Introduction to Computer Vision

Corner Matching Using bidirectional normalized cross correlation

-Yashwanth Telekula

Summary:

In this assignment, we are trying to find the feature extractions and feature matching points by using extending our work on Harris corner detection that we have done in the assignment-2. We find the corner pixels in both images independently and obtain the edges that are similar to each other using the method of bidirectional normalized cross correlation. Then we mark both the edge corners using a straight line.

In order to run the code, run the maincode.m file and manipulate the image reading lines in the maincode.m file to test the project on different images.

- ⇨ If Bi-directional matching was not used, a lot of non-related corners are matched which lead to decrease in efficiency.
- ⇨ During bi-directional matching, to avoid more edge matching we assumed both the images have similar y coordinate sizes and the pixel values are saved only if the Y values are close to each other. This lead to achieve more accurate points.
- ⇨ During the code running if the image sizes are small like 500X500 it takes little time, but if the image size is big like 5000X4000 ie an image taken from a high end mobile, it takes a long time.

Code Explanation:

- Maincode.m
- detectHarriscorners.m
- imagesmooth.m
- maxvalues.m
- erase.m
- compareweights.m
- find_weight.m
- markcorners.m
- `showMatchedFeatures`

1. **Maincode:**

   This is the main function that has to be run to get the matched edge corners that seem similar . In this file we read two images, assign the S,N,D,M values and call the detectHarrisCorners function independently for both images. This function returns R and corner matrices. The corner matrix is used later to find the

position of edges and the weight of each edge for each corner matrix. The values of S,N,D,M and image can be changed in this file to get variable results.

- ⇨ To reduce the confusion, I have taken a long image with big y axis and divided the image into two images with a considerable common area and used those two images to find the similar corners.
- ⇨ If we want to avoid this method we can run an alternate main code(**Alternatemaincode.m)** where the code takes two input images independently and does the same process afterwards.

2. **DetectHarrisCorners:**

This function takes the S,N,D,M and image(2-D matrix) values as inputs and returns the 'corners' and 'R' value matrix. This is the key function. This function calls a lot of functions like imagesmooth, findgradientimage etc. in order to smooth the image, find the gradient image, compute Harris corner 'R' values etc.

```
function [corners, R] = detectHarrisCorners(image, S, N ,D, M)
```

3. **Markcorners:**

This function takes the image matrix and corner matrix where the corner matrix contains the coordinates of pixels which are edges. This function saves the all the edge coordinates in a new weightedmatrix and calls another function find_weight to get the weight of its pixel by comparing with its neighboring pixels. The weight is also saved in weightedmatrix and the weightedmatrix matrix is returned.

- ⇨ This function is called twice independently for both the images to get the weights of both the image edge pixels.

```
function [matrix ] = markcorners( corners ,im )
```

4. **Find_weight:**

This function takes the image matrix and the x,y co ordinates of a edge pixel and returns the weight of that pixel. In this function the center (x,y) pixel intensity is compared with its neighboring 8 pixels and a weight- value(8 bit) between 0 to 255 is obtained. This weight is calculated just as said in the class.

```
function [ value ] = find_weight( R,i,j )
```

5. **Compare-weights:**

This function takes the both image edge weight matrices as input variables and does the following. First it compares each weight of one image edge pixel with all the other weights in second image edge pixel weights and finds the best match for each edge pixel of image1. Then it does the same for the image2 edge pixels.

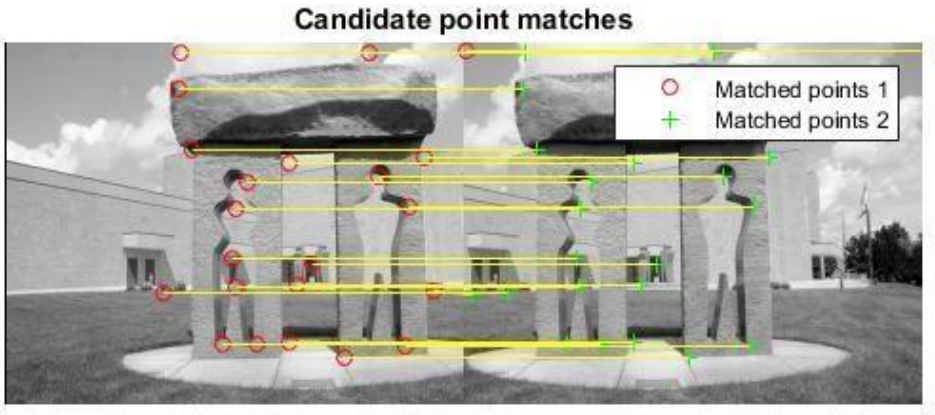
In the step 2, it takes each pixel preference and compares that preference's preference to get the best match. If they are best match it saves the both x,y co ordinates ie x,y co ordinates of left image and right image.

In the end this function returns the best matched pixel co ordinates of both images as the outputs.

⇨ To improve the efficiency of finding the best matched pixels, we assume that the images are almost similar in the size and only if the Y- coordinates are close to each other by a certain value then we save the pixel co ordinates of both images. This helps in avoiding cross matchings.

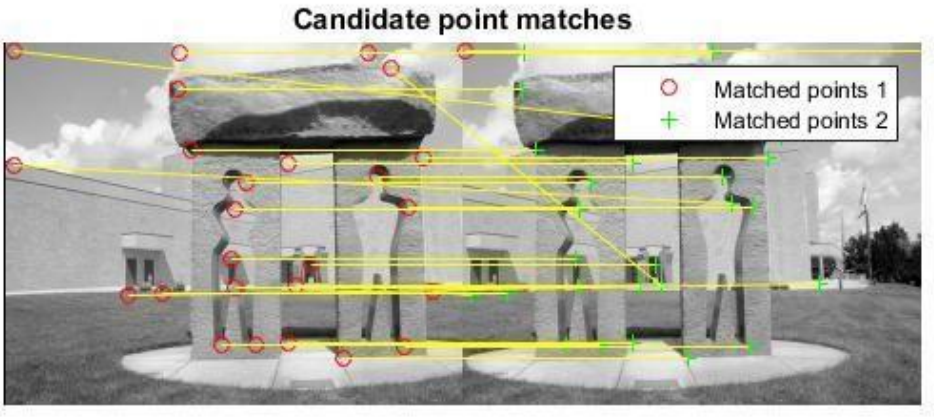
```
function [MatchPointsInI1, MatchPointsInI2  ] = compareweights( weightmatrix1,
weightmatrix2 )
```

IMAGE WITH Y-DIRECTION RESTRICTION OF 5 PIXELS.



⇨ The cross-line matches are avoided by using Y-direction Restriction.

IMAGE WITHOUT Y-DIRECTIONAL RESTRICTION



6. **image_smooth:**

This function takes the 2-D matrix image and S(sigma) value as the inputs and smoothes the image by calculating the Gaussian matrix and returns the smoothed image.

```
function [ image ] = imagesmooth( image, S )
```

7. **Maxvalues:**

This function takes the R value matrix and tries to find the M maximum value points from the matrix. For every iteration after finding the max value, this function calls the erase function.

```
function [ corners ] = maxvalues( R,D,M )
```

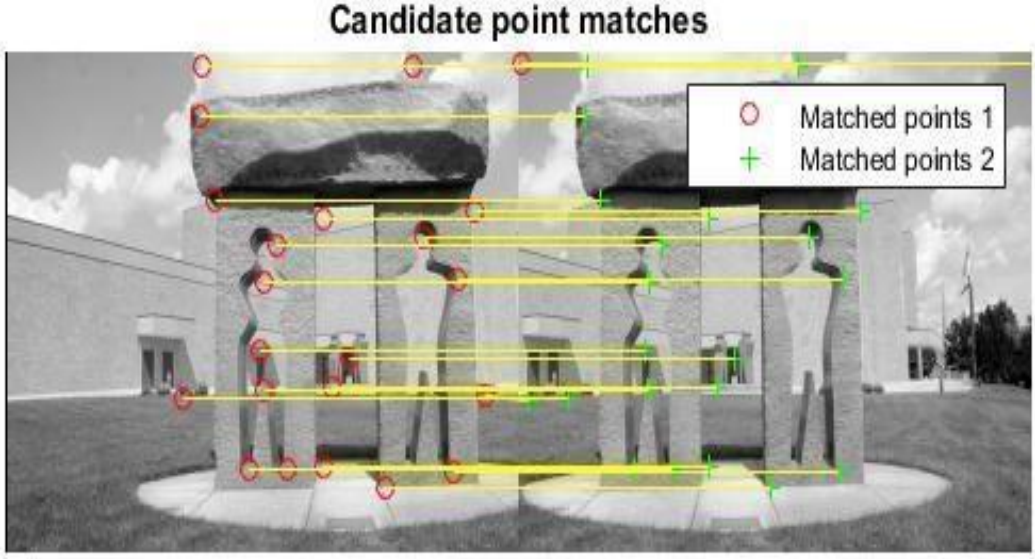
8. **Erase:**

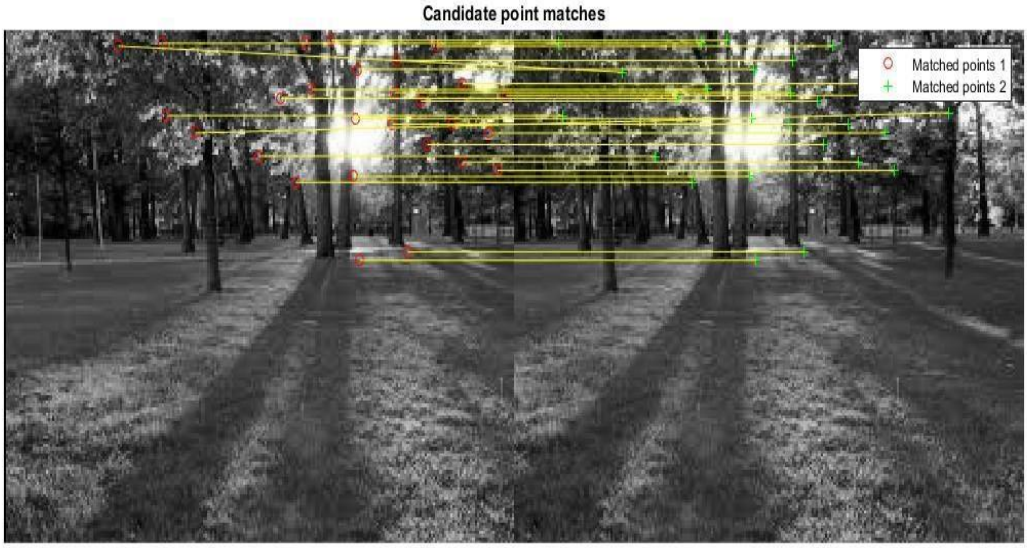
This function takes a co-ordinate value and makes all the values around D-radius region around it as minus infinity.

```
function [ R ] = erase( R,x,y,D )
```

Project Outputs:

Output image of bi directional matching pixels image.


Candidate point matches
Matched points 1
Matched points 2


Candidate point matches
Matched points 1
Matched points 2