Hough transform, Sol Amara

This work is based on an article on the Hough transform written by Fatoumata Dembele.[1] The Hough transform is a method for finding imperfect instances of certain shapes within an image by pointing. It is based on algorithms for finding edges that turn a grayscale image into a binary image like the Canny or Sobel algorithms. By converting the image to binary form, only the edges of the shapes are visible, and this is the basis upon which the method is based.

Straight lines detections:

A straight line in general is defined using $y = mx + b$, but for vertical lines this equation is not uniquely defined, so the use here will be in the polar form: $y \cdot \sin(\theta) + x \cdot \cos(\theta) = r$. This equation describes a sinusoidal curve in the (r, θ) plane and the Hough transform uses this type of equation. For several points located on the same straight line, their sinusoidal curves will intersect at (r, θ) values suitable for describing this straight line equation, and by this property the straight lines can be found from the points located on the edge image.

The transformation works on this principle: it receives the binary image (that is, after using algorithms for finding edges such as Canny), calculates the curve r for each pixel whose value is 1. After that, the matrix H is initialized to zero so that each row represents a certain distance (r values) and each column an angle (θ values). A count is made for each value in the matrix H of the number of curves that pass through the point. The algorithm is based on the assumption that the more points there are on a straight line, the more curves will pass through the point with the corresponding (r,θ) values and so the score will be higher. The straight lines are determined according to the score each point in the H matrix has, where a high score indicates a high probability that it is more likely that a straight line appears in the image (because many points passing through it are shown in the image).

Figure 1 shows the results of Matlab's algorithm and my implementation on the gantrycrane image presented in the article. It can be seen that the results obtained are the same. In my implementation I used the resolution of theta and radius given as input to the function, and calculated as follows:

```
% The definition of the resolutions of variables:
T=-pi/2:Theta_resolution:pi/2-Theta_resolution;
M=length(BW(:,1)); N=length(BW(1,:));
max_R= ceil(sqrt((N-1)^2 + (M-1)^2));  % maximum distance: The
length of the diagonal according to Pythagoras
R =-max_R:R_resolution:max_R;
```

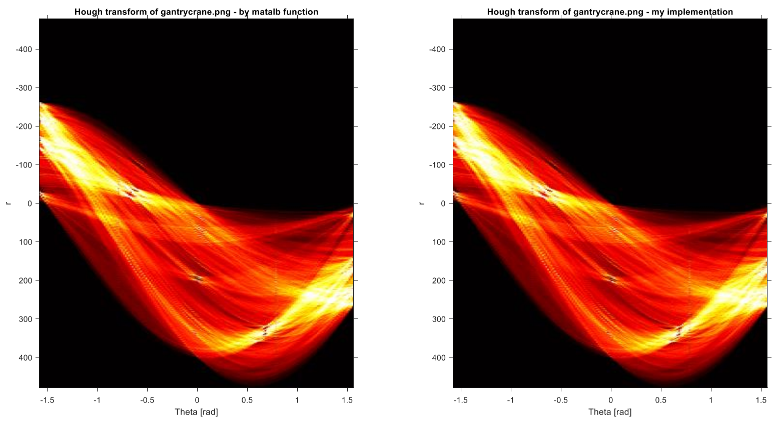

Figure 1: Algorithm implantation

Circular Hough transform:

The main idea is illustrated in Figure 2. A circle will be drawn around each of the edge pixels in an image after finding the edge (similar to the algorithms used to find straight lines). After that, we will increase the score for each point that the new circle we created passes through. When the radius of the circle in the sketch matches the radius of the circle in the image, the center of the circle will receive the highest score. All of the circles we've drawn pass through it so all of the points associated with the circles increase the score. In this algorithm, a three-dimensional array is built that contains the centers of the circles (a,b), and the variable radius. The formula for representing a circle is $r^2 = (x - a)^2 + (y - b)^2$ where each point on the circle is given by: $(x, y) = (a + r \cdot \cos(\theta), b + r \cdot \sin(\theta))$.

The function on page 10 of the article performs this transformation and selects the circuits from the resulting matrix. After defining a three-dimensional matrix R_{map} containing circles around the middle, it performs the scoring in the hough matrix by moving the corresponding circles in R_{map} (the first loop) so that their center is the corresponding point on the edge as illustrated in Figure 2. Then, it checks for every possible circle center if the grade received is greater than: 2πR·0.9·threshold . As a result of this test, the score increases by one for each pixel of the circle in the image. When it is contained completely, there are two times R, which equals the circumference of the circle. However, because we are dealing with discrete values a threshold of 90% is taken from these pixels. Also, the value is multiplied by an additional threshold chosen by the user and allows us to get a better result.

There are several problems with the function as presented in the article. A lot of circles are formed around the same center (or close to it) with different radii. A condition must be added to the test on the minimum distance between the centers of two identified circles. Second, the edges of the image are not taken into account, meaning checking that the results are within the image range in the first loop. By using this code on the coins image in matalb, for thresh=1 there is an identification of 28731 circuits. The time it took to perform this operation was 136.12 seconds. I performed tests for a diffrenets thresholds and although there is an improvement in the running time and I received fewer detections, the detections received do not correspond to the circles in the original image (with increasing the threshold, there are indeed fewer detections, but there are also no detections for certain coins). Since I would not receive the original coins if I changed the threshold, I chose to improve by using additional functions in addition to low thresholds.

Isclose: The purpose of this function is to check if there are circles that are too close. The user can enter a minDist value. For all circles whose center distances are smaller than this, only one circle is selected. The choice is made by taking the circle with the highest score in this group. If there are several circles with the same score, the circle is chosen randomly. This function improves the problem that occurred in the function result described in the article. During the test, circuits that have already been tested are deleted, which significantly improves the running time.

IsEmpty: This function checks whether a certain circle received a high score by chance and therefore passed the threshold in the article function. It uses 'dilation' with a circle on the edge image. This action creates an expansion of the boundaries of the edge and allows a wider range of error. After that, it creates a binary image with only this circle for each circle that is checked. Multiplication between the binary image and the expanded edge image finds the amount of pixels in which there is an overlap. Circuit thresholds are selected based on the thresh input value. The test is done using the condition: $lenngth(n) \geq 2\pi \cdot r \cdot thresh$ that is, if after multiplying the images the number of overlapping pixels is greater than $2\pi \cdot r \cdot thresh$ the circuit will select and if not delete. By entering a threshold value, the user can check whether this percentage of edges is present on the image.

As a result of the calculation time being large, I made another optimization of the algorithm. For finding circles in images, there is a problem of calculation cost. Performing the calculation when the radius is unknown can take a lot of time and therefore be ineffective. A solution to this situation is to determine the value of the radius you are looking for or the range in advance. This will reduce the number of calculations and complications. Another way is a different approach to the scores: instead of drawing a circle around each point, calculate the gradient of each point on the edge image. The gradient will give the line on which the center of the circle is located, but we will not know in which direction. Given the radius, it is possible to vote for only 2 points (instead of the entire circle) - points that are in the direction of the gradient and at the distance of the radius. Once we do this for a large number of points on a circle, the center of the circle will score higher compared to the other points. Figure 3 illustrates the concept visually.[2] [3] I completed the implementation and improvements I explained above. Figure 4 shows the results for the image of coins. From this figure it can be seen that both algorithms gave identical and accurate results. While for the algorithm presented in the article the calculation time was 136.12 [sec], for the second algorithm I presented with the calculation of the gradient the calculation time is 0.099 [sec] which means a significant optimization.

Additional problems that exist in the algorithm and suggestions for their solution:

• Continuity – the algorithm for finding straight lines draws the curves described in Figure 1 for each of the edge points. It gives a score according to the number of times the curve passes through a certain point of r and theta. There are situations in which several points are found in the image after appropriate processing (edge detection), and a straight line does not pass through them continuously, but the algorithm will detect them as a line. There are situations where this feature is helpful for the resulting image but there are situations where we would like more continuous lines. Therefore, in my opinion, the algorithm can be improved by adding an index given by the user on the level of continuity of the line. This will enable classification according to the desired case.[3]

• Discreteness - a transition occurs in the algorithm from continuous values calculated to discrete values. The resolution of the theta can greatly affect the results, so we will prefer high resolution. However, high resolution requires a larger amount of calculations and therefore can be inefficient. In addition, after the calculations, for example for circles, the values must be transferred to a separate table (when the calculation is continuous). The transition to the discrete table is done by rounding the results (since we are looking for the location of the centers of the circles which are natural numbers), therefore this can lead to an overdraw situation which means that a certain point is counted several times due to the rounding result. Additionally, the lack of values in the case of a large radius is another issue. The solution offered today is to round the results only after multiplying by the radius. Another solution is the Bresenham algorithm, which is an algorithm to create discrete circles that avoids the problems I mentioned.[3]

• Finding the circles/straight lines in the tables - after running the logarithm we got a table with a score for each point. There is currently no convention on how to convert this table into a decision. One way is to set a threshold such as a percentage of the returns. A second way is to check whether the height of the peak is equal to the number of pixels on the edge. However, it has problems because there may be situations where we want discontinuities or the shape is an ellipse.[3]

• A circle is uniquely defined by a center point and radius and can then be drawn on the image to visually see the identification. On the other hand, a straight line is also uniquely defined in this way, but once the length of the straight line and its starting point are not established, it is not possible to visually illustrate the line in the image. There are functions in Matlab that do this calculation.

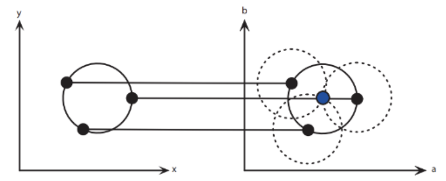

Figure 2: The idea of the algorithm visually

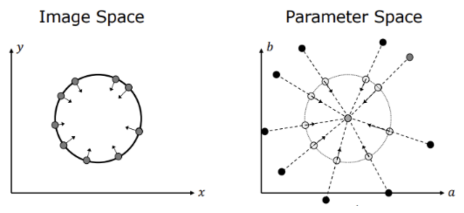

Figure 3: The idea of the gradient improvment

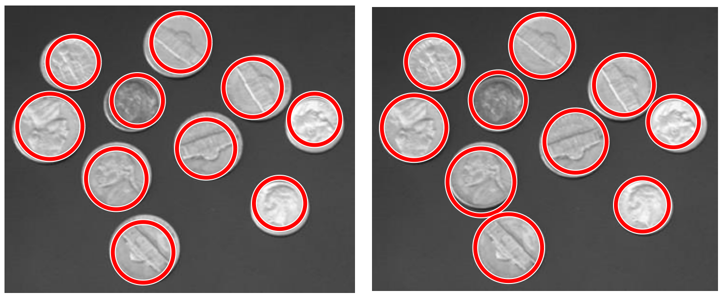
Figure 4: The result of the algorithm.
Right: the algorithm in the article. Left: Gradient algorithm

[1] F. Dembele, “Object Detection using Circular Hough Transform”.

[2] “Boundary Detection FPCV-2-2.pdf.” Accessed: Feb. 01, 2023. [Online]. Available: https://cave.cs.columbia.edu/Statics/monographs/Boundary%20Detection%20FPCV-2-2.pdf

[3] S. Just Kjeldgaard Pedersen, “Circular Hough Transform.” Aalborg University, Vision, Graphics, and Interactive Systems, Nov. 2007.