

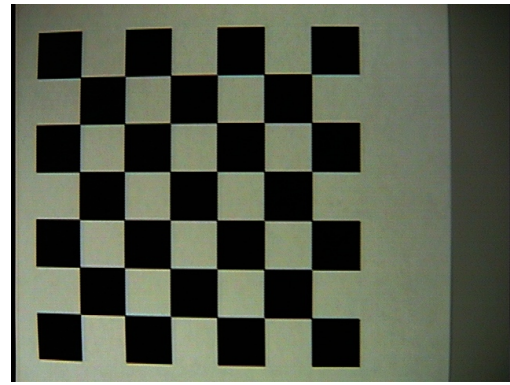
March 3, 2017

Point Features: Corner Detection

This report details corner detection of the algorithmic process, to varying results in change of variables, and to my personal proving-solving and debugging that went into this assignment. This assignment will refer to two images. One on the left will be referred to as *Building* and the one on the right will be referred to as *Checkerboard* further along in the text.



Building



Checkerboard

These images are both distinct in features, particularly with the amount of corners and with their discrete intensity in comparison to the background. To demonstrate concrete detection, it is very helpful to have images of this specific composition in order to easily apply algorithms. Both images appear to be black and white, but *Checkerboard* in actuality is composed of 3D matrix, because of the RGB color composition. When inserting these images to be tested, I had to make sure that the *Checkerboard* was converted to a gray scale image. This can be done in a photo editing application or in Matlab. I chose to simply use one of the Matlab's built-in functions: `rgb2gray(Image)`.

The Corner Detection algorithm will be described in four parts of this report: Computing Image Gradient, Determining Eigen Values from Matrix C, Sorting a List of Lambda 2 Values, and Creating Exclusive Neighborhood for each point P

Corners are included in the category of point features in an image. Corners can be easily described in mathematical terms, which are captured in patterns of intensity. Corners in this program will be depicted by white pixels. Through this report I've screenshot zoomed in parts of the image for the quality in the image are low and hard to see the detected corners because of that.

I. Computing Image Gradient

Before computing the image gradients, it is important to filter the image of any noise, or ‘smooth’ the image. Like always, we assume that the noise is Gaussian. Gaussian smoothing is important when applying edge detection, or point features algorithms. Without it then the data that you collect from the non-filtered image will be messy and inaccurate to the true values of the image. When creating the Gaussian mask, there are specific values to change to optimize the smoothing of the image, and in turn, optimizing the corner detection. The images below are zoomed in versions of *Building* with a 5x5 neighborhood, and a 70000 threshold on λ^2 . Specific sigma values are described below specifically.

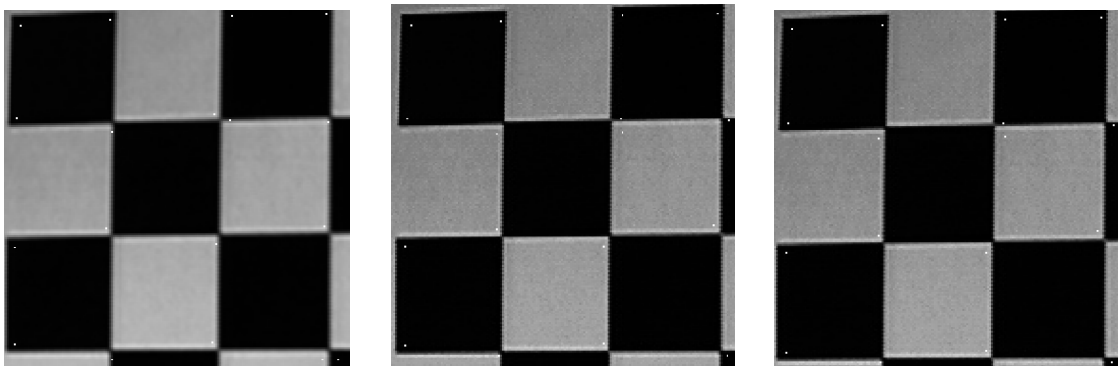


sigma = 0.5

sigma = 1.5

sigma = 2.5

As you can see there is little to no difference when there is a change in sigma value. The same concept is applied to the image *Checkerboard*.



sigma = 0.5

sigma = 1.5

sigma = 2.5

Once the image is ‘smoothed’, the E_x and E_y gradients are calculated, which is the same concept of the J_x and J_y gradients calculated in the Canny Edge Detector algorithm. These values support that in a region there is a corner when there is a strong gradient in two distinct directions. In layman terms, a corner is met by two strong edges. With these E_x and E_y gradients, it is able to obtain the C matrix for each pixel in the image. The C matrix captures corner structures in patterns of intensities where the sums are taken over the neighborhood Q of a general image point p . The matrix C characterizes the structure of the grey levels. The matrix C is symmetric, and thus can be diagonalized by a rotation

of the coordinate axes. We can think of C as a diagonal matrix:

$$C = \begin{bmatrix} \sum E_x^2 & \sum E_x E_y \\ \sum E_x E_y & \sum E_y^2 \end{bmatrix} \quad C = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}.$$

There is a three-step assumption for this process. **(1) Consider a perfectly uniform neighborhood Q**, where the image gradient vanishes everywhere, and so C becomes the null matrix. **(2) Assume that the neighborhood Q contains an ideal black and white step edge**, where lambda 2 is equal 0, and lambda 1 is greater than 0. **(3) Assume that the neighborhood A contains the corner of a black square against a white background**, and so the larger the eigenvalues, the stronger (higher contrast) their corresponding image lines.

II. Determining Eigen Values from Matrix C

When determining eigen values, from the matrix C, you must first determine the threshold. When the threshold is determined, the lambda 2 values with their points are kept if they are larger than the lambda 2. A corner is identified by two strong edges; therefore a corner is a location where the smaller eigenvalue, lambda 2, is large enough.

```
for r=row-n_val:row+n_val
    for c=col-n_val:col+n_val
        mat_C(1,1) = mat_C(1,1) + e_x(r,c)*e_x(r,c);
        mat_C(1,2) = mat_C(1,2) + e_x(r,c)*e_y(r,c);
        mat_C(2,1) = mat_C(2,1) + e_x(r,c)*e_y(r,c);
        mat_C(2,2) = mat_C(2,2) + e_y(r,c)*e_y(r,c);
    end
end
e = eig(mat_C);

lambda = e(2);
%check smaller eig value
if (e(2)>e(1))
    lambda = e(1);
end
```

a higher time complexity throughout this whole code. This part of the code takes the longest to excuse because of the nested for loop in the nested for loop.

In the code featured in the left, matrix C is calculated for the neighborhood (2N*1)x(2N*1). The built-in function *eigen* takes in this matrix as a parameter and returns two lambda values. The smaller lambda value is taken to compare against the threshold later in the code. This part of the code is within a function that is called within a nested for loop, resulting in

III. Sorting a List of Lambda 2 Values

When determining that the lambda 2 value is greater than the determined threshold, the point corresponding to this eigen value will be saved into a list. I created a three-column matrix. The first column holds all the x-coordinate values, the second column holds all the y-coordinate matrix, and the third column is the eigen value corresponding to the point. Once the all the possible points in the image is iterated

through, with all the matrix C neighborhood computations and the eigen values, it is time to sort the saved list in the end. When sorting this matrix, there are several possibilities that ran through my head. I first thought that I would iterate through the matrix row by row and do a simple bubble sort. This was very hard to execute because it was hard to swap all values of three columns in the matrix to be swapped with another three columns in the matrix. I later found that I could use the built-in function *sortrows()*.

IV. Creating Exclusive Neighborhood for each point P

In this part of the algorithm, it is very similar to the non-maximum suppression in the canny edge detector. It is desirable to get rid of points that are marking the same corner, and to only store the strongest point for each corner. For each point in the list the algorithm removes in which another point occurs in the neighborhood of another point with a higher eigen value.

```

flag = 0;
new_r = 1;
flag_mat = zeros(size(sorted_L));
[L_rows, ~] = size(sorted_L);

for r=1:L_rows-1
    x_hi = sorted_L(r,1) + N;
    x_lo = sorted_L(r,1) - N;
    y_hi = sorted_L(r,2) + N;
    y_lo = sorted_L(r,2) - N;

    flag = flag_mat(r,3); %flag = 1 when there is redundancy

    if(~flag)
        new_L(new_r,1) = sorted_L(r,1);
        new_L(new_r,2) = sorted_L(r,2);
        new_L(new_r,3) = sorted_L(r,3);
        new_r = new_r+1;
    end


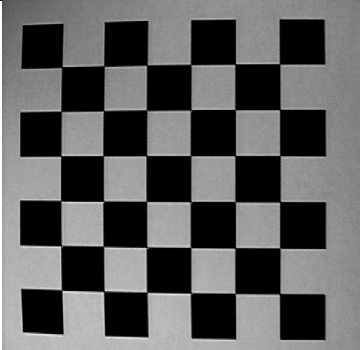

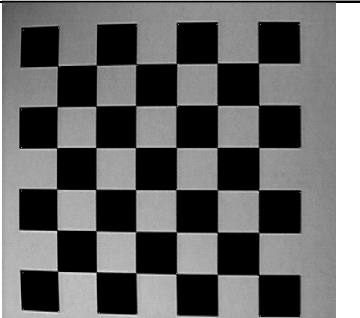

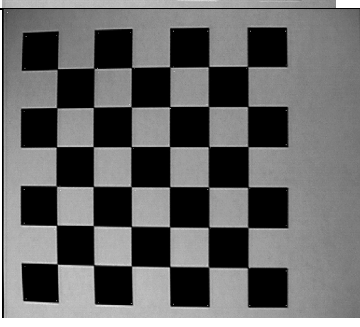
    for a=r+1:L_rows
        if(sorted_L(a,1) <= x_hi && sorted_L(a,1) >= x_lo)
            if(sorted_L(a,2)<= y_hi && sorted_L(a,2) >= y_lo)
                flag_mat(a,3) = 1;
            end
        end
    end
end
end

```

This algorithm implementation uses a flag to determine what rows to skip when iterating through the list and adding to a new list (new_L in Matlab) of the ‘real’ and ‘true’ corners of the image.

Results Varying Parameters

There are three degrees of freedom in the corner detector: the width of the smooth gradient filter (earlier in this report, I determined the results varying on the sigma value with the Gaussian mask), the size of the neighborhood, and the lower threshold for accepting something as a corner.

Sigma = 2.5, threshold = 70000	<i>Building</i>	<i>Checkerboard</i>
N = 1 Neighborhood = 3x3		
N = 3 Neighborhood = 7x7		
N = 5 Neighborhood = 11x11		

Above is a table that shows the neighborhood varying. The corner detection improves with the increased neighborhood size. It is very hard to physically see this for *Checkerboard*, but it is true.

Increasing the threshold made the corners more accurate to the true corner value.