Project 4 Report

Feature Detection and Matching

CPE428-01,02 - Computer Vision

Prepared for:	Professor Xiaozheng Zhang
Prepared By:	The “A” Team; Nikhil Patolia, Alec Hardy
Date Submitted:	Saturday, February 15, 2020

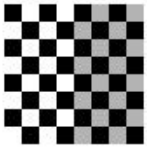
Table of Contents

Part 1 - Feature Detection

For Part 1 of the project, the object was to get the main features of a given image. We were meant to use a checkerboard image so the feature points of this would be the corners created by the shifting of colors from black to white. To obtain the locations of the corners, we used the Harris Corner Detector and did not use any MATLAB functions.

Compute Gaussian derivatives (horizontal and vertical) at each pixel

The first step of the process was to get the checkerboard image.



Then we computed the Gaussian Derivates and labelled them Ix for the horizontal and Iy for the vertical. The Gaussian Derivatives are obtained by filtering the image with a prewitt filter

Compute second moment matrix M in a Gaussian window around each pixel

The next step in the Harris Corner Detector is to compute the second moment matrix. This was done by getting the Ix^2, Iy^2 and the Ix*Iy. This computation was done on three different matrices and then later combined together for one R Score. The second moment matrix is the following matrix for each pixel.

Ix^2	IxIy
xIy	Iy^2

Compute corner response function R

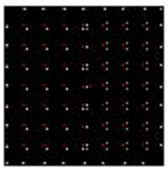
Then we computed the R score by getting the determinant of the second moment matrix which is the product of the right diagonals minus the product of the left diagonals. Once you have the second moment matrix, you subtract alpha which is 0.5 times the trace of the second moment matrix squared. This value is the R score and there is an R score for each pixel. The R score gives the locations where the values change the most in each direction.

Threshold R

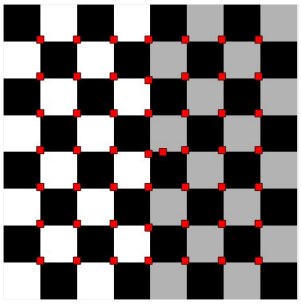
Next we set a threshold for the image to make it binary and also to only get the most prominent spots on the image. The threshold in this case was set to 0.5.

Find local maxima of response function (non- maximum suppression)

Then, we found each location of the pixels that were white in this binary image. With these locations, we looked at the R score image and checked each pixel to see if it was the max of its neighbors. If it was the max value of its neighbors, we kept it, but if it wasn't, we set it to zero. After this w



Next, we used bwconncomp to cluster the remaining points together into groups and overlaid those groupings over the original image.



Adjustments

Gaussian filter size

Adjusting the gaussian filter size changed how much blurring there was and since this image is so clear where the corners are, increasing the gaussian filter size blurred the image more than it needed to be and caused the accuracy to decrease. Increasing the gaussian filter size can be used for an image that has too many fine details but for a simple image such as a checkers board, we can use a 3x3 gaussian filter.

Threshold value for validating a corner

Adjusting the threshold value for the R score made a huge difference for the end product. Having a high threshold such as 3 only allowed one point to show up as a corner for the whole image. On the other hand, having anything less than 1 was ideal and it caught all the relevant corners. There is one detected corner that shouldn't be there(the one in the middle of two squares) but even reducing the threshold very low did not get rid of it while still keeping the relevant corners.

Part 2 - Feature Description

Two functions, "my_ExtractFeatures_a" and "my_ExtractFeatures_b", were written to extract feature vectors from feature points. The feature points could be found using the Harris Corner Detection method from Part 1, but in order to guarantee a higher metric of detected feature points, MATLAB's built-in

"detectFASTFeatures" method was used. The feature vector extraction functions take in an image matrix and a matrix of (x,y) coordinates as arguments and return a matrix of row vectors corresponding to the coordinates containing the feature descriptors.

The feature points were extracted as follows, where I_n are grayscale image matrices :

```
%% Extract feature points
% Feature points in [x,y] coordinates
% The selectedStrongest(N) value is chosen manually such that there are not
% more than 100 FP matches after performing the bestMatch() function.
fp1 = selectStrongest(detectFASTFeatures(I1),200).Location;
fp2 = selectStrongest(detectFASTFeatures(I2),200).Location;
fp3 = selectStrongest(detectFASTFeatures(I3),200).Location;
fp4 = selectStrongest(detectFASTFeatures(I4),200).Location;
fp5 = selectStrongest(detectFASTFeatures(I5),400).Location;
fp6 = selectStrongest(detectFASTFeatures(I6),400).Location;
```

my_ExtractFeatures_a determines feature vectors by returning the raw pixel values in a 5x5 window around the feature point. If the feature point is on the edge of the image, zero-padding is used. The feature vectors are extracted using the following code, and the function definition is in the Appendix of the report:

```
% Use crappy neighboring raw-pixel values
ef_1b = my_extractFeatures_a(I1, fp1);
ef_2b = my_extractFeatures_a(I2, fp2);
ef_3b = my_extractFeatures_a(I3, fp3);
ef_4b = my_extractFeatures_a(I4, fp4);
ef_5b = my_extractFeatures_a(I5, fp5);
ef_6b = my_extractFeatures_a(I6, fp6);
```

my_ExtractFeatures_b takes in an additional parameter - an image filter to use before processing. For correct use, a gaussian filter with tuned parameters of σ shall be used. The function works by first padding the image with 8 zeros on all sides to avoid getting errors. Next, we get the locations of the feature points that were obtained by using the FAST feature detector and drawing a 16x16 grid around each of these feature points. Then we put a gaussian blur on this subgrid and get the gradient direction and magnitude. We round each gradient direction to the closest 8 directional and then split the 16x16 grid into 16, 4x4 grids. Using these smaller grids, we sum up the magnitudes of each directional in that 4x4 grid and then insert this value in a vector. Each 4x4 grid has 8 directionals and there are 16 4x4 grids per 16x16 grid. This gives us a 128 feature descriptor for each feature point, giving us far more information to do feature matching.

The implementation of the function is shown below, and the function definition is included in the Appendix of this report.:

```
% Use SIFT-like feature extraction algorithm
ef_1 = my_extractFeatures_b(I1, fp1, fspecial('gaussian', 5, 1));
ef_2 = my_extractFeatures_b(I2, fp2, fspecial('gaussian', 5, 1));
ef_3 = my_extractFeatures_b(I3, fp3, fspecial('gaussian', 3, .5));
ef_4 = my_extractFeatures_b(I4, fp4, fspecial('gaussian', 3, .5));
ef_5 = my_extractFeatures_b(I5, fp5, fspecial('gaussian', 3, .5));
ef_6 = my_extractFeatures_b(I6, fp6, fspecial('gaussian', 3, .5));
```

An explanation of the performance differences between the two different methods are described in the following section.

Part 3 - Feature Matching

In order to perform feature matching, the euclidean distance between each feature vector needed to be computed. In MATLAB, this is as simple as calling the following function:

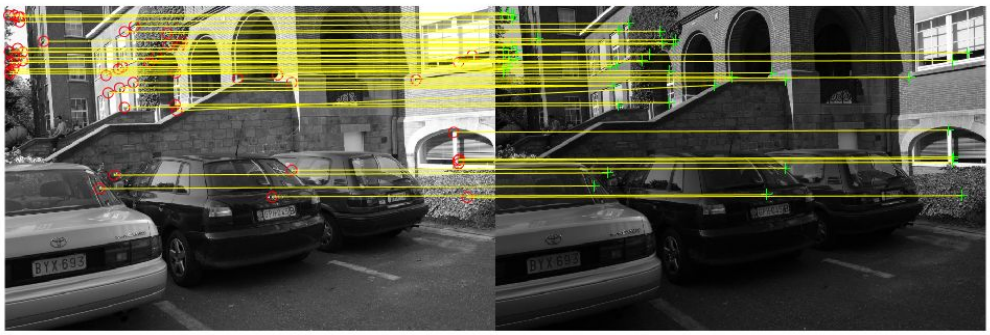
```
function D = fpDist(fp1, fp2)
    % Returns the Euclidean distance between the two vectors
    D = norm(fp2-fp1);
end
```

The function bestMatch() takes in the feature points and extracted features for two different images and returns two new lists of (x,y) coordinates for matched feature points from the two images. Best matches use the "ratio method" to determine if the match is valid - the ratio of the best match to the second best match is computed, and if the ratio is greater than some threshold then the pair is discarded as a match and "NaN" is recorded for that feature point pair. The implementation of the method is provided in the Appendix of the report. The results of the feature matching are shown below in the following figures.

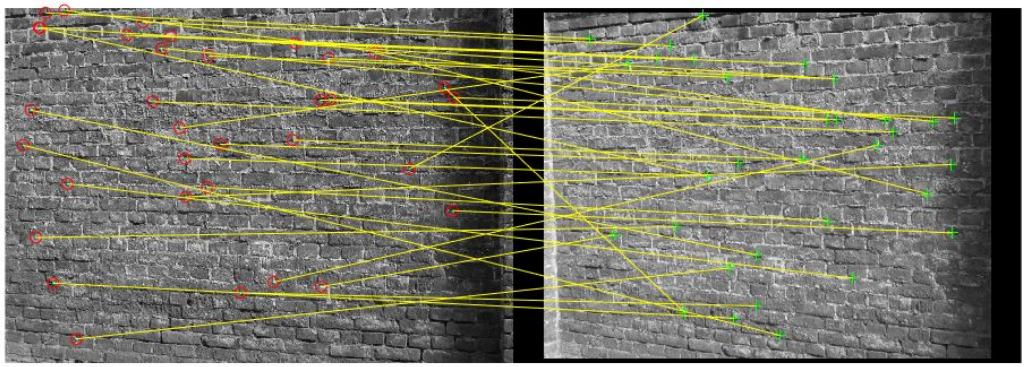
Feature matching using neighboring pixel feature descriptor
Matching ratio threshold: 0.75

Feature matching using SIFT-like feature descriptor
Matching ratio threshold: 0.5

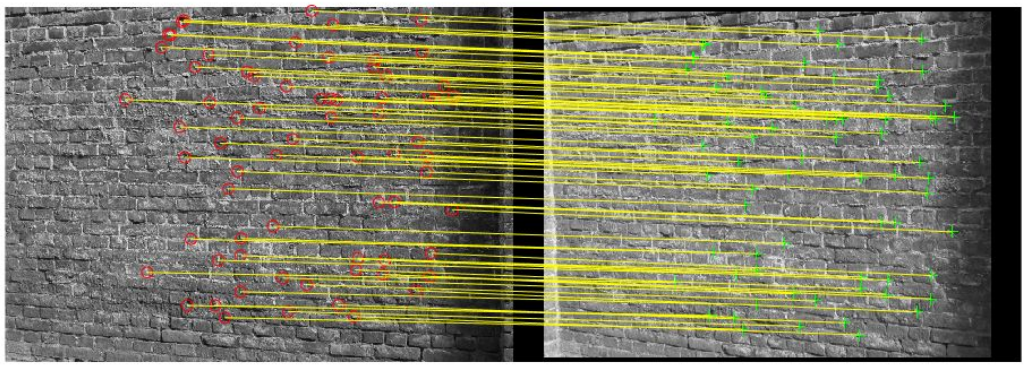
Feature matching using neighboring pixel feature descriptor
Matching ratio threshold: 0.75



Feature matching using SIFT-like feature descriptor
Matching ratio threshold: 0.6



Feature matching using neighboring pixel feature descriptor
Matching ratio threshold: 0.8



Feature matching using SIFT-like feature descriptor
Matching ratio threshold: 0.6

As evident from the figures, the performance of the SIFT-like feature descriptor performs significantly better than the local-neighborhood raw-pixel-value method. This is because the local-neighborhood method compares the immediate raw-pixel values around each feature point, and in the corresponding image the pixel values may be higher or lower or skewed if the second image is brighter, lighter, or taken from a different angle. That said, the method should work great for pure translations, and from the bikes image it is evident that the method isn't completely useless. If the results of this method were refined using the RANSAC method then results would be much more accurate. The SIFT-like feature descriptor method, on the other hand, worked great for images of different brightness and from a different angle, because it uses normalized local changes in gradient to describe each feature point. Achieving the above results required manual tweaking of the sigma values used for image Gaussian smoothing for the SIFT-like feature detector, and manual tuning of the "ratio-test" threshold value used to determine if a match is valid.

Reflection

This lab reinforced the differences and uses for feature points and feature descriptors. The Harris Corner Detector was successful in identifying "corner" points, otherwise useful as "points of interest". These corner points could then be used to try to match different images. Two very similar images, say taken from a different angle or two parts of an image to be stitched together, could be matched using these feature points. To perform the matching, each feature point needed to be "described" using a feature descriptor technique. Each feature point has a specific feature descriptor associated with it. In this lab, we used two different methods to describe each feature point, one using raw neighboring pixel values and the other utilizing a "SIFT-like" gradient based technique. As expected, the gradient-based technique performed far superior to the raw-pixel value technique because differences in brightness and angle could be accounted for. It was interesting to see the profound effect that changing the gaussian filter size pre-SIFT-describing had on the success of matching - if the sigma value was too high there was a large number of false-matches. Accordingly, it was interesting seeing the different effects of changing the "best match ratio threshold" value - if this value was too low then there were a lot of false matches, and vice versa. In the end, the feature point matching was very successful using the SIFT-like technique, and the local-neighbor method could be bettered using a RANSAC-like technique to attain similar results.
- Alec Hardy

For me, this lab taught me how to manually find feature points using the Harris Corner Detector. Doing this without using the MATLAB functions let me truly see each and every step that is needed to do the Harris Corner Detector and understand how it works rather than just implementing it for the sake of the project. I was also responsible for part 2b where I learned what the importance of a good feature descriptor is. For part 2a, we used a bad/simple feature descriptor that didn't really feature match as well as we had hoped. Using a 128 dimensional feature detector gave much more accuracy to the feature matching since there was so much more information to base it off of. This method of finding feature points (what I learned in part 1) and feature descriptor(what I learned in part 2b) is foundational to being able to feature match effectively and feels like we are getting closer to learning what the max capabilities of computer vision are currently. I suspect that we will learn more about making guesses of image recognition when there are multiple images at different angles next to obtain information about a subject that can only be known using both images. Feature matching would allow us to identify the subject across the two images in this case and would be essential.
- Nikhil Patolia

Appendix A - MATLAB code

Script for Part 1

```
clc;
clear;
close all;

%These variables can be changed
sigma = 1;
hsize = 3;
alpha = 0.05;
R_threshold = 0.5;

%Choose between a gaussian filter and averaging filter by uncommenting
%either of the lines below:

weights = fspecial('gaussian', hsize, sigma);
%weights = [1/9,1/9,1/9; 1/9,1/9,1/9; 1/9,1/9,1/9];


checkerboard_image = checkerboard();
[xsize, ysize] = size(checkerboard_image);



prewitt_filter_x = [-1,0,1;-1,0,1;-1,0,1];
prewitt_filter_y = [-1,-1,-1; 0,0,0; 1,1,1];

%Apply the Horizontal and Vertical prewitt filters
Ix = imfilter(checkerboard_image, prewitt_filter_x);
Iy = imfilter(checkerboard_image, prewitt_filter_y);

%Compute the necessary components of the second moment matrix
Ix2 = imfilter(Ix.^2, weights);
IxIy = imfilter(Ix.*Iy , weights);
Iy2 = imfilter(Iy.^2, weights);

%Determinant of the second moment matrix
det_of_m = (Ix2.*Iy2) - IxIy.^2;

%Trace of the second moment matrix
trace_of_m = Ix2+Iy2;

%Calculating the R-score of the second moment matrix
R_score = det_of_m - (alpha * trace_of_m.^2);

%Applying a threshold to the R-score
thresholded = R_score > R_threshold;

%Getting the locations of white pixels in binary image
[x_locs, y_locs] = find(thresholded == 1);

%Padding the image with zeros to avoid crashing program
padded = padarray(R_score,[1 1],0,'both');

%Non maximum suppression
for i = 1:size(x_locs,1)
    neighbors = padded(x_locs(i):x_locs(i)+2, y_locs(i):y_locs(i)+2);
    max_of_neighbors = max(max(neighbors));
```

```
    if padded(x_locs(i) + 1, y_locs(i) + 1) ~= max_of_neighbors
        padded(x_locs(i) + 1, y_locs(i) + 1) = 0;
    end
end

%Removing the zero padding
harris_corner_detect = padded(2: size(padded, 1) - 1, 2: size(padded, 2) - 1);


BWC = bwconncomp(harris_corner_detect);
s_w = regionprops(BWC, 'Centroid', 'FilledArea');
centroids_w = cat(1,s_w.Centroid);

figure()
imshow(checkerboard_image);
for i = 1:size(centroids_w,1)
    C = centroids_w(i,:);
    rectangle('Position',[C(1) - 1,C(2) - 1,2,2],'FaceColor',[1 0 0])
end
```

Script for Part 2 and 3

```
%% Feature Description:
%  Write a function that extract the feature vector from the detected
%  keypoints. The function should have the format:
%  [extracted_features] = my_extractFeatures_a/b(image, detected_pts)
%  For this part, you can use MATLAB's built-in feature detectors
%  (such as FAST, SURF) to detect keypoints. Limit the number of detected
%  keypoints to no more than 100 per image.
%   a. First use raw pixel data in a small square window (say 5x5) around
%      the keypoint as the feature descriptor. This should work well when
%      the images you are comparing are related by only a translation.
%   b. Next, implement a SIFT-like feature descriptor. You do not need to
%      implement the full SIFT (for example no orientation normalization if
%      no rotation is involved).


clear;
close all;

%% Let's get started
I1 = rgb2gray(imread('bikes1.ppm'));
I2 = rgb2gray(imread('bikes2.ppm'));
I3 = rgb2gray(imread('cars1.ppm'));
I4 = rgb2gray(imread('cars2.ppm'));
I5 = rgb2gray(imread('wall1.ppm'));
I6 = rgb2gray(imread('wall2.ppm'));

%% Extract feature points
% Feature points in [x,y] coordinates
% The selectedStrongest(N) value is chosen manually such that there are not
% more than 100 FP matches after performing the bestMatch() function.
fp1 = selectStrongest(detectFASTFeatures(I1),200).Location;
fp2 = selectStrongest(detectFASTFeatures(I2),200).Location;
fp3 = selectStrongest(detectFASTFeatures(I3),200).Location;
fp4 = selectStrongest(detectFASTFeatures(I4),200).Location;
fp5 = selectStrongest(detectFASTFeatures(I5),400).Location;
```

```
fp6 = selectStrongest(detectFASTFeatures(I6),400).Location;

%% Extracted feature vectors
% ef(row_num) corresponds to the location in fp(row_num)

% Use SIFT-like feature extraction algorithm
ef_1 = my_extractFeatures_b(I1, fp1, fspecial('gaussian', 5, 1));
ef_2 = my_extractFeatures_b(I2, fp2, fspecial('gaussian', 5, 1));
ef_3 = my_extractFeatures_b(I3, fp3, fspecial('gaussian', 3, .5));
ef_4 = my_extractFeatures_b(I4, fp4, fspecial('gaussian', 3, .5));
ef_5 = my_extractFeatures_b(I5, fp5, fspecial('gaussian', 3, .5));
ef_6 = my_extractFeatures_b(I6, fp6, fspecial('gaussian', 3, .5));

% Use crappy neighboring raw-pixel values
ef_1b = my_extractFeatures_a(I1, fp1);
ef_2b = my_extractFeatures_a(I2, fp2);
ef_3b = my_extractFeatures_a(I3, fp3);
ef_4b = my_extractFeatures_a(I4, fp4);
ef_5b = my_extractFeatures_a(I5, fp5);
ef_6b = my_extractFeatures_a(I6, fp6);

%% Returns [fp1,fp2] ordered by matching points
% Using the basic neighboring raw-pixel descriptor
[fp_match_1b,fp_match_2b] = bestMatch(fp1, ef_1b, fp2, ef_2b, .75);
[fp_match_3b,fp_match_4b] = bestMatch(fp3, ef_3b, fp4, ef_4b, .75);
[fp_match_5b,fp_match_6b] = bestMatch(fp5, ef_5b, fp6, ef_6b, .8);

% Using SIFT-like descriptor
[fp_match_1,fp_match_2] = bestMatch(fp1, ef_1, fp2, ef_2, .5);
[fp_match_3,fp_match_4] = bestMatch(fp3, ef_3, fp4, ef_4, .6);
[fp_match_5,fp_match_6] = bestMatch(fp5, ef_5, fp6, ef_6, .6);

%% Display everything
figure();
set(gcf,'color','w');
subplot(2, 1, 1);
showMatchedFeatures(I1, I2, fp_match_1b, fp_match_2b, 'montage');
xlabel(["Feature matching using neighboring pixel feature descriptor","Matching ratio threshold: 0.75"]);
subplot(2, 1, 2);
showMatchedFeatures(I1, I2, fp_match_1, fp_match_2, 'montage');
xlabel(["Feature matching using SIFT-like feature descriptor","Matching ratio threshold: 0.5"]);

figure();
set(gcf,'color','w');
subplot(2, 1, 1);
showMatchedFeatures(I3, I4, fp_match_3b, fp_match_4b, 'montage');
xlabel(["Feature matching using neighboring pixel feature descriptor","Matching ratio threshold: 0.75"]);
subplot(2, 1, 2);
showMatchedFeatures(I3, I4, fp_match_3, fp_match_4, 'montage');
xlabel(["Feature matching using SIFT-like feature descriptor","Matching ratio threshold: 0.6"]);

figure();
set(gcf,'color','w');
subplot(2, 1, 1);
showMatchedFeatures(I5, I6, fp_match_5b, fp_match_6b, 'montage');
xlabel(["Feature matching using neighboring pixel feature descriptor","Matching ratio threshold: 0.8"]);
subplot(2, 1, 2);
showMatchedFeatures(I5, I6, fp_match_5, fp_match_6, 'montage');
xlabel(["Feature matching using SIFT-like feature descriptor","Matching ratio threshold: 0.6"]);

function [extracted_features] = my_extractFeatures_a(image, detected_pts)

    % Pre-allocate extracted_features
    extracted_features = zeros(size(detected_pts,1),25);
```

```
    % Iterate through each of the extracted features
    for i = 1:size(detected_pts)
        x = floor(detected_pts(i,1));
        y = floor(detected_pts(i,2));
        try
            window = image(y-2:y+2,x-2:x+2);
        catch
            % Feature is on an edge or corner so I can't grab the entire
            % 5x5 window of neighbors.  Let's use zero padding in this
            % wierd, convoluted way.  It works.
            window = zeros(5,5);
            for w_x = 1:5
                for w_y = 1:5
                    if x+(w_x-3)>2 && y+(w_y-3)>2 && ...
                            x+(w_x-3) < size(image,1) && ...
                            y+(w_y-3) < size(image,2)
                        window(w_x,w_y) = image(x+(w_x-3),y+(w_y-3));
                    end
                end
            end
        end
        % Okay, at this point we have our 5x5 window.  Phew.
        row = reshape(window,1,[]);
        extracted_features(i,:) = row;
    end
end


function [extracted_features] = my_extractFeatures_b(image, detected_points, gaussian_filt)

%padding the image to avoid errors when getting 16x16 grid
padded = padarray(image,[8 8],0,'both');

%Set up the vector that needs to be returned
extracted_features = zeros(size(detected_points,1), 128);

%For each detected point
for i = 1:size(detected_points, 1)
    %Get the x_loc and y_loc + adjust for the buffer
    x_loc = detected_points(i,2) + 8;
    y_loc = detected_points(i,1) + 8;

    %create a 16x16 grid and do a gaussian filter on it.
    temp = padded(x_loc - 7 : x_loc + 8, y_loc - 7: y_loc + 8);
    grid16x16 = imfilter(temp,gaussian_filt);

    %Get the gradient magnitude and direction of each pixel
    [Gmag, Gdir] = imgradient(grid16x16,'sobel');

    %round the values to the 8 directionals
    new_dir = wrapTo360(Gdir);
    round_values = [0,45,90,135,180,225,270,315,360];
    directionals = interp1(round_values, round_values, new_dir,'nearest');

    %Split the 16x16 grid into a 4x4 grid with a 4x4 in each
    dirs_in_cells = mat2cell(directionals, [4 4 4 4], [4 4 4 4]);
    mags_in_cells = mat2cell(Gmag, [4 4 4 4], [4 4 4 4]);

    % add the magnitudes of each direction in order, which gets you the feature vector
    for row = 1:4
        for col = 0:3
            default_pos = ((row - 1) * 32) + (col * 8);
            for each_pixel_dir = 1:16
```

```
                pixel_dir = dirs_in_cells{row + (col * 4)}(each_pixel_dir);

                if pixel_dir == 0 || pixel_dir == 360
                    pos = 1;
                elseif pixel_dir == 45
                    pos = 2;
                elseif pixel_dir == 90
                    pos = 3;
                elseif pixel_dir == 135
                    pos = 4;
                elseif pixel_dir == 180
                    pos = 5;
                elseif pixel_dir == 225
                    pos = 6;
                elseif pixel_dir == 270
                    pos = 7;
                else
                    pos = 8;
                end
                extracted_features(i, default_pos + pos) = ...
                    extracted_features(i, default_pos + pos) + ...
                    mags_in_cells{row + (col * 4)}(each_pixel_dir);
            end
        end
    end
end


function [fp_o1,fp_o2] = bestMatch(fp1, ef_1, fp2, ef_2, null_ratio)

    fp_o1 = fp1;
    fp_o2 = zeros(size(fp1,1),2);

    for i=1:size(ef_1,1)

        % Determine the distances for the current feature point in image 1
        % and compare against every feature point in image 2
        distances = zeros(min(size(fp1,1),size(fp2,1)),1);
        for j=1:size(ef_2,1)
            d = fpDist(ef_1(i,:),ef_2(j,:));
            distances(j) = d;
        end

        % See what the lowest and second lowest distances are
        [smallest_1, smallest_index] = min(distances);
        smallest_2 = min(setdiff(distances,smallest_1));
        ratio = smallest_1/smallest_2;
        % If we fail the ratio test, return NULL for this fp
        if ratio > null_ratio
            fp_o1(i,:) = [NaN,NaN];
            fp_o2(i,:) = [NaN,NaN];
            continue;
        end

        fp_o2(i,:) = fp2(smallest_index,:);
        fp_o1(i,:) = fp1(i,:);

    end

end

function D = fpDist(fp1, fp2)
    % Returns the Euclidean distance between the two vectors
    D = norm(fp2-fp1);
```

```
end
```