Final Project
Digital Signal Processing
Dr. Arash Amini

Mohammad Hossein Shafizadegan
99104781

July 2, 2023

Contents

1 Implementing Direct correlation method

In this section we aim to complete the ”direct_correlator()” function. In this function we compute the normalized cross-correlation of an input image and a pattern using the direct formula of calculating correlation. Normalized cross-correlation is a measure of similarity between two images that is invariant to changes in intensity. In the code of this function we :

- Pad the input image with zeros on all sides by half of the pattern size using the ”padarray” function. This is done to avoid boundary effects when sliding the pattern over the image.

```
        extended_input = padarray(input_image_no_DC, floor(pattern_size / 2), 0, 'both');
```

- Compute the frobenius norm of the pattern using the norm function. This computation is done out side the for loop since the pattern is invariant and it remains unchanged in the process.

```
        pattern_norm = norm(pattern_no_DC, 'fro');
```

- In a two nested for loops we do the followings:
  - Extract a sub-matrix from the padded input image that has the same size as the pattern. The sub-matrix is centered at the current position of the pattern.

```
            sub_matrix = extended_input(row_ind:row_ind+pattern_size(1)-1,...
                                        col_ind:col_ind+pattern_size(2)-1);
```

  - Compute the dot product of the sub-matrix and the pattern using element-wise multiplication and summation. This is a scalar value that represents how well the sub-matrix and the pattern match.

```
            dot_product = sum(sub_matrix .* pattern_no_DC, 'all');
```

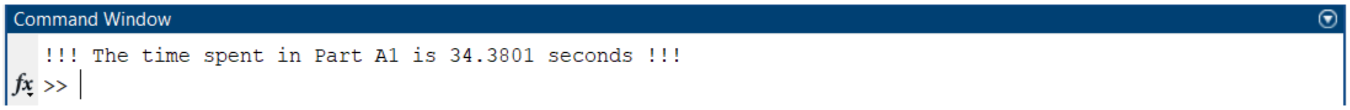
  - Compute the norm of the sub-matrix using the norm function.

```
            sub_matrix_norm = norm(sub_matrix(:));
```

  - Compute the normalized correlation coefficient by dividing the dot product by the product of the norms.

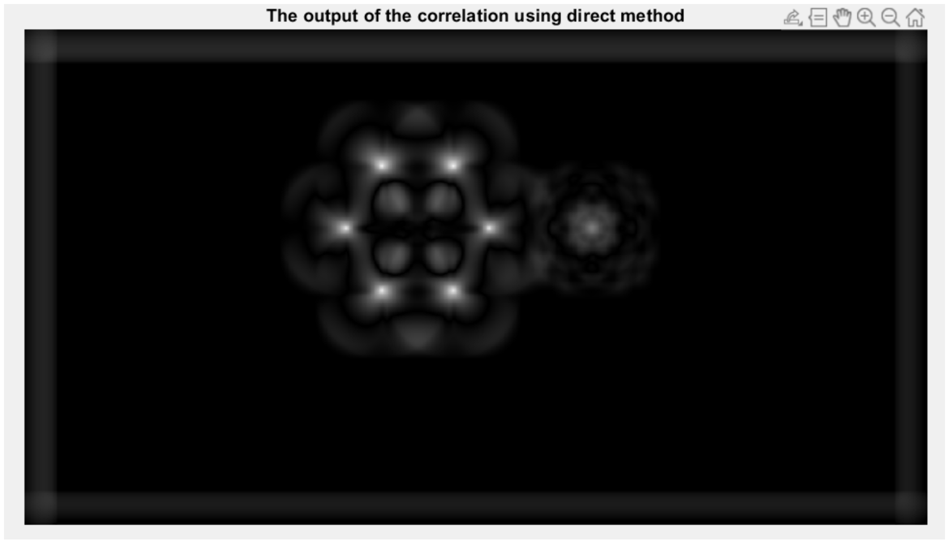
```
            normalized_correlation(row_ind,col_ind) = dot_product / (sub_matrix_norm *
                pattern_norm);
```

Now we set the value of the parameter ”Part_Enable” to ”A1” to run the code of this section.
In this section we want to find the big circles in the given image. The results are as follows :

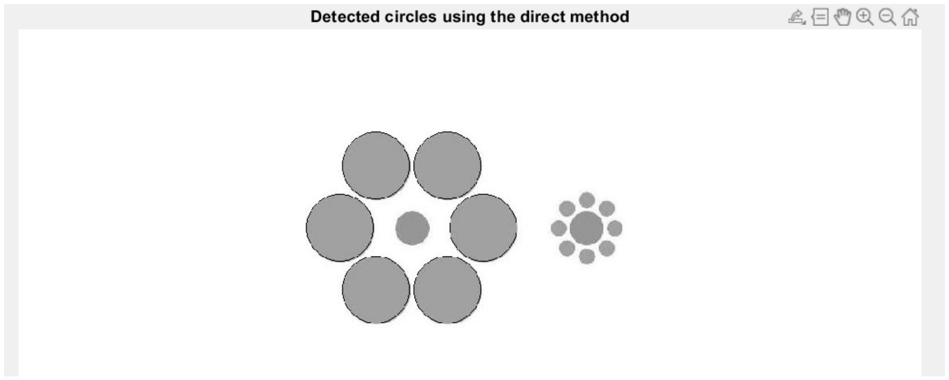


```
!!! The time spent in Part A1 is 34.3801 seconds !!!
fx >>
```

The output of the normalized correlation can be seen here:



The brighter the spots in the above figure, the more similar the spot (the location of the image) is with the given pattern. As we expected the six big circles on the left side which are similar to the given pattern, are the brightest spots in the figure.



Here it can be seen that the big circles have been detected and there is a black stroke for these circles.

2 Correlation using FFT

The special thing about this section is that we will compute the correlation using FFT and IFFT instead of using the direct method and direct formula. We know that computing correlation using FFT is a better and faster method in comparison to using direct formula of correlation because it reduces the computational complexity from O(N^2) to $O(N \log N)$, where N is the length of the data. This is because FFT can efficiently compute the convolution or cross-correlation of two signals by multiplying their frequency spectra, instead of summing over all possible lags in the time domain.

In order to complete the "fft2_correlator" function we do the followings:

- Flip the pattern horizontally and vertically to prepare for the convolution operation. In order for correlation to be the same as convolution, you must flip the pattern signal horizontally and vertically

```
        fliped_pattern = flip(flip(pattern_no_DC, 1), 2);
```

- Compute the 2D fast Fourier transform (FFT) of the input image and the flipped pattern, padding them with zeros to match the desired discrete Fourier transform (DFT) size.

```
        input_image_fft = fft2(input_image_no_DC, DFT_size(1), DFT_size(2));
        pattern_fft = fft2(fliped_pattern, DFT_size(1), DFT_size(2));
```

- Compute the cross-power spectrum, which is the element-wise product of the FFTs of the input image and the flipped pattern. Then we find inverse FFT of the cross-power spectrum to obtain the correlation result in the spatial domain.

```
        cross_power_spectrum = input_image_fft .* pattern_fft;
        correlation_result = ifft2(cross_power_spectrum, DFT_size(1), DFT_size(2));
```

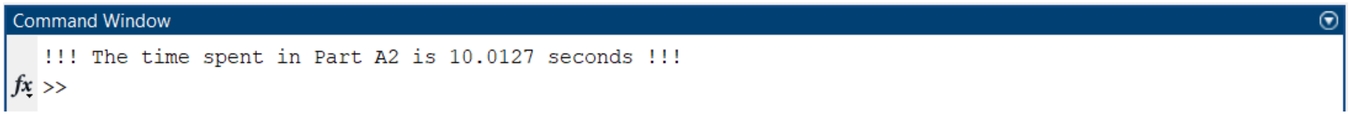
- Compute frobenius norm of the pattern the same as we do in the previous section.
- Compute a matrix that contains the Frobenius norm of each sub-image of the input image with the same size as the pattern. This can be achieved by finding the convolution of a all one matrix with the element-wise square of the input matrix. The convolution can be implemented using FFT and inverse FFT operations.

```
        temp = ones(pattern_size);
        temp_fft = fft2(temp, DFT_size(1), DFT_size(2));
        input_image2_fft = fft2(input_image_no_DC.^2, DFT_size(1), DFT_size(2));
        input_image_norm = sqrt(ifft2(input_image2_fft .* temp_fft, DFT_size(1), DFT_size(2)));
```

- Normalize the correlation by dividing the correlation result by the product of the input image norm and the pattern norm. We shouldn't forget to use the square root of the input image norm.
- Crop the normalized correlation matrix to match the size of the input image, removing the extra padding and selecting the correct result.

```
        % crop the correlation result to match the size of the input image
        padx = ceil(pattern_size(1)/2) - 1;
        pady = ceil(pattern_size(2)/2) - 1;
        normalized_correlation =
            normalized_correlation(padx:padx+input_size(1)-1,pady:pady+input_size(2)-1);
```

The results of this section are as follows :

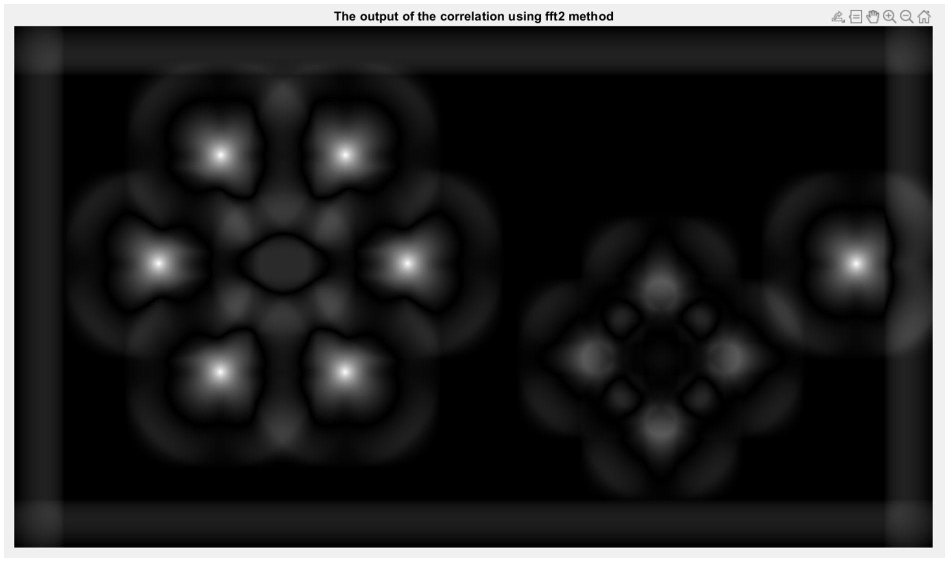


Command Window

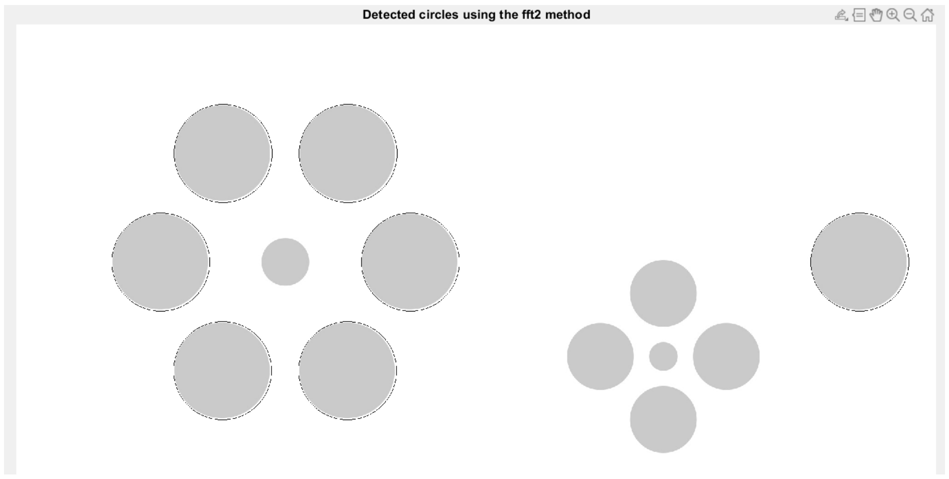
```
!!! The time spent in Part A2 is 10.0127 seconds !!!
>>
```

As we expected and explain before, this method is much faster and better optimized method for computing correlation and detecting the pattern in the given image.

The output of the normalized correlation can be seen here:



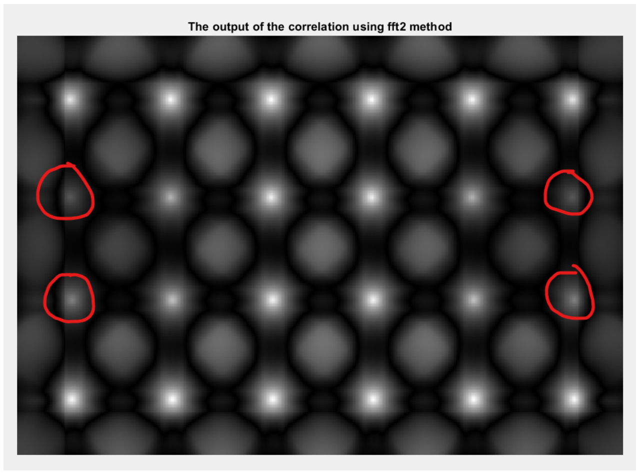
Based on the above figure, since we are looking for detecting big circles, the locations where these circles are located have become brighter.



Here it can be seen that the big circles have been detected and there is a black stroke for these circles.

3 Pattern detection in colorful images

Let's first check the results of the first method which is finding the normalized correlation without edge detection and without using gradients. Figure 1 and Figure 33 can be seen below:



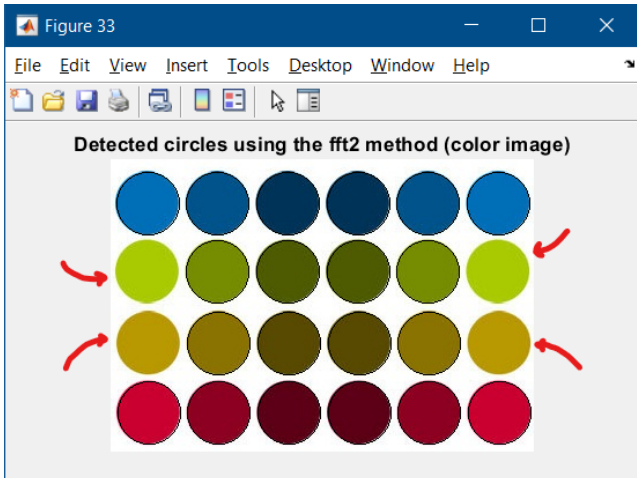
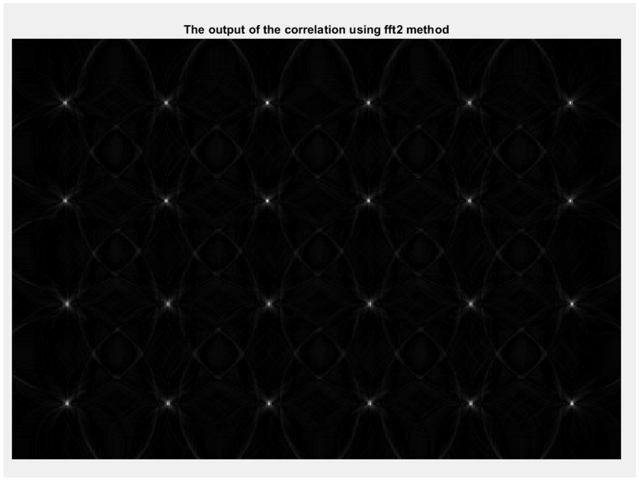


Figure 33
File Edit View Insert Tools Desktop Window Help
Detected circles using the fft2 method (color image)

The marked circles in the above figure have not been detected and there is no black stroke around them. Also the marked spots on the left side figure have not reached the threshold on the value of the correlation to detect a match so the circles located at these places have not been detected.

In the second method after extracting the edge of circles of the main image and the edge of the pattern circle, we try to find the correlation between the edges instead. The results will be as follows:



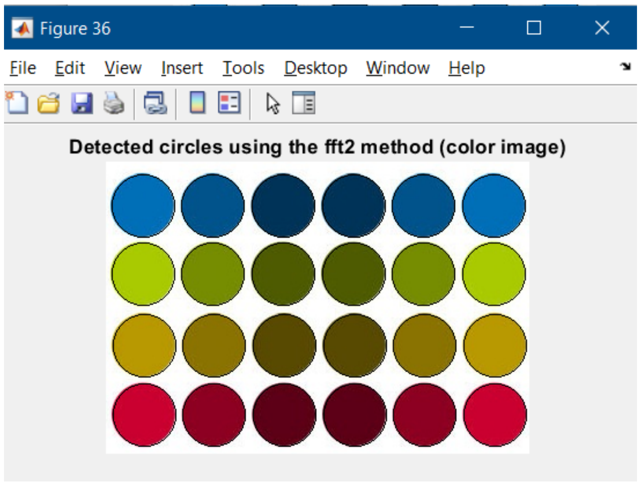
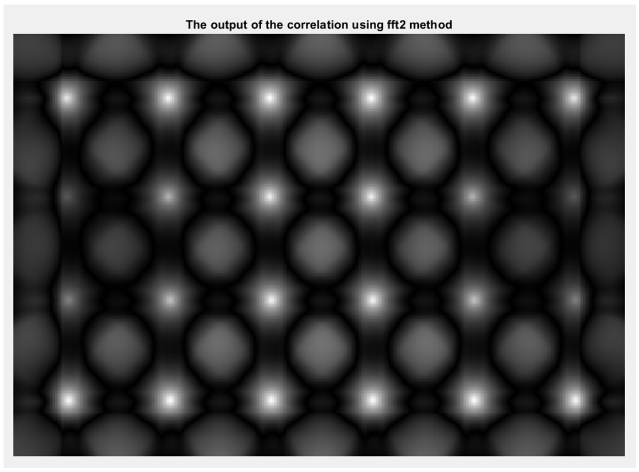


Figure 36
File Edit View Insert Tools Desktop Window Help
Detected circles using the fft2 method (color image)

Unlike the previous method, all the circles have been detected successfully.

Now we change the threshold_A3 to 0.4 and run the code again. Here are the result



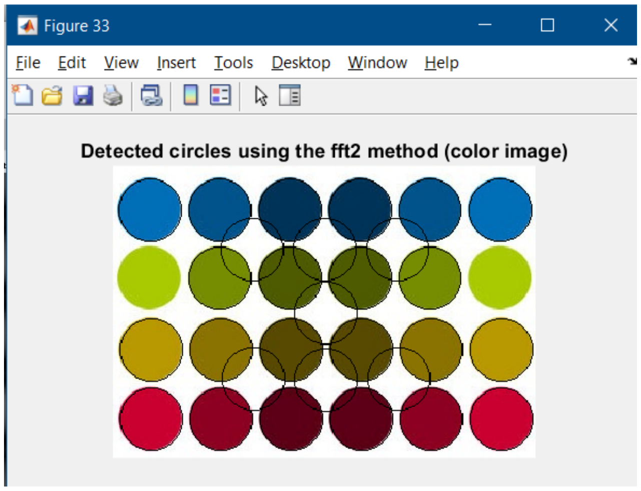


Figure 33
File Edit View Insert Tools Desktop Window Help
Detected circles using the fft2 method (color image)

Since we have reduced the threshold on the value of the correlation to detect a match, it can be seen that some other circles which actually don't exist, have been detected in wrong places around the center of the image.

4 Gradient pair vectors

By following these steps we try to complete the "circle_locator()" function :
In a a for loop that iterates over all the edge points except the last one, where edge_points is a matrix that contains the coordinates, gradient magnitude and gradient direction of each edge point. The main things to do are:

- get the current edge point as a column vector.

```
        current_point = edge_points(:,edge_points_ind);
```

- get the remaining edge points as a matrix.

```
            remaining_points = edge_points(:,edge_points_ind+1:end);
```

- Regarding the fact the points located on edge of a circle have same gradient magnitude and the fact that the gradient direction of the points we are looking for which are the ones that located on opposite side of a diameter, are 180 degree different, we apply some filters to find the similar points among the remaining ones, based on their gradient direction and magnitude. The degree_tolerance is a parameter that controls how much deviation is allowed in the gradient direction.

```
        idx_filt1 = abs(mod(remaining_points(4,:) - current_point(4) + 180, 360) - 360) <
            degree_tolerance;
        idx_filt2 = abs(mod(remaining_points(4,:) - current_point(4) + 180, 360)) < degree_tolerance;
        idx_filt3 = abs(remaining_points(3,:) - current_point(3)) < 10;
        similar_points = remaining_points(:,(idx_filt1 | idx_filt2) & idx_filt3 );
```

- In another for loop that iterates over all the similar points :
  - We compute the center of the circle that passes through the current point and the similar point. The center coordinates are rounded to integers.

```
        center_x = round((current_point(1) + similar_point(1))/2);
        center_y = round((current_point(2) + similar_point(2))/2);
```

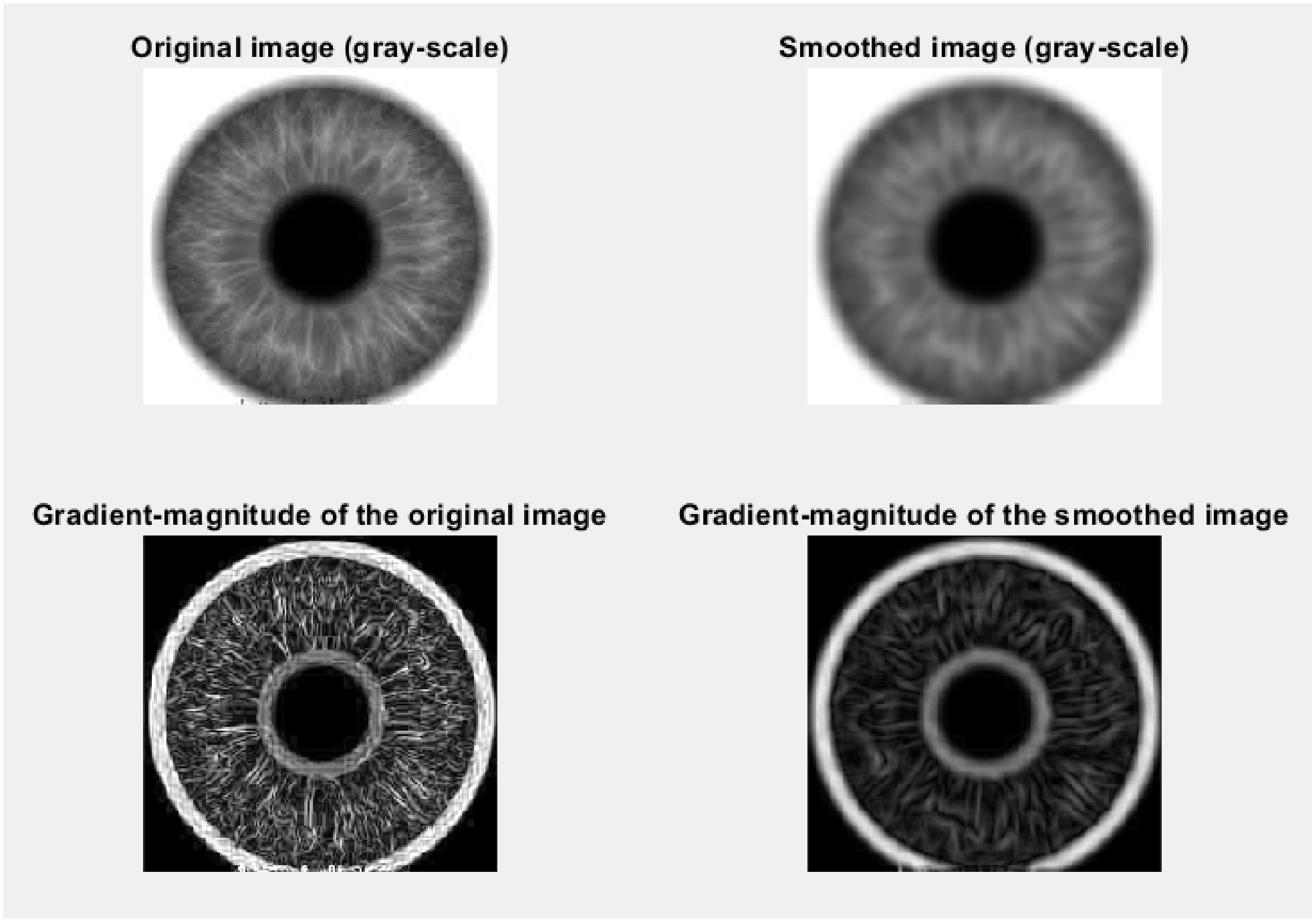
– compute the radius of the circle using the distance formula.

```
        radius = sqrt((current_point(1) - similar_point(1))^2 + (current_point(2) -
            similar_point(2))^2)/2;
```

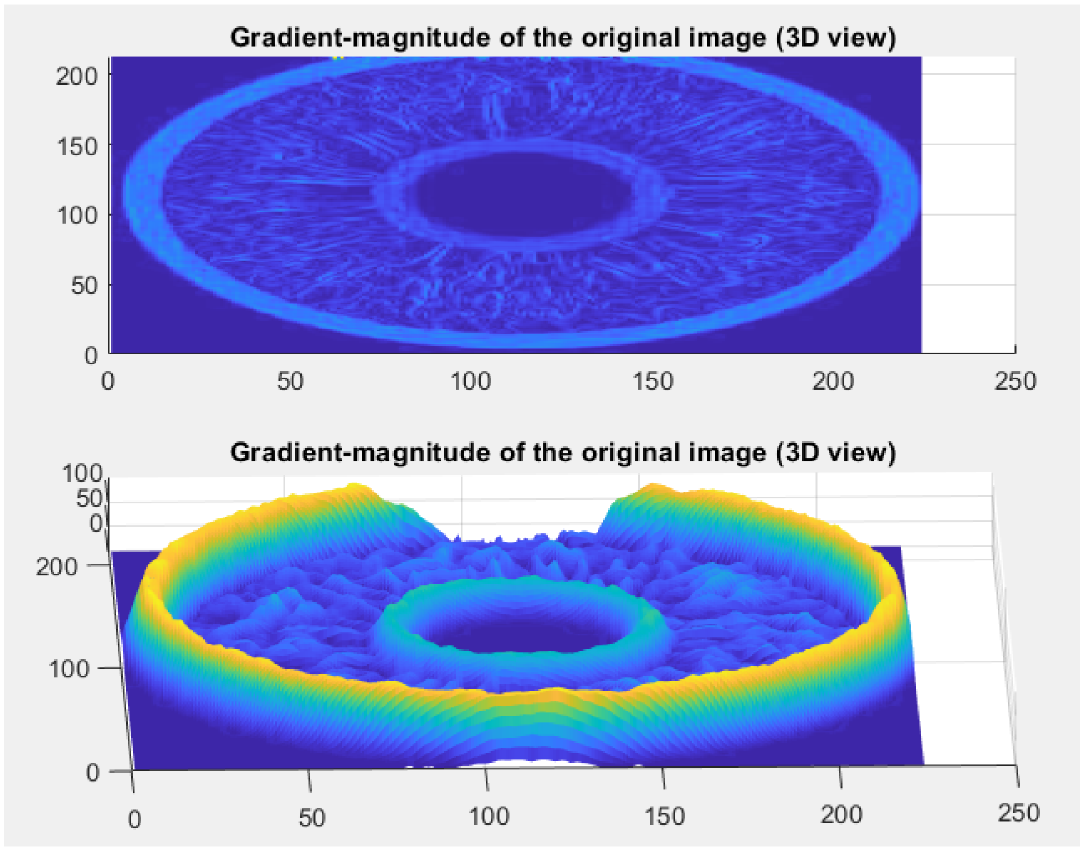
– update two output matrices with the center and radius information. The selection_counter matrix keeps track of how many circles have been detected with the same center, and the center_radius matrix stores the radius of each circle in a third dimension.

```
        selection_counter(center_x , center_y) = selection_counter(center_x , center_y) + 1;
        center_radius(center_x , center_y , selection_counter(center_x , center_y)) = radius;
```

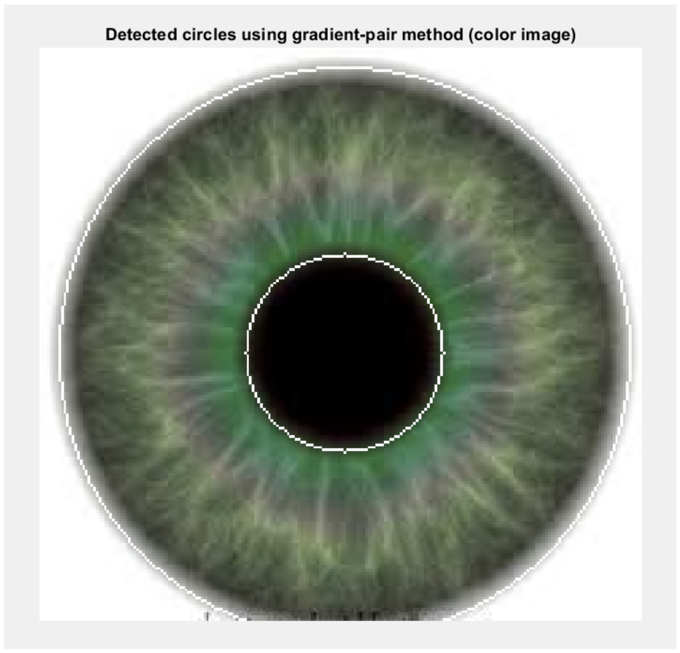
With ”repetition_Thresh” equal to 20 we run the code. The results can be seen here:


Original image (gray-scale)
Smoothed image (gray-scale)
Gradient-magnitude of the original image
Gradient-magnitude of the smoothed image

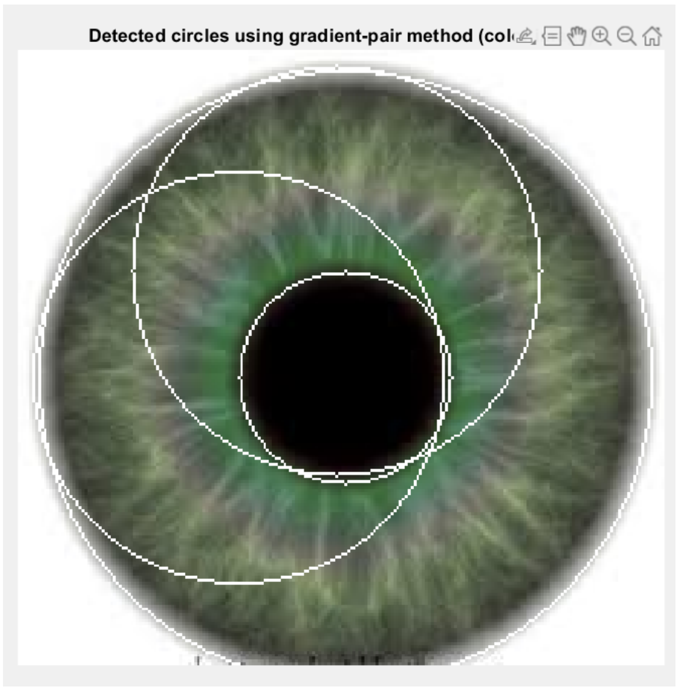
As we explained before, it can be seen that the two circle of pupil have approximately same gradient magnitude.



It can be seen that the two circles of pupil have been successfully detected and marked with white stroke.



Now we reduce the repetition threshold (”repetition_Thresh” = 10)and run the simulation again.



Since we have decreased the threshold, some other circles in the image with lower repetition have been detected too. These circles actually don't exist.