8* shows an image and it's rotated version (right). As one can see, even though it is clear that the line identified in both images should be the same, the algorithm identifies two distinct lines.

   As a baseline test, the algorithm does a great job in re-identifying lines in the same exact image (as one should expect). One thing to note is that the swaths (S_1, S_2) taken around the endpoints of lines help to distinguish lines which are very similar in slope. *Figure 9* demonstrates this phenomenon: as one can see, without the swaths, what should be two different lines in the right image correspond to the same line in the left image (the cyan colored lines) since their slopes and Hough Features are very similar. However, once we account for swath similarity as in *Figure 10*, all lines are correctly identified.

   Of course identifying lines in the same image is a pretty trivial process, and we should expect all lines to be correctly re-identified. *Figure 11* shows the algorithm trying to re-identify lines in two similar images. As one can see, only the big vertical line is correctly identified since the slopes, Hough features, and the swaths are within our comparison thresholds. The algorithm decides that all other lines are different.

10 Conclusions

The Distance Comparison algorithm (our first approach) is much more robust since it uses SIFT points found with the Harris-Steven's algorithm. Therefore, it can correctly identify lines across images which have been rotated (as seen below) and images which have been slightly scaled - if two images have very differing resolutions, the algorithm will most likely produce incorrect results; this is not necessarily a bad thing since for robotic motion planning we assume the sensor (in this case a camera) stays constant throughout the movement pattern of the robot. However, the robot must roughly maintain its distance from the lines and features in order to maintain the scale across frames which may not be a realistic expectation. Additionally, the algorithm relies on sturdy features: if features move between frames, the algorithm will not be able to match lines (this is especially true of the Longest Line variant).

The Similarity Mapping Algorithm works relatively well if Images are not too dissimilar. However, after analyzing our results, mostly due to the naive similarity comparisons done by this algorithm, the Distance Comparison Algorithm seems to be a much better, reliable choice for line re-identification. One way to improve on the Similarity Mapping is to consider more similarity swaths along the line, instead of just on the endpoints, as this might give us a more robust comparison across lines. Finally, using a convolution kernel on the similarity swaths (such as a Gaussian Kernel) in order to reduce noise, and comparing the convoluted swaths might give more robust comparisons.

11 Appendix

11.1 Appendix A - Contributions

Report:

- Background, Literature Review - Ryan
- Problem Identification, Project,Aim Theoretical Framework - Eugen
- The rest of the report sections were a group effort.

Presentation:

- The presentation was a group effort overall, with each group member focusing on their particular implementation explanation (see below).

Solution:

- Similarity Mapping - Eugen
- Distance Comparison (Concept and All Lines Variant) - Evan
- Distance Comparison (Longest Line Variant) - Ryan

11.2 Appendix B - Code

This section contains the code to the different approaches for line re-identification mentioned in the *Theoretical Framework*.

11.2.1 Distance Comparison Variant 1

This first variant offers extra flexibility in setting the thresholds of the Hough algorithm and the threshold to declare a match as arguments to the reidentifyLines function call.

```
function reidentifyLines(inputImages, fill, minLine, matchThresh)
%reidentifyLines Given array of image names, find lines and reidentify
    colors = {'b','g','y','m','c','b','g','y','m','c','b','g','y','m','c','r'};
    for imgNum = 1:length(inputImages(:,1))
        %read image
        curImg = imread(inputImages(imgNum,:));
        %convert to grayscale to work with the edge function
        curImg = rgb2gray(curImg);

        %perform canny alg on image
        cannyThresh = [0.05, 0.3];
        cannyImg = edge(curImg, 'canny', cannyThresh);

        %perform hough transform on result of canny
        [H,theta,rho] = hough(cannyImg);
        P = houghpeaks(H,5,'threshold',ceil(0.4*max(H(:))));
        lines = houghlines(cannyImg,theta,rho,P,'FillGap',fill,'MinLength',minLine);

        %first imgage is just setup - store first image and first image lines
        if imgNum == 1
            last_img = curImg;
            %find image features from Harris-Stephens algorithm
            last_points = detectHarrisFeatures(curImg);
            [last_features,last_valid_points] = extractFeatures(curImg,last_points);
            last_lines = lines;
        %subsequent images get lines of current image and compare to last
        else
            curPoints = detectHarrisFeatures(curImg);
            [features_cur,valid_points_cur] = extractFeatures(curImg,curPoints);
            %match the feature points of the two images
            indexPairs = matchFeatures(last_features,features_cur);
            matchedPoints1 = last_valid_points(indexPairs(:,1),:);
            matchedPoints2 = valid_points_cur(indexPairs(:,2),:);

            %show the matched features on a figure
            figure
            showMatchedFeatures(last_img,curImg,matchedPoints1,matchedPoints2);

            last_lines_dist = [1:length(last_lines);1:length(last_lines)];

            %for each line in last lines, find total distance from all matched points
            for lastCurLineNum = 1 : length(last_lines)
                lineStart = last_lines(lastCurLineNum).point1;
                lineEnd = last_lines(lastCurLineNum).point2;
                P = [lineStart; lineEnd];
                curDist = 0;
                %for each feature, accumulate total distance from line to
                %all features
                for pointNum = 1 : length(matchedPoints1)
                    [curSep, throwaway] = sep(matchedPoints1(pointNum).Location',P');
                    curDist = curDist + curSep;
                end
                %store distance to current line in last_lines_dist
                last_lines_dist(2,lastCurLineNum) = curDist;
```

```
55              end
56
57              figure, imshow(last_img), hold on
58              for k = 1:length(last_lines)
59                 xy = [last_lines(k).point1; last_lines(k).point2];
60                 plot(xy(:,1),xy(:,2),'LineWidth',3,'Color',colors{k});
61              end
62
63              cur_lines_dist = [1:length(lines);1:length(lines)];
64              %repeat above process for the current lines and features
65              for curLineNum = 1 : length(lines)
66                  lineStart = lines(curLineNum).point1;
67                  lineEnd = lines(curLineNum).point2;
68                  P = [lineStart; lineEnd];
69                  curDist = 0;
70                  for pointNum = 1 : length(matchedPoints2)
71                      [curSep, throwaway] = sep(matchedPoints2(pointNum).Location',P');
72                      curDist = curDist + curSep;
73                  end
74                  cur_lines_dist(2,curLineNum) = curDist;
75              end
76
77              matched_index = [1:length(cur_lines_dist)];
78
79              %compare cur_lines_dist to last_lines_dist
80              for lineIndex = 1 : length(cur_lines_dist)
81                  best = 100;
82                  bestI = 16;
83                  for matchIndex = 1 : length(last_lines_dist)
84                      cur = abs(1 - (cur_lines_dist(2,lineIndex) / last_lines_dist(2,matchIndex)));
85                      if cur < best && cur < (1 - matchThresh)
86                          best = cur;
87                          bestI = matchIndex;
88                      end
89                  end
90                  matched_index(1,lineIndex) = bestI;
91              end
92
93              figure, imshow(curImg), hold on
94              for k = 1:length(lines)
95                 xy = [lines(k).point1; lines(k).point2];
96                 plot(xy(:,1),xy(:,2),'LineWidth',3,'Color',colors{matched_index(1,k)});
97              end
98
99              %set cur image to last image - will be compared to the next img
100             last_img = curImg;
101             last_features = features_cur;
102             last_valid_points = valid_points_cur;
103             last_lines = lines;
104         end
105     end
106 end
```

11.2.2 Distance Comparison Variant 2

The portion of code that makes this variant unique from the other begins on line 66.

```
1  function reidentifyLongLine( inputImages )
2  %reidentifyLines Given array of image names, find lines and reidentify
3
4      %Variable used to store the longest line
5      longest_line1 = -1;
6
7      for imgNum = 1:length(inputImages(:,1))
```

```
        %read image
        curImg = imread(inputImages(imgNum,:));
        %convert to grayscale to work with the edge function
        curImg = rgb2gray(curImg);
        %perform canny alg on image
        cannyThresh = [0.05, 0.3];
        cannyImg = edge(curImg, 'canny', cannyThresh);
        %perform hough transform on result of canny
        [H,theta,rho] = hough(cannyImg);

        %from matlabs findlines tutorial
        P = houghpeaks(H,5,'threshold',ceil(0.3*max(H(:))));
        x = theta(P(:,2));
        y = rho(P(:,1));

        lines = houghlines(cannyImg,theta,rho,P,'FillGap',25,'MinLength',50);

        figure, imshow(curImg), hold on
        max_len = 0;
        for k = 1:length(lines)
           xy = [lines(k).point1; lines(k).point2];
           plot(xy(:,1),xy(:,2),'LineWidth',2,'Color','green');

           % Plot beginnings and ends of lines
           plot(xy(1,1),xy(1,2),'x','LineWidth',2,'Color','yellow');
           plot(xy(2,1),xy(2,2),'x','LineWidth',2,'Color','red');

           % Determine the endpoints of the longest line segment
           len = norm(lines(k).point1 - lines(k).point2);
           if ( len > max_len)
              max_len = len;
              xy_long = xy;
           end
        end
        % highlight the longest line segment
        plot(xy_long(:,1),xy_long(:,2),'LineWidth',2,'Color','red');

        if imgNum == 1
            %Save longest line, featureset, and image for use during second
            %image processing
            last_line = xy_long;
            last_img = curImg;
            last_points = detectHarrisFeatures(curImg);
            [last_features,last_valid_points] = extractFeatures(curImg,last_points);
        else
            %Extract points using Harris & Stephens
            curPoints = detectHarrisFeatures(curImg);
            [features_cur,valid_points_cur] = extractFeatures(curImg,curPoints);

            indexPairs = matchFeatures(last_features,features_cur);

            matchedPoints1 = last_valid_points(indexPairs(:,1),:);
            matchedPoints2 = valid_points_cur(indexPairs(:,2),:);

            maxDifference = -1;
            %Combine the two images for display later
            C = imfuse(curImg,last_img,'blend','Scaling','joint');

            if (length(matchedPoints1)>0)
                %Loop through all matched features
                for i = 1:length(matchedPoints1)
                    %Compute distances from the longest lines to a matched feature
                    distance1 = sep(matchedPoints1(i).Location',last_line');
                    distance2 = sep(matchedPoints2(i).Location',xy_long');

                    difference = abs(distance1 - distance2);
```

```
                    %Set maxDifference to the largest discrepancy between
                    %any distance1 and distance2 values
                    if (difference>maxDifference)
                        maxDifference = difference;
                    end
                end

                figure, imshow(C), hold on
                %If there was a decently low maximum difference, highlight
                %the two longest lines to show they are the same
                if (maxDifference < 200)
                    plot(xy_long(:,1),xy_long(:,2),'LineWidth',2,'Color','blue');
                    plot(last_line(:,1),last_line(:,2),'LineWidth',2,'Color','blue');
                end
            end

            %If there were no matched features, display the overlayed images
            %without highlighting lines to suggest no findings
            if (maxDifference == -1)
                figure, imshow(C), hold on
            end

            figure;
            showMatchedFeatures(last_img,curImg,matchedPoints1,matchedPoints2);

            %reset last_img params (for more than 2 images)
            last_line = xy_long;
            last_img = curImg;
            last_features = features_cur;
            last_valid_points = valid_points_cur;

        end %end if that checks which image is being processed
    end %end for that loops through input images
end
```

11.2.3 Sep Function

The function used in the previous two functions to compute the distance between a point (feature) and a line.

```
function [s,p] = sep(q,P)
% separation between point q and polygon P (actually, only a point or an edge)
    if size(P,2) == 1 % P is a point
        s = norm(P-q);
        p = P;
    elseif size(P,2) == 2 % P is a pair of points
        a = P(:,1);
        b = P(:,2);

        t = dot(q-a,b-a)/norm(b-a)^2;

        if t <= 0 % q is closest to a
            s = norm(a-q);
            p = a;
        elseif t >= 1 % q is closest to b
            s = norm(b-q);
            p = b;
        else
            p = (1-t)*a+t*b; % q is closest to some point in between a and b
            s = norm(p-q);
        end
    end
end
```

11.2.4 Similarity Mapping

The portion of code that makes this approach unique from the two variants above begins on line 29.

```
1  function[map] =  reidentifyLines_swath( inputImages )
2  % reidentifyLines Given array of image names, find lines and reidentify
3  % this is the swath version of reidentify lines
4
5      map = {};
6
7      for imgNum = 1:length(inputImages(:,1))
8
9          % read image and convert to grayscale
10         curImg = imread(inputImages(imgNum,:));
11         curImg = rgb2gray(curImg);
12
13         % find canny point proposals
14         cannyThresh = [0.05, 0.3];
15         cannyImg = edge(curImg, 'canny', cannyThresh);
16
17         % find lines through hough transform on canny proposals
18         [H,theta,rho] = hough(cannyImg);
19         P = houghpeaks(H,5,'threshold',ceil(0.3*max(H(:))));
20         lines = houghlines(cannyImg,theta,rho,P,'FillGap',200,'MinLength',250);
21
22         % plot
23         figure, imshow(curImg), hold on
24
25         % append to map for current image
26         map = swath_map(lines,map,curImg);
27     end
28
29     function[map] = swath_map(lines, map, img)
30         newmap = {}; % current image map
31         len = 0; % current image map length
32
33         for i = 1:length(lines)
34             % points array for easier plotting
35             xy = [lines(i).point1; lines(i).point2];
36
37             curr = lines(i);
38             flag = 1;           % flag to check if line in global map
39             match = -1;         % index of match
40             color = rand(1,3); % random color to assign a new line
41
42             % build line features
43             p1 = curr.point1;
44             p2 = curr.point2;
45
46             t = curr.theta;
47             r = curr.rho;
48             slope = (p2(2) - p1(2))/(p2(1) - p1(1));
49             p1_swath = img(p1(1)-1:1:p1(1)+1,p1(2)-1:1:p1(2)+1);
50             p2_swath = img(p2(1)-1:1:p2(1)+1,p2(2)-1:1:p2(2)+1);
51
52             % creat feature struct
53             curr_s = struct('s',slope, 'p1_swath',p1_swath, ...
54                             p2_swath',p2_swath,'color',color, ...
55                             'rho',r, 'theta', t);
56
57             % (naive) search for match
58             for j=1:length(map)
59                 if test_same(map{j}, curr_s)
60                     flag = 0;
61                     match = j;
```

```
                    break;
                end
            end

            % plotting and current image map construction
            if flag
                % if line not in global map, plot, add to current image map
                len = len + 1;
                newmap{len} = curr_s;
                match = len;
                plot(xy(:,1),xy(:,2),'LineWidth',3,'Color',newmap{match}.color);
            else
                % if line in global map, plot with corresponding color
                plot(xy(:,1),xy(:,2),'LineWidth',3,'Color',map{match}.color);
            end

            % plot endpoints of line
            plot(xy(1,1),xy(1,2),'x','LineWidth',2,'Color','yellow');
            plot(xy(2,1),xy(2,2),'x','LineWidth',2,'Color','red');
          end

        % append current image map to global map
        lenmap = length(map);
        for i = 1:length(newmap)
            lenmap = lenmap + 1;
            map{lenmap} = newmap{i};
        end
    end

    % similarity function (thresholds based on best attained results)
    function[bool] =  test_same(l1,l2)
        bool = 0;
        if abs(l1.s-l2.s) < 3 && ...
           abs(abs(l1.rho)-abs(l2.rho))<50 && ...
           abs(abs(l1.theta)-abs(l2.theta))<10 && ...
           sum(sum(abs(l1.p1_swath - l2.p1_swath))) < 9 * 20 && ...
           sum(sum(abs(l1.p2_swath - l2.p2_swath))) < 9 * 20
            bool = 1;
        end
    end
end
```

11.3 Appendix C - Figures

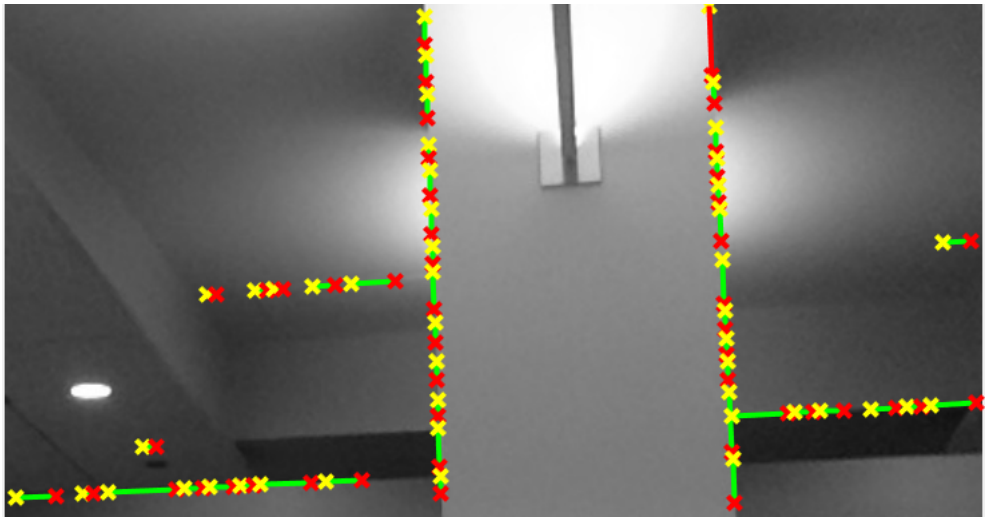

Figure 1: An initial attempt to find lines.

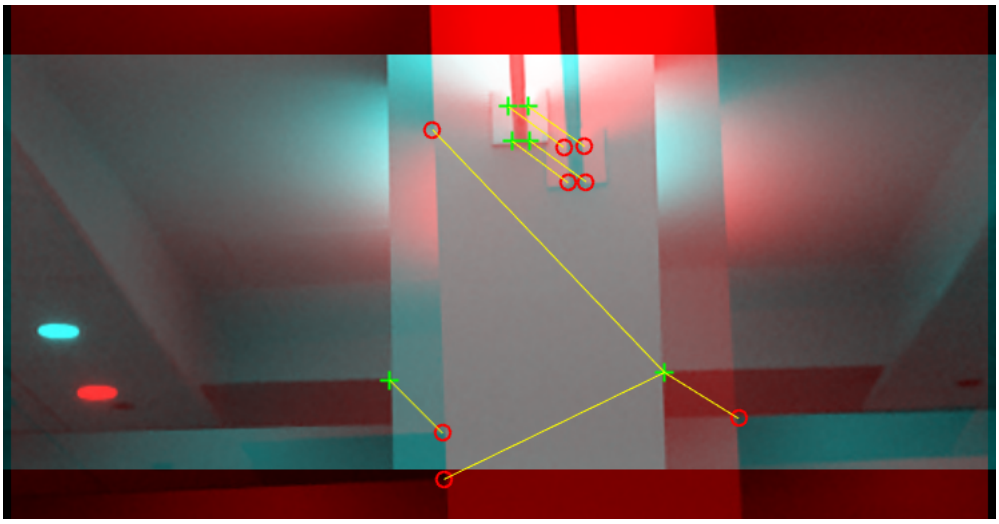

Figure 2: The Harris & Stephens algorithm attempting to match sets of features in similar pictures. There is an incorrect match between a feature on the left of the red image and the right of the blue image. This can cause incorrect results for our Distance Comparison technique.

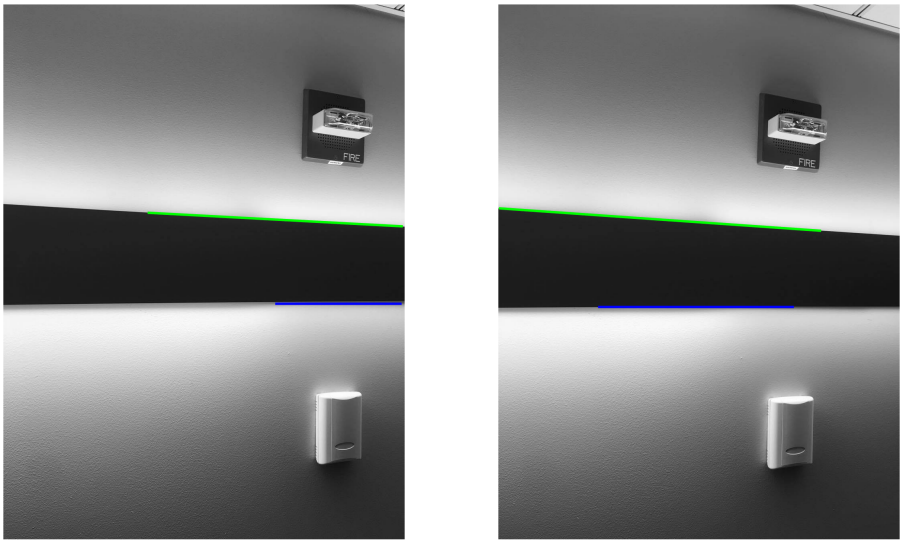

Figure 3: With a 99% similarity threshold for a match, the algorithm correctly re-identifies both major edges in these test images, although the start/end points of the liens could use some work.

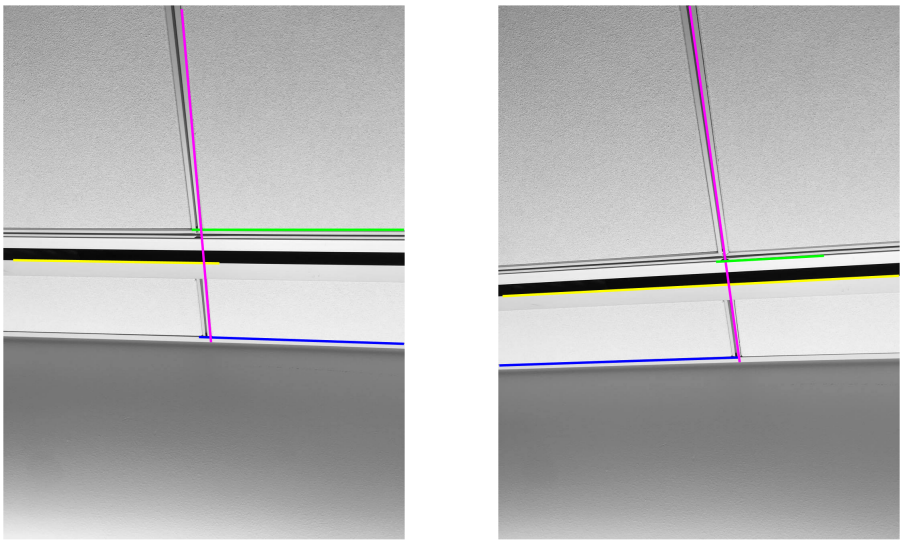

Figure 4: The algorithm correctly re-identifies all 4 lines found in this image, although it took a 84% match threshold, which is relatively low, in order for the algorithm to declare the long vertical line as matched.

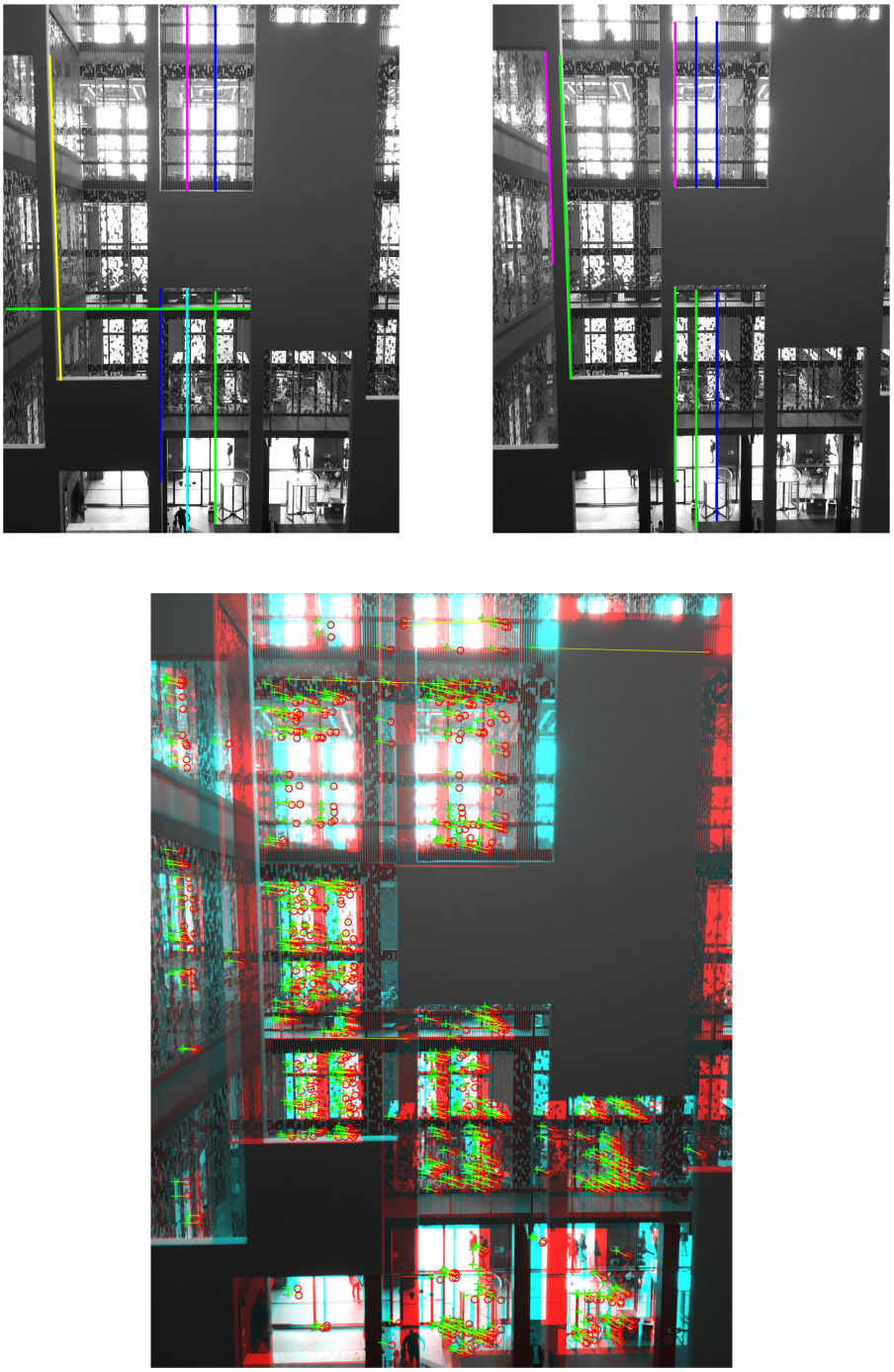

Figure 5: This image with many more feature points than previous images and many more lines misidentifies lines between the two images (note that unmatched lines would appear in red). This mismatching was produced at a 95% match threshold so clearly this method has flaws.

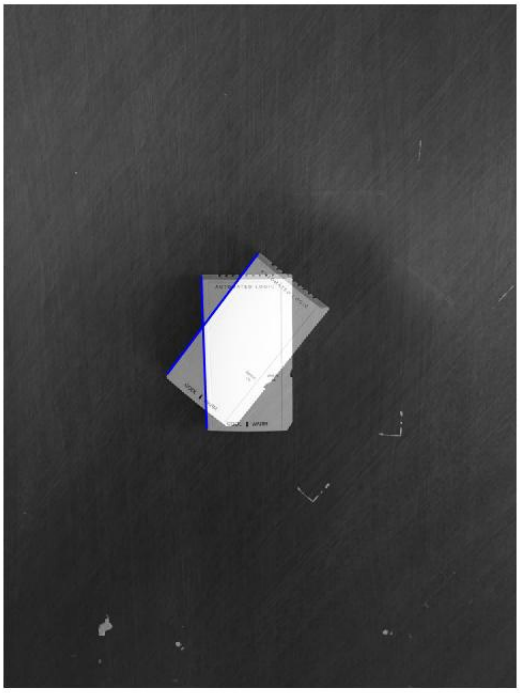

Figure 6: An instance of successful line matching using the Distance Comparison (Longest Line Variant). The longest lines are drawn in blue.

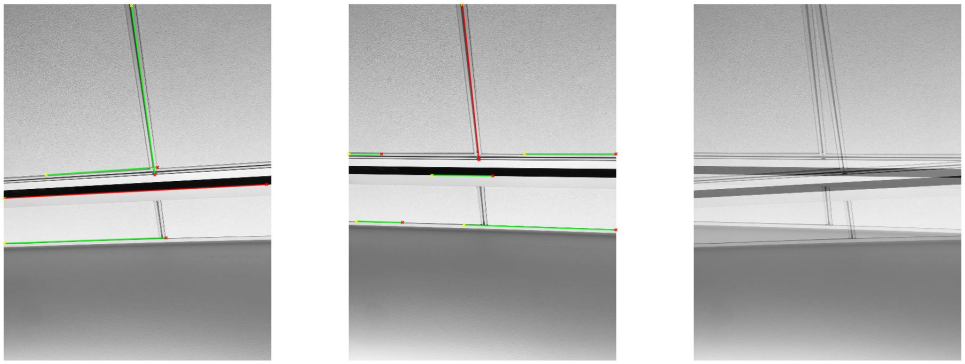

Figure 7: An instance of *unsuccessful* line matching using the Distance Comparison (Longest Line Variant). Here we see Hough found different lines that it considered "longest line" in each picture (drawn in red) so a longest line match is not possible.

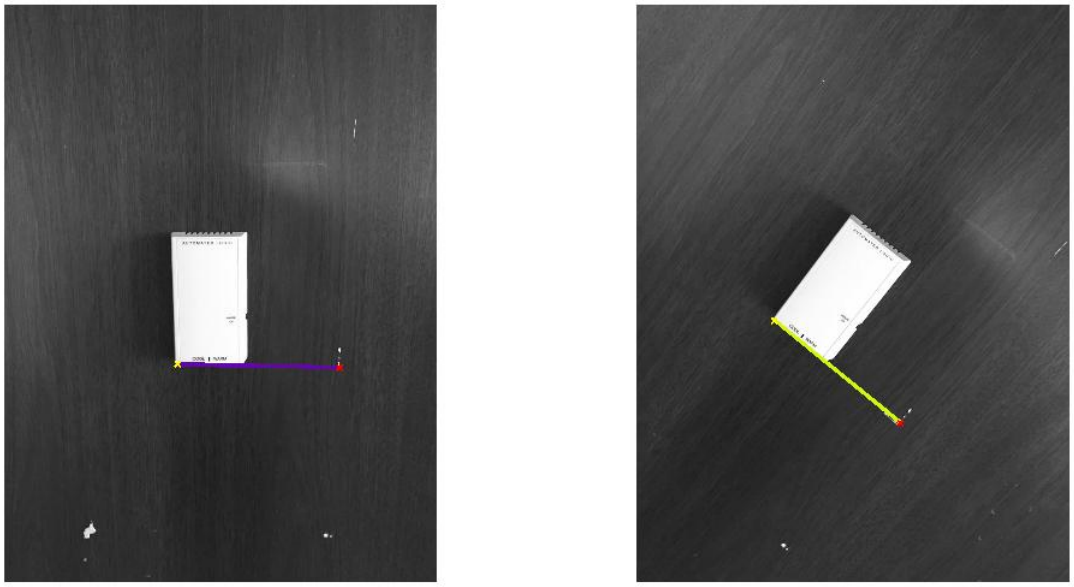

Figure 8: The Similarity Mapping method is not robust with respect to rotation as demonstrated here. One can clearly see the line is not re-identified.

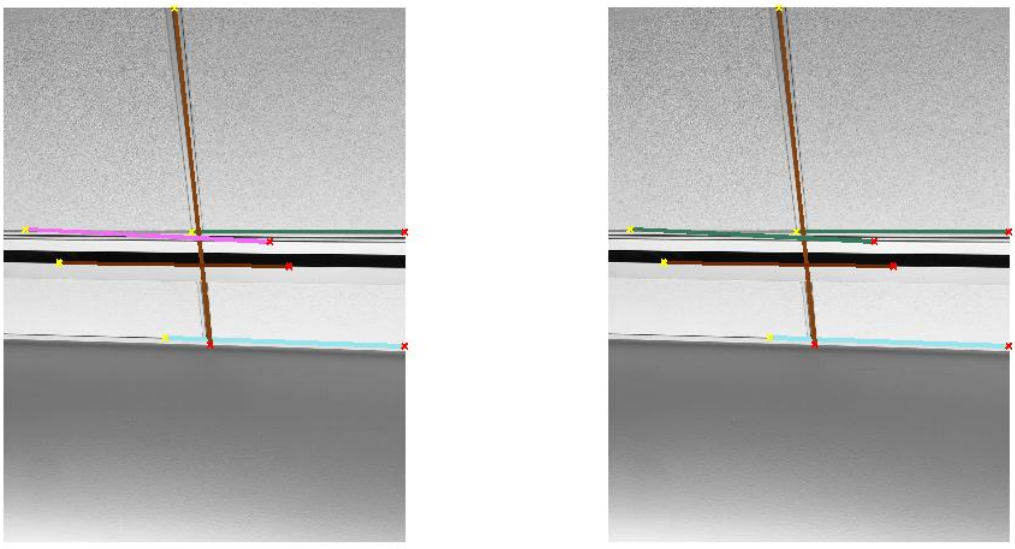

Figure 9: Incorrect classification of lines on the same image with Similarity Mapping. This is due to lack of endpoint similarity swaths. As one can see, the cyan colored line in the middle on the left image is detected twice on the right image.

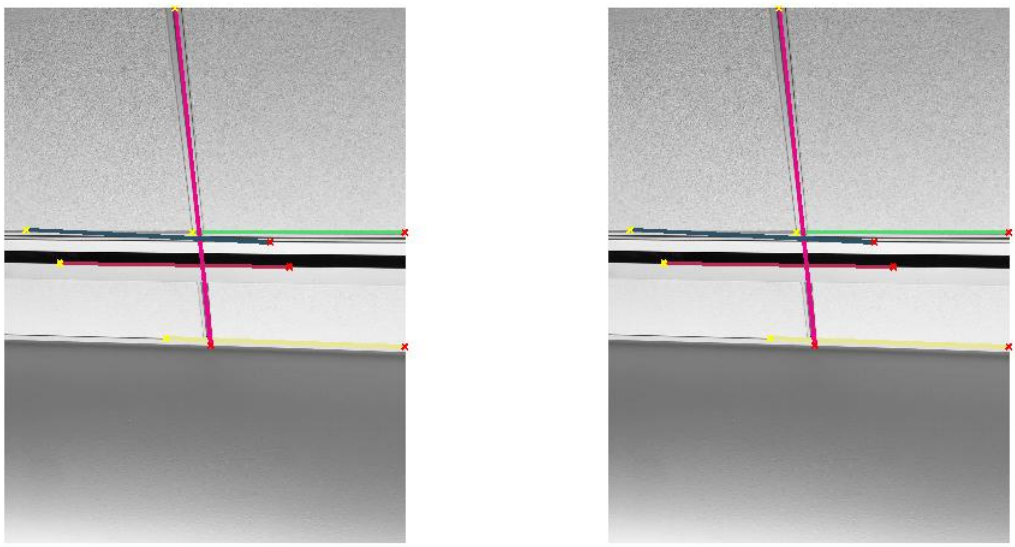

Figure 10: Correct classification of lines on the same image using Similarity Mapping with endpoint similarity swaths.

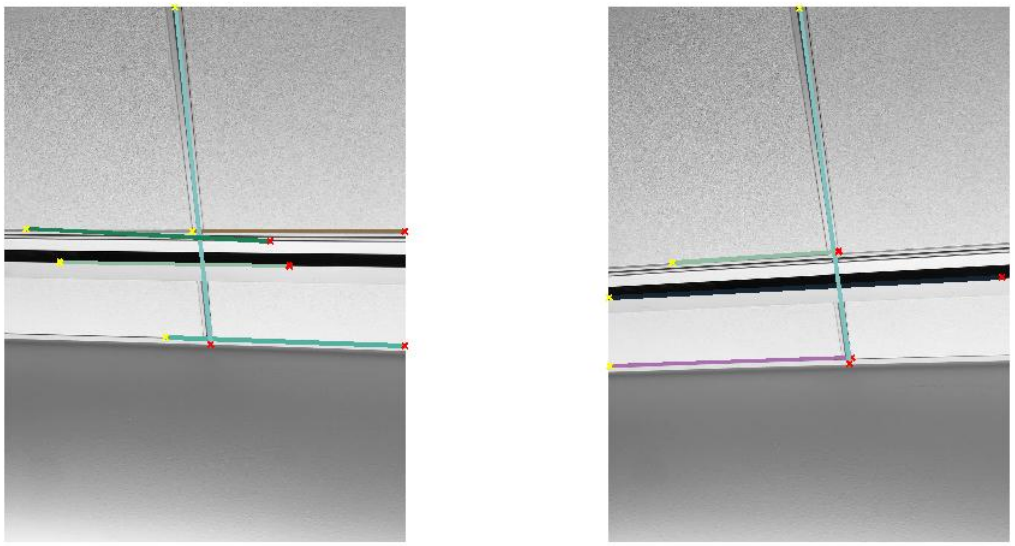

Figure 11: Classification using Similarity Mapping on two similar but distinct images. Only the big vertical line is correctly classified. All other lines are (incorrectly) considered unique by the algorithm.

References

[1] Canny Edge Detector. (n.d.). Retrieved May 02, 2016, from https://en.wikipedia.org/wiki/Canny_edge_detector

[2] Detect Harris Features. (n.d.). Retrieved May 02, 2016, from http://www.mathworks.com/help/vision/ref/detectharrisfeatures.html MathWorks description of using MATLAB's Harris-Stephens algorithm to detect corner points in an image.

[3] Detect Lines in Images Using Hough. (n.d.). Retrieved May 02, 2016, from http://www.mathworks.com/help/images/hough-transform.html#buh9ylp-26 MATLAB tutorial on using Hough transform to find lines in images.

[4] Hough Transform. (n.d.). Retrieved May 02, 2016, from https://en.wikipedia.org/wiki/Hough_transform

[5] Körting, T.S. [Thales Sehn Körting]. (2016, April 16). *How Hough Transform works* [Video File]. Retrieved from https://www.youtube.com/watch?v=4zHbI-fFIlI

[6] MATLAB & SIMULINK student version [Computer software]. (n.d.). MATLAB software used for our line detection and re-identification.

[7] Pound, M. [Computerphile]. (2015, November 4). *Finding the Edges (Sobel Operator) - Computerphile* [Video File]. Retrieved from https://www.youtube.com/watch?v=uihBwtPIBxM

[8] Pound, M. [Computerphile]. (2015, November 11). *Canny Edge Detector - Computerphile* [Video File]. Retrieved from https://www.youtube.com/watch?v=sRFM5IEqR2w

[9] Ilić, S. [Technischen Universität München]. (Winter 2009/2010) *Tracking and Detection in Computer Vision using SLAM*. Retrieved May 02, 2016 from http://campar.in.tum.de/twiki/pub/Chair/TeachingWs09MATDCV/SLAM.pdf