Report: Understanding and Implementing the Hough Transform

1. Introduction to the Hough Transform:

The Hough Transform is a powerful technique in computer vision and image processing used for detecting shapes, particularly lines, circles, and other parametric curves, in digital images. Developed by Paul Hough in the 1960s, the Hough Transform is widely employed in various applications, including object recognition, image analysis, and computer vision.

2. Basic Concepts of the Hough Transform:

2.1 Parameter Space:

The Hough Transform is a technique designed to detect shapes that can be represented by mathematical parameters. For the case of lines, the relevant parameters are the slope (m) and the y-intercept (b). The idea is to transform the Cartesian coordinates of points in an image into a parameter space where each possible line is represented by a unique curve.

3.2 Transforming Cartesian to Polar Coordinates:

When accumulating curves in the Hough Transform, each point in the Cartesian coordinate system (x,y) contributes to curves in parameter space defined by ρ and θ. The transformation from Cartesian to polar coordinates involves using the cos and sin functions.

Given a point (x,y), the parameters ρ and θ can be calculated using the following relationships:

$$\rho = x\cos(\theta) + y\sin(\theta)$$

$$\theta = \arctan\left(\frac{x}{y}\right)$$

In the Hough Accumulator, each point in the image votes for possible lines in parameter space. The cos and sin functions help map the Cartesian coordinates of a point to the ρ and θ values in polar coordinates.

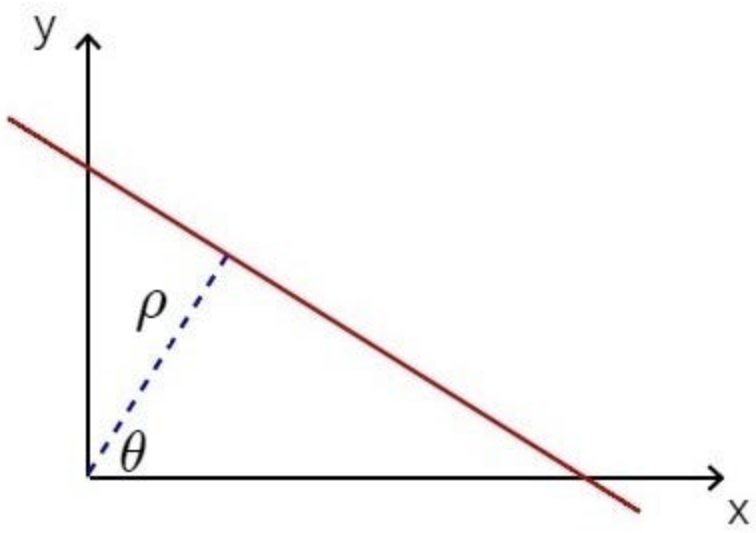
The trigonometric functions play a crucial role in the voting process, as they ensure that different lines passing through a point contribute to the correct regions in parameter space. The accumulation of these votes results in peaks in the parameter space, indicating the most likely parameters for lines that pass through a significant number of points.

Here I am providing more detatails to understand the transition form cartesian system to polar system which is very important in the development our code and for understanding the Hough Transform.

Straight lines representations

The straight line can be represented by two parameters (a, b) which correspond to slope and intercept. The line is then described as: $y = a\,x + b$

We can also describe the line using the pair (ρ, θ) in polar system. The first parameter, ρ, is the shortest distance from the origin to the line The second, θ, is the angle between x-axis and the distance line. One of the benefits of such representation is that we can describe vertical lines by ρ and θ which is impossible by using only (a, b) parameters in Cartesian system.

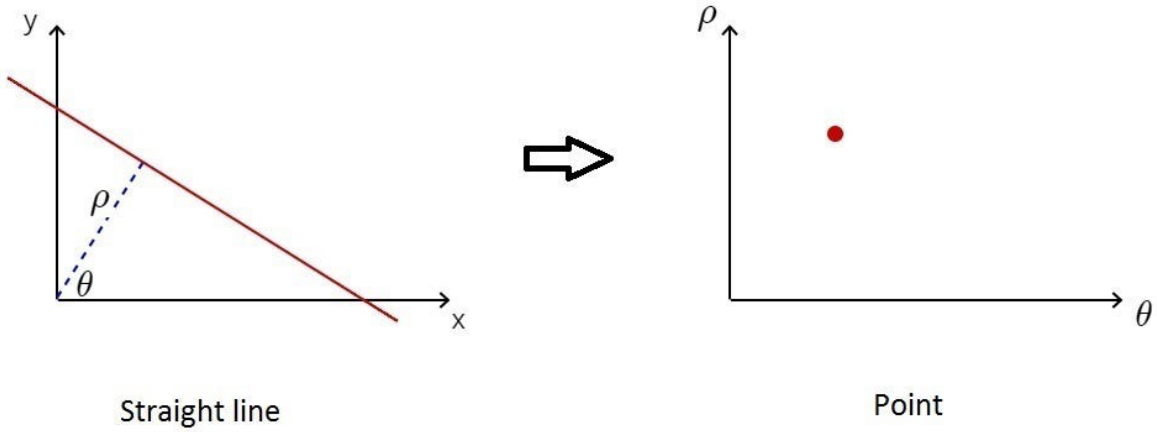


For a given line, we can determine specific ρ and θ. Then, the following equation is satisfied for each x, y point belonging to this line:

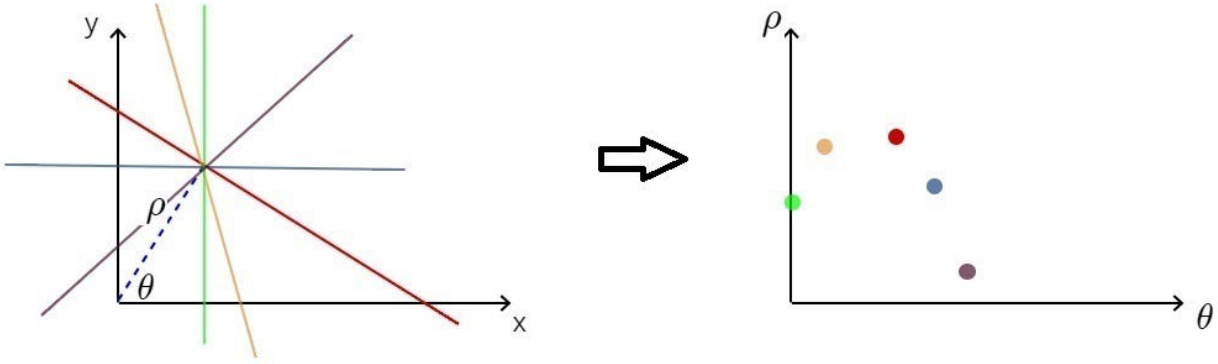
$$\rho = x \cos(\theta) + y \sin(\theta)$$

Mapping from Image space to Hough space

Let's draw a line on an image space again. As we already know, it is represented by some ρ and θ. So, we can draw such point in (ρ, θ) coordinates which is called a Hough space.



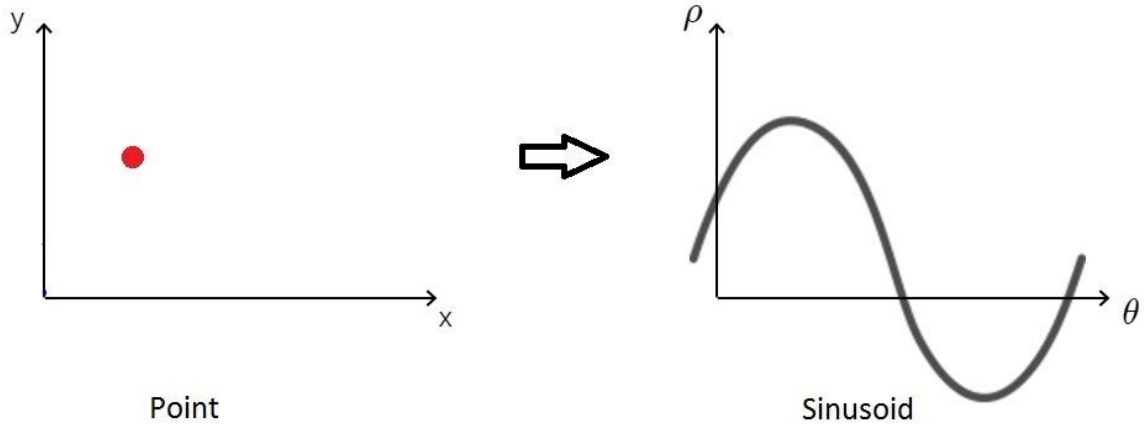
Now, in the image space, we are drawing other lines which are intersecting at one common point. Let's see what points will be produced in Hough space which are corresponding to these lines.



Bunch of lines intersecting at one point

Points which form a sinusoid

It turns out that these points in *(ρ, θ)* space are forming a sinusoid. Drawing infinite number of additional lines intersecting at this one point would result in a continuous sinusoid in Hough space. So, maybe, we can say that a point in image space results in a sinusoid in Hough space Let's recall the equation $\rho = x\cos(\theta) + y\sin(\theta)$. Indeed, for fixed (x, y) parameters representing point in image space and sliding through all possible values of θ in some range, we obtain ρ values which form a sinusoid.



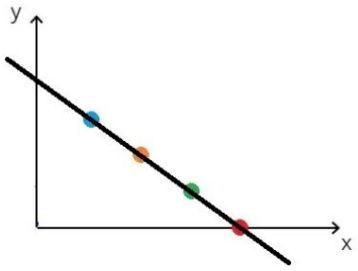
Point

Sinusoid

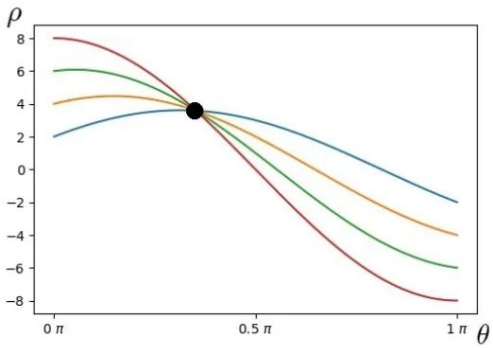
Finding Hough Lines

Finally, maybe the most interesting effect If we draw points which form a line in the image space, we will obtain a some of sinusoids in the Hough space. But, they are intersecting at exactly one point

the intersection point maps to the line that passes through all the points in the polar coordinate system



Points which form a line



sinusoids intersecting at one point

It means that, to identify candidates for being a straight line, we should seek for intersections in Hough space. And here comes the concept of the accumulator array.

Accumulator Array:

To detect patterns, an accumulator array is used to accumulate votes for possible parameters. For the case of lines, the parameters ρ and θ define each cell in the accumulator array. The array is discretized, with each cell corresponding to a unique combination of ρ and θ. The size and resolution of the accumulator array are determined by the range of ρ and θ values considered during the transformation.

Voting Process:

The voting process involves examining each point in the image and determining which curves in parameter space intersect with the parameters of the line passing through that point. For each point, the accumulator array is updated by adding votes to the corresponding cells. The goal is to identify cells in the accumulator array with high vote counts, as they indicate potential parameters for lines that pass through multiple points in the image.

Accumulation of Votes:

The accumulation of votes creates patterns in the accumulator array, and cells with high values represent potential lines or patterns in the image. Peaks in the accumulator array correspond to the parameters (ρ, θ) of the lines that are most likely to represent prominent features or edges in the image.

Intersection of Curves in Parameter Space:

The intersection of curves in parameter space occurs at the cells in the accumulator array with the highest vote counts. These intersections indicate the parameters that are most supported by the points in the image. For lines, the intersection of curves corresponds to the detected lines in the image.

Thresholding and Peak Detection:

To extract meaningful information, a thresholding step is often applied to the accumulator array. This helps identify significant peaks above a certain threshold, ignoring spurious or less prominent patterns. The detected peaks correspond to the parameters of the lines detected in the image.

The accumulator array is a crucial component of the Hough Transform, providing a space for accumulating votes and detecting patterns. It serves as a representation of

the parameter space, facilitating the identification of parameters corresponding to lines or other geometric shapes in the image.

Code Implementation and image analysis

Image Preprocessing:

The initial step involves loading the original image that serves as input to the Hough Transform. The choice of the image depends on the application, and it should contain the features or patterns you want to detect using the Hough Transform.

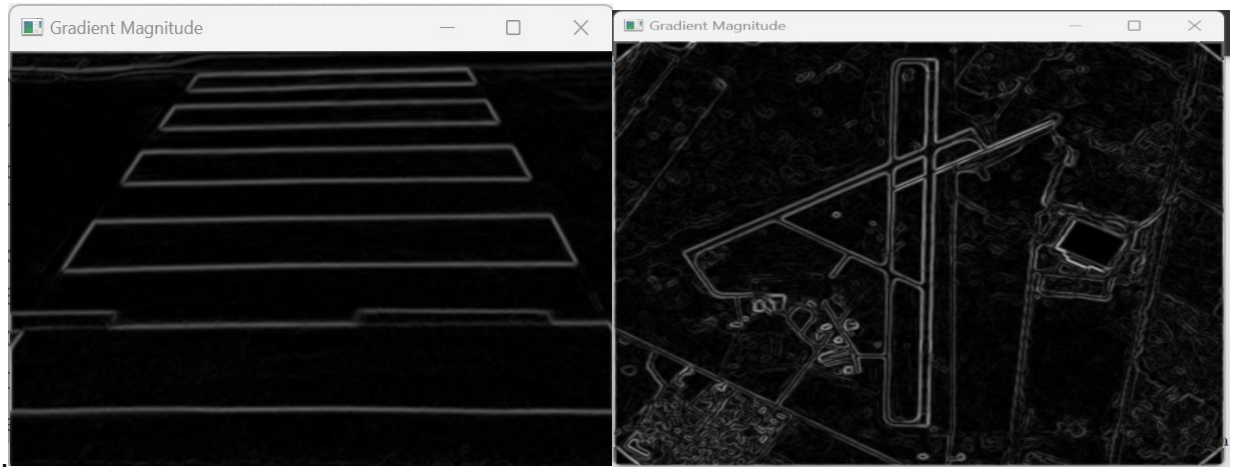
Converting to Grayscale:

Converting the image to grayscale simplifies the subsequent processing steps. The Hough Transform for lines typically operates on grayscale images. Grayscale representation condenses the color information into a single channel, making it easier to analyze intensity variations.

I am using cross walk photos and runway photo below


Computing the Gradient:

The gradient of the image is crucial for detecting edges. The Hough Transform is particularly useful for detecting lines, and edges are essential for identifying potential lines. Computing the gradient involves finding the rate of change of intensity in the x and y directions. Two 1D derivative filters are used to compute the image gradients along the x and y directions. These filters are designed to approximate the derivative of the image intensity. The gradient magnitude is then calculated by combining the gradients along the x and y directions using the Euclidian norm.



The Gaussian filter is then applied to the image using convolution. Convolution involves sliding the Gaussian kernel over the image and computing the weighted sum of pixel values at each position. The application of a Gaussian filter has a smoothing effect on the image. It reduces high-frequency noise and details, making the image more visually appealing. This is particularly useful as a preprocessing step before performing edge detection or other image analysis tasks.

Thresholding to get Binary image

Otsu's method is employed to automatically determine a binary threshold based on the gradient magnitude. This binary image highlights areas with significant changes in intensity, which are potential edges.

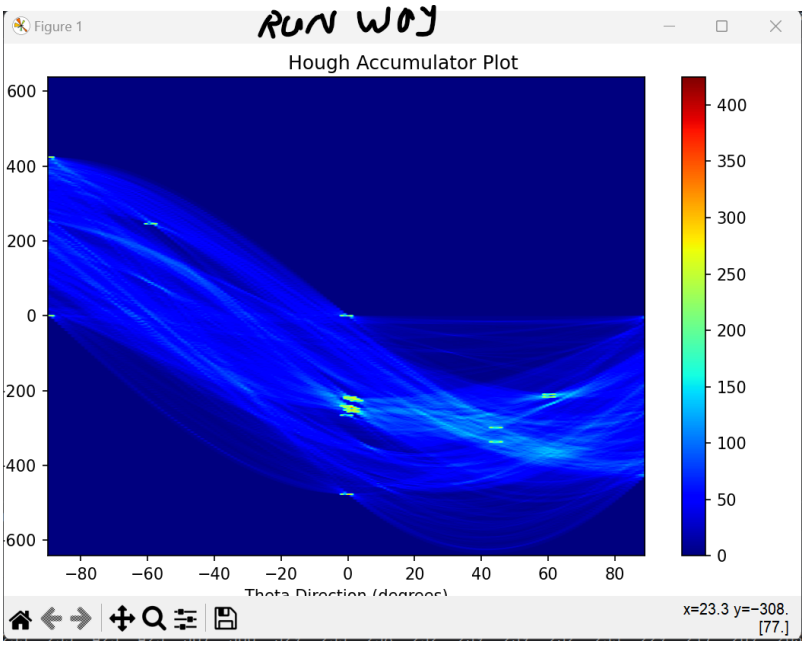
Thresholding is beneficial for focusing on the most relevant parts of the image, reducing. the impact of noise on subsequent processing steps.

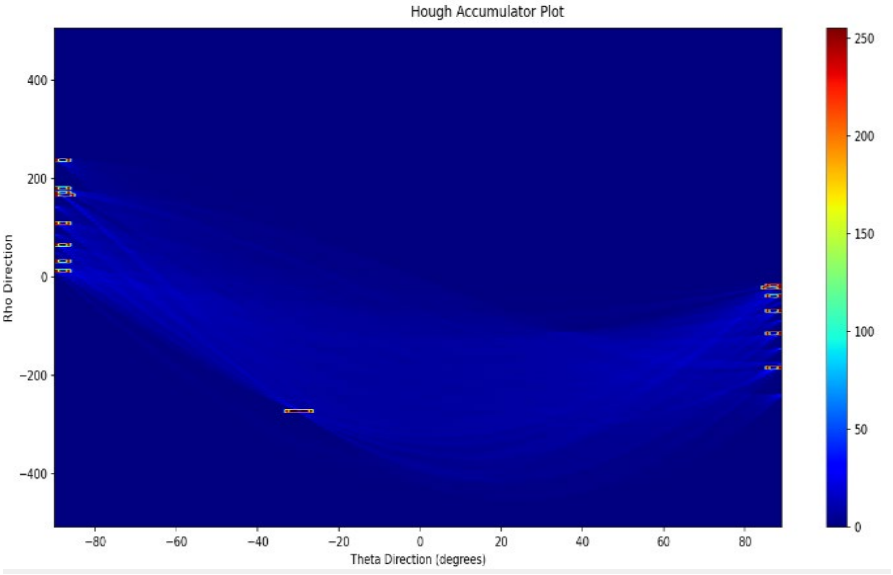
Non-Maximum Suppression:

Edges in an image can have a certain width, and gradient-based edge detection algorithms may produce thicker edges in the output. Non-maximum suppression is applied to reduce the width of detected edges to a single-pixel width. And retain only local maxima in the gradient direction. This step is essential for refining the edge map and ensuring that the subsequent Hough Transform focuses on the most salient features. The suppressed edges image is used as input for the Hough Transform.

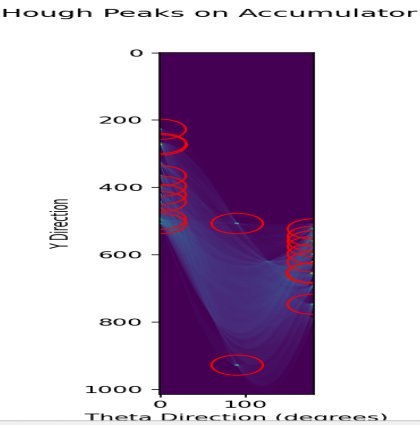
3.3 Hough Lines Accumulator:

The Hough Transform is applied to the binary image to create an accumulator array. This array represents possible lines in the image by accumulating votes for different combinations of rho (distance from the origin) and theta (angle).

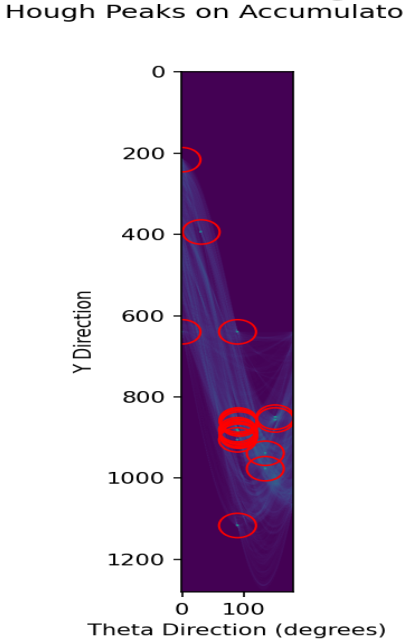




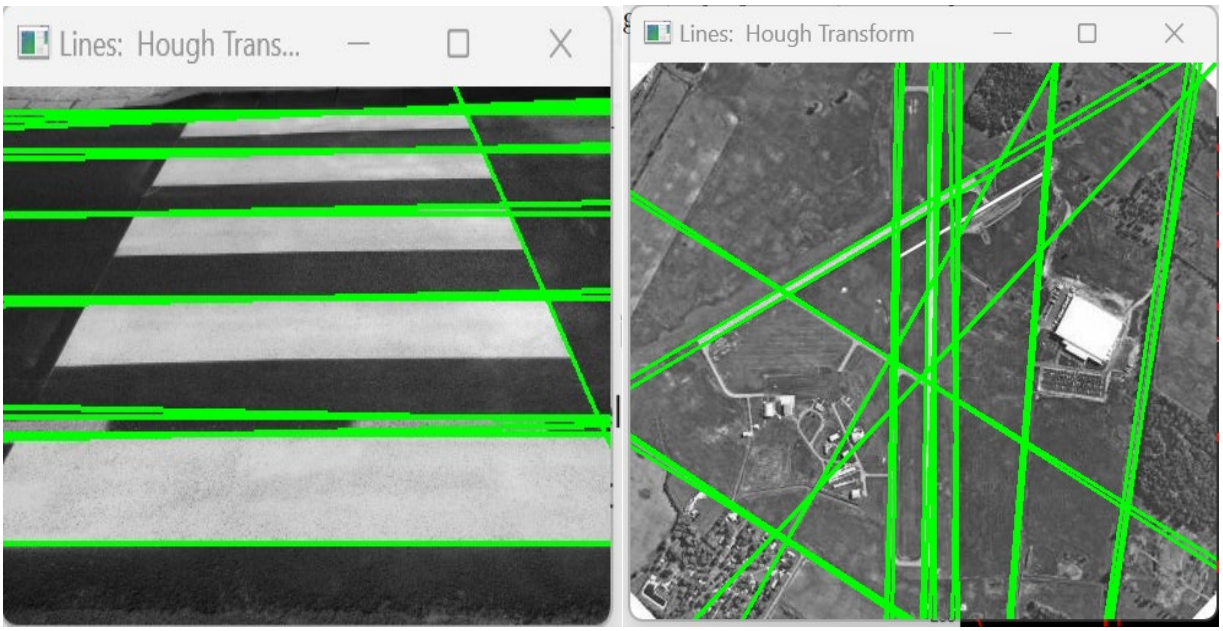
In the case of line detection, the peaks reveal the orientations and positions of lines that are prevalent in the image. identifies peaks in the accumulator array, representing the most prominent lines in the image. Non-maximum suppression is applied to refine the peaks.



RUN WAY



3.5 Drawing Detected Lines:



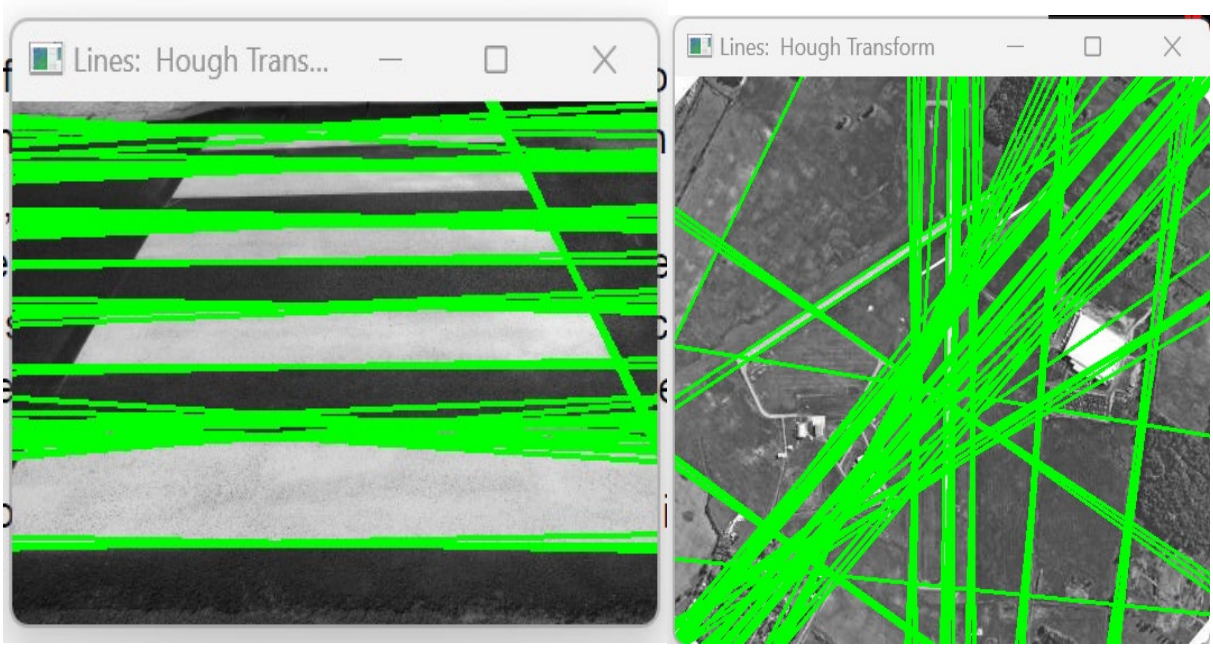
The final step involves drawing the detected lines back onto the original image using the parameters obtained from the accumulator array. The lines are drawn with green color to highlight them. As you can see in the output, the lines detected by the standard hough transform range from one end of the image to the other.

Also, sometimes captures the noise while detecting the straight lines in an image. In the above image we have a threshold of 15 peaks for cross walk photo so the algorithm draw lines for most of the edges. And for the runway photo Right side we used a threshold of 25 and got most of the edges in the photo.

Below image when we increase the threshold number of peaks to 50 for cross walk and 70 for runway location , the algorithm draw lines and detect edges that maybe does not exist and increase the redundancy

Also when we provide the algorithm to take into consideration more neighbors pixels in the selection of the peaks point we get the result belows in the images.

We have several computed lines associated to a single edge, due to the detection of neighboring pixel of a maximum value.



Also if our threshold is too small comparing to the number of actual edges in the image we detect smaller number of edges in the image as below we set the threshold for number of peaks = 10

As we can see since we asked the Hough algorithm to pick only 10 peaks we ended up with less detection of the actual edges in the images and we lost some of the detected lines and mainly the short lines or short edges in the image are missing.

Notice that detecting long lines is not challenging because they have higher value for their peak, but with small lines it is challenging to detect when we lower our threshold of number of peaks and other reasons for example :-

Hough space is discretized, and the resolution of the accumulator grid can affect the ability to detect small lines. If the spacing between bins is too small lines might not accumulate enough votes to stand out from the noise.

The size of the accumulator space is often determined by the range of possible parameters slope and intercept for lines, If the range is too large, small lines might not be well-represented, and their contributions could be spread over a wide range of accumulator cells.

Hough transform is sensitive to noise in the edge detection stage. Small lines may be overwhelmed by noise, making it difficult to distinguish true lines from spurious accumulations.

Conclusion:

The provided code successfully implements the Hough Transform for line detection in images. Despite some challenges, including issues with Gaussian smoothing and line drawing, the core functionality of the Hough Transform is effectively demonstrated. Further optimizations and fine-tuning can enhance the accuracy and visual representation of the detected lines.

Hough Circle Detection

Hough Circle Detection is a technique used in computer vision to identify circles within an image. This method is an extension of the Hough Transform, which is originally designed for detecting lines. The algorithm works by mapping the edge points in an image to circles in the parameter space, and the circles with the highest accumulations of votes are considered as the detected circles. we first guess its radius to construct a new circle. This circle is applied on each black pixel of the original picture and the coordinates of this circle are voting in an accumulator. From this geometrical construction, the original circle center position receives the highest score.

Implementation

The code demonstrates the implementation of Hough Circle Detection. The script is structured to take an input image and several parameters that define the characteristics of the circles to be detected. These parameters include the minimum and maximum radius, delta radius, the number of steps for theta, bin threshold, and edge detection thresholds.

Default Parameters in the code.

Minimum Radius (r_min):

Default Value: 10

The minimum radius parameter specifies the smallest size of circles to be considered during the detection process. A lower value allows the algorithm to detect smaller circles in the image.

Maximum Radius (r_max): Default Value: 200

The maximum radius parameter sets the upper limit for the size of circles to be detected. Circles larger than this radius will not be considered during the detection process.

Delta Radius (delta_r): Default Value: 1

Delta radius represents the step size between consecutive radii to be checked during circle detection. A smaller delta allows for more fine-grained control over the sizes of circles that the algorithm will consider.

Number of Steps for Theta (num_thetas): Default Value: 100

Theta represents the angle parameter in polar coordinates. The number of steps for theta (num_thetas) determines the granularity of angles to be considered during the detection process. A higher value results in a more detailed search for circles at different angles and also will be computationally expensive.

Bin Threshold (bin_threshold): Default Value: 0.4

The bin threshold is a percentage value that determines the minimum percentage of votes a circle must receive to be considered a valid detection. A higher threshold filters out weaker circle candidates, resulting in a more confident set of detected circles.

Min Edge Threshold (min_edge_threshold): Default Value: 100

The minimum edge threshold is used in the Canny edge detection step. It sets the lower threshold for considering a pixel as an edge pixel. Adjusting this threshold influences the sensitivity of the edge detection step.

Max Edge Threshold (max_edge_threshold): Default Value: 200

Similar to the minimum edge threshold, the maximum edge threshold sets the upper limit for considering a pixel as an edge pixel. It influences the specificity of the edge detection step.

Significance of Default Parameters

The default parameters in the code are chosen to provide a reasonable starting point for circle detection. They are based on typical scenarios and can be adjusted by the users depending on the characteristics of the input images and the circles they want to detect. These defaults aim to balance sensitivity and specificity in circle detection while offering flexibility for customization.

Key Components

find_hough_circles Function:

The core of the algorithm is encapsulated in the find_hough_circles function.

It begins by defining the ranges for radius and theta, creating a set of candidate circles based on these ranges.

An accumulator is then initialized to collect votes for each candidate circle.

For every edge pixel in the edge-detected image, the algorithm votes for possible circle centers.

The candidate circles with the highest votes are considered as the detected circles.

Main Function:

The main function serves as the entry point and demonstrates the usage of the find_hough_circles function.

It reads an image, performs edge detection, and applies the Hough Circle Detection algorithm.

Detected circles are drawn on the image, and the results are saved to files.

Post-processing

To enhance the quality of the detected circles, a post-processing step is implemented in the script. This step involves filtering out circles that are too close to each other. The script avoids duplicate circles based on a pixel threshold, providing a cleaner and more accurate result.

Usage

To utilize the script, the user can provide an input image (e.g., onecoin.jpg) and adjust parameters as needed. Parameters such as minimum and maximum radius, delta radius, and thresholds for edge detection can be fine-tuned based on the characteristics of the circles in the input image.

Results

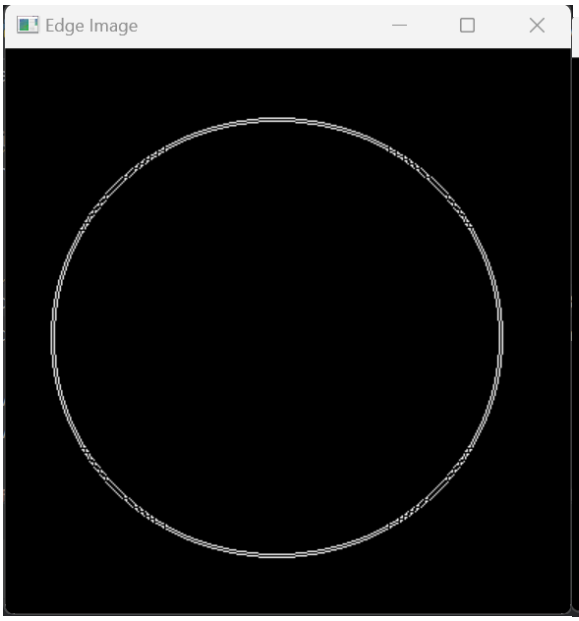
The script outputs both an image with detected circles drawn on it (circles_img.png) and a text file (circles_list.txt) containing the coordinates, radius, and voting percentage for each detected circle.

In Hough transform I decided to use Canny edge detection. So first the input image is converted to grayscale using cv2.cvtColor to simplify processing.

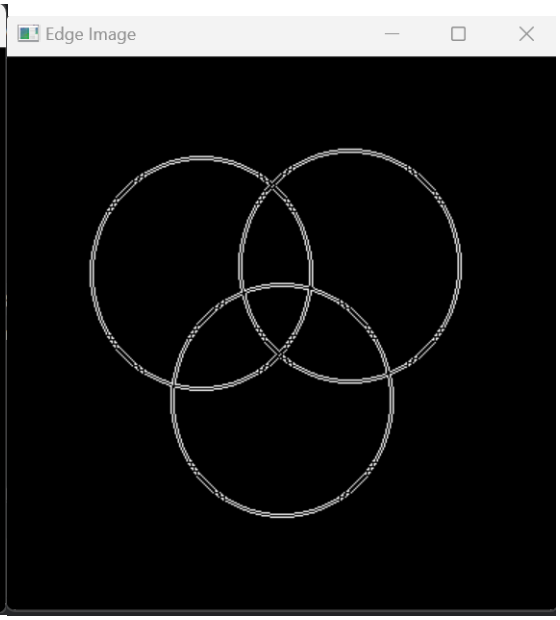
The Canny edge detection algorithm is then applied using cv2.Canny. It helps in detecting edges in the image.

The resulting edge_image is a binary image highlighting edges in the original image. This edge-detected image is then used as input to the Hough Circle detection algorithm.

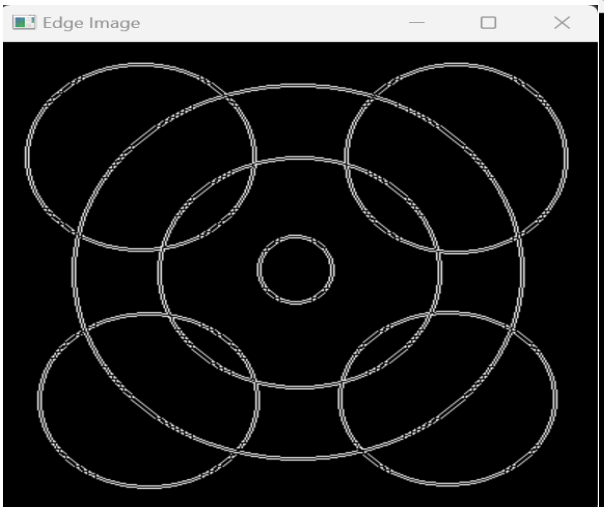
I have used multiple images as examples but mostly I adjusted my defaults parameters to catch these images as their radius varies.



(1)



(2)

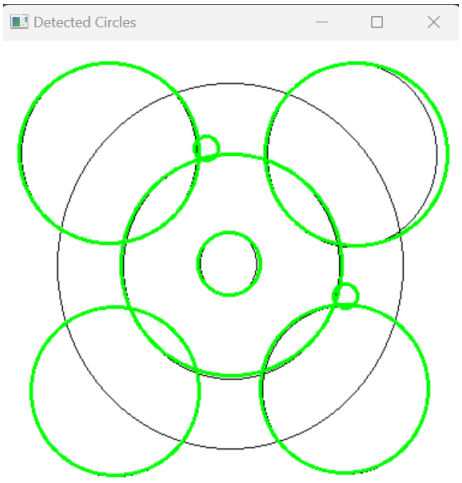


(3)

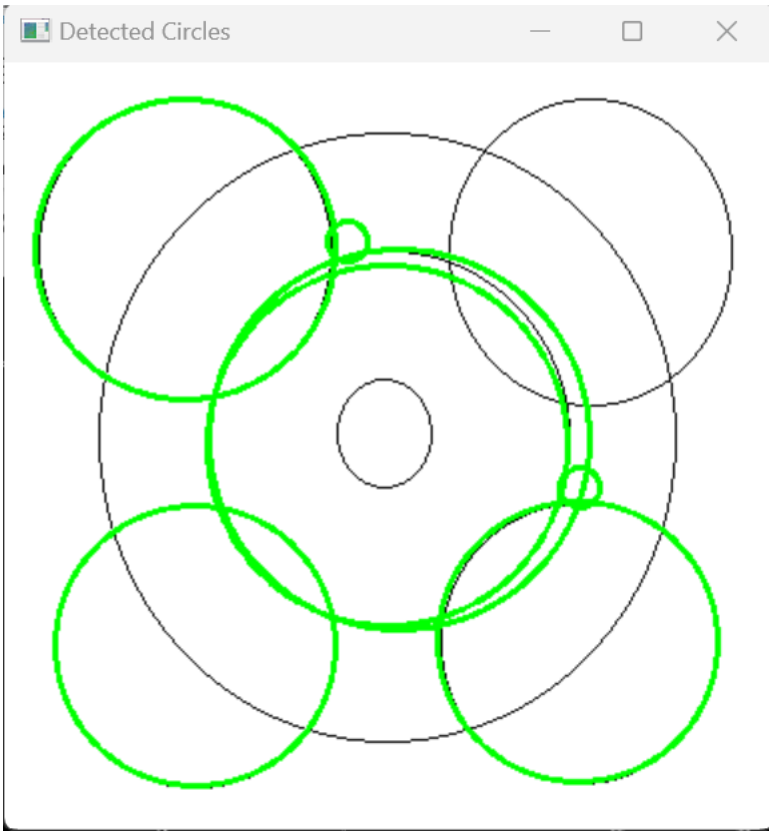
(4) Coins

Hough Circle detection is applied to find circles in the image based on the edges detected. The detected circles are then drawn on the original image. The parameters for Hough Circle detection, such as minimum radius, maximum radius, delta radius, number of thetas, and bin threshold, can be adjusted to fine-tune the circle detection for different images and scenarios. I already talked before about default values above.

For ex. When I changed the max radius = 100 for the algorithm and applied it to image 3.



As we can see the algorithm did not detect the larger radius circle and only got the circles that below the max radius = 100. Also the bin_threshold parameter controls the percentage of votes required for a circle to be considered as detected. If you reduce the maximum radius, it might affect the number of votes a circle receives, and it might not meet the threshold for detection.



Also from the photo above when I changed the delta radius increased from 1 to 5 the algorithm also could not detect some of the circles. increasing the delta radius means that the algorithm will sample fewer radii values within the specified range. This can result in a lower resolution in the search space, potentially causing the algorithm to miss circles with radii that fall between the sampled values. Also If there are circles in the image with radii that don't align well with the sampled radii values due to the larger step size, they may not accumulate enough votes to be detected, especially if the voting is spread across a larger range. Also we see small circle created and that’s because the Hough transform relies on edge detection, and the sensitivity of the edge detection parameters (like thresholds in Canny edge detection) can affect the results. If the edge detection is too sensitive, it may pick up noise or weak edges, creating small circles.

That’s why It's recommended to experiment with different parameter values and find a balance that works well for the specific characteristics of the circles in your images.

Adjusting parameters often involves trade-offs, and fine-tuning is required based on the details of the image content.

Image (4)

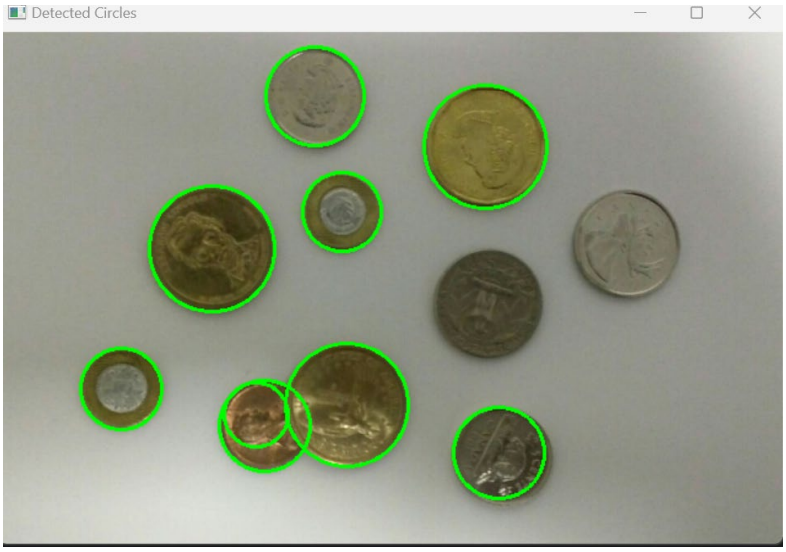
For image 4 that has coins the image shows some challenges that make the circle detection difficult. Firstly, the coins in the image are quite distorted. Some of the coins overlap Again, this makes the detection uneasy. The coins are of various brightness.

Especially the left coin is quite bright, with blurred edge, and contains a lot of noisy texture

All these factors require a more sensitive setting of threshold parameters than in the case of all other images. In order not to lose contour of the bright coins, the edge threshold should be set low. This transfers a lot of noise into the edge map this can be partly removed by smoothing.

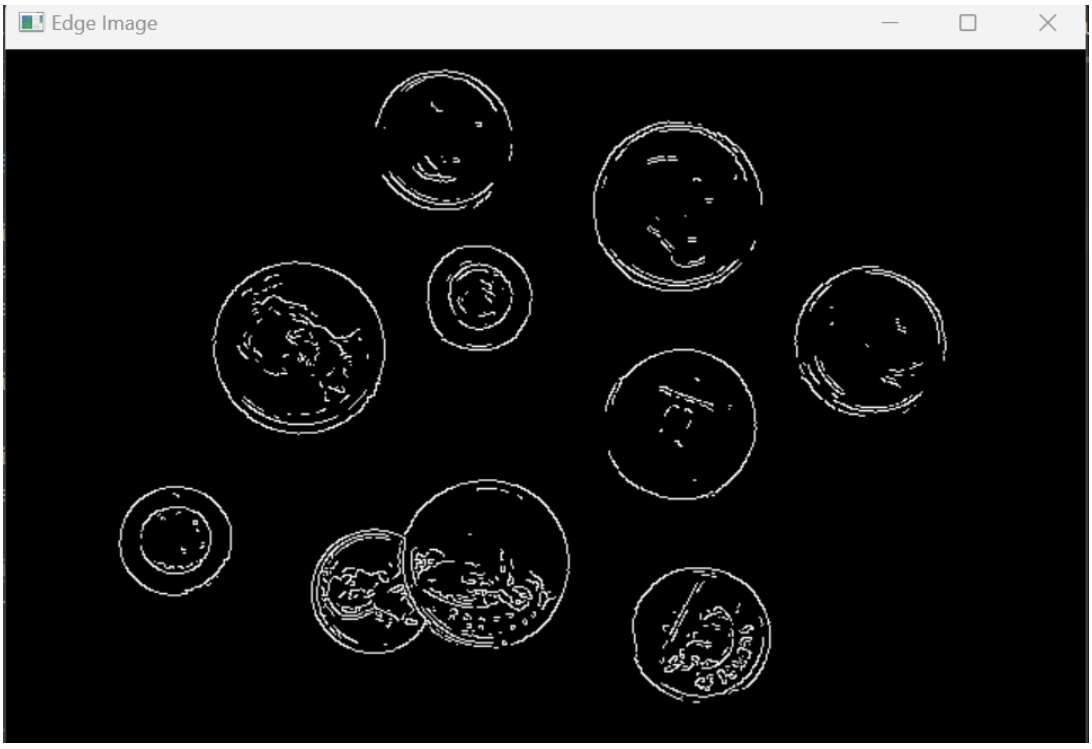
For the 2 coins on top of each other partially we notices the presence of small circles drawn over the coins in the Hough Circle Detection output and that could be attributed to a few reasons. If the input image contains noise or artifacts, especially in areas where the coins overlap, the edge detection step might identify these as edges, leading to the detection of small circles. Noise in the image or artifacts from the imaging process can result in false positives during the Hough Circle Detection.

Also The bin_threshold parameter in the Hough Circle Detection algorithm controls the threshold for shortlisting candidate circles based on the percentage of votes. If the threshold is set too low, it may include circles with fewer votes, leading to the detection of additional, unwanted circles. we may need to adjust this threshold to filter out circles with lower confidence.



Also in the image above, the algorithm does not detect the most right 2 coins one dark and one light and that can be for many reasons If some coins are significantly darker or brighter than the background or other coins, it may lead to difficulties in edge detection and subsequent circle detection. The edge detection step, particularly in the Canny edge detector, relies on setting appropriate threshold values. If these values are not well-tuned for the specific characteristics of your images, certain edges, especially those associated with darker or brighter coins, might not be detected accurately.

I changed the threshold to min_edge_threshold = 100 and max_edge_threshold = 120 so canny edge detection detect the edges for the last 2 circles. See below the binary image after adjusting the threshold.



Now the last 2 coins are detected and our algorithm identifies them and mark them

And here is our final results.

Conclusion

Hough Circle Detection is a powerful technique for identifying circles in images, commonly used in computer vision applications. The provided script offers a flexible implementation, of different circle sizes that varies between min, and max radius as a default. But it is allowing users to adapt parameters to different scenarios. By combining edge detection and a voting-based approach, the algorithm achieves robust circle detection.

```python
# Hough Line Transform

import numpy as np
import cv2
import matplotlib.pyplot as plt



def convolution_2d(image, kernel):
    # Convert the input image and kernel to float arrays
    image = image.astype(float)
    kernel = kernel.astype(float)
    print("image shapes dimensions")
    print(image.shape)

    # Get the dimensions of the image and kernel
    image_height, image_width = image.shape
    kernel_height, kernel_width = kernel.shape

    # Calculate the padding for boundary handling
```

```
    pad_height = kernel_height // 2
    pad_width = kernel_width // 2

    # Create an output array to store the result
    output = np.zeros(image.shape, dtype=float)

    # Pad the image with zeros
    padded_image = np.pad(image, ((pad_height, pad_height), (pad_width, pad_width)),
mode='constant')

    # Flip the kernel
    kernel = np.flipud(np.fliplr(kernel))

    # Perform 2D convolution
    for i in range(image_height):
        for j in range(image_width):
            output[i, j] = np.sum(padded_image[i:i+kernel_height, j:j+kernel_width] * kernel)

    return output

# Gaussian filter for denoising
def gaussian_filter(size, sigma):
    kernel = np.fromfunction(lambda x, y: (1/ (2 * np.pi * sigma**2)) * np.exp(-((x - (size-
1)/2)**2 + (y - (size-1)/2)**2) / (2 * sigma**2)), (size, size))
    return kernel / np.sum(kernel)


def auto_threshold_otsu(image):
    # Convert image to 8-bit unsigned integer
    image_8bit = cv2.convertScaleAbs(image)
    _, thresholded = cv2.threshold(image_8bit, 0, 255, cv2.THRESH_BINARY +
```

```
cv2.THRESH_OTSU)
    return thresholded


def hough_lines_acc(img, rho_resolution=1, theta_resolution=1):
    # Function for creating a Hough Accumulator for lines in an image.
    height, width = img.shape
    img_diagonal = np.ceil(np.sqrt(height ** 2 + width ** 2))

    # Ensure that rhos cover the expected range
    rhos = np.arange(-img_diagonal, img_diagonal + 1, rho_resolution)

    # Ensure that thetas cover the expected range
    thetas = np.deg2rad(np.arange(-90, 90, theta_resolution))

    # Create an accumulator array
    H = np.zeros((len(rhos), len(thetas)), dtype=np.uint64)

    y_idxs, x_idxs = np.nonzero(img)

    for i in range(len(x_idxs)):
        x = x_idxs[i]
        y = y_idxs[i]

        for j in range(len(thetas)):
            rho = int((x * np.cos(thetas[j]) + y * np.sin(thetas[j])) + img_diagonal)

            # Check if rho is within the bounds of the accumulator array
            if 0 <= rho < len(rhos):
                H[rho, j] += 1
```

```
    return H, rhos, thetas


def hough_peaks(H, num_peaks, threshold=0, nhood_size=3):
    indicies = []
    H1 = np.copy(H)
    for i in range(num_peaks):
        idx = np.argmax(H1)
        H1_idx = np.unravel_index(idx, H1.shape)
        indicies.append(H1_idx)

        idx_y, idx_x = H1_idx
        if (idx_x - (nhood_size / 2)) < 0:
            min_x = 0
        else:
            min_x = idx_x - (nhood_size / 2)
        if ((idx_x + (nhood_size / 2) + 1) > H.shape[1]):
            max_x = H.shape[1]
        else:
            max_x = idx_x + (nhood_size / 2) + 1

        if (idx_y - (nhood_size / 2)) < 0:
            min_y = 0
        else:
            min_y = idx_y - (nhood_size / 2)
        if ((idx_y + (nhood_size / 2) + 1) > H.shape[0]):
            max_y = H.shape[0]
        else:
            max_y = idx_y + (nhood_size / 2) + 1

        for x in range(int(min_x), int(max_x)):
```

```
            for y in range(int(min_y), int(max_y)):
                H1[y, x] = 0
                if (x == int(min_x) or x == int(max_x) - 1):
                    H[y, int(x)] = 255
                if (y == int(min_y) or y == int(max_y) - 1):
                    H[int(y), x] = 255

    return indicies, H

def plot_hough_acc(H, plot_title='Hough Accumulator Plot'):
    ''' A function that plots a Hough Space using Matplotlib. '''
    fig, ax = plt.subplots(figsize=(5, 5))  # Adjust figsize as needed
    fig.suptitle(plot_title)

    # Set the range of theta values based on the accumulator shape
    theta_values = np.deg2rad(np.arange(-90, 90, 1))

    # Set the range of rho values based on the accumulator shape
    rho_values = np.arange(-H.shape[0] // 2, H.shape[0] // 2, 1)

    cax = ax.imshow(H, cmap='jet',
                    extent=[np.rad2deg(theta_values.min()), np.rad2deg(theta_values.max()),
rho_values.min(),
                            rho_values.max()])

    plt.xlabel('Theta Direction (degrees)')
    plt.ylabel('Rho Direction')
    cbar = plt.colorbar(cax)

    # Set the aspect ratio to 'auto' to prevent squeezing
    ax.set_aspect('auto')
```

```
    plt.tight_layout()
    plt.show()


def plot_hough_peaks_on_Accumulator(image, indicies, rhos, thetas, plot_title='Hough
Peaks on Accumulator'):
    ''' A function that plots the image with selected peaks. '''

    print("Your indices ")
    print(indicies)
    fig, ax = plt.subplots(figsize=(15, 8))
    fig.suptitle(plot_title)

    # Display the original image
    ax.imshow(image)

    for center in indicies:
        x, y = center[1], center[0]
        print("Center:", x, y)
        ax.add_patch(plt.Circle((x, y), 30, color='r', fill=False))

    plt.xlabel('Theta Direction (degrees)')
    plt.ylabel('Y Direction')
    plt.show()


def hough_lines_draw(img, indicies, rhos, thetas):
    # Function that takes indices, a rhos table, and thetas table and draws
    # lines on the input images that correspond to these values.
```

```
    for i in range(len(indicies)):
        rho = rhos[indicies[i][0]]
        theta = thetas[indicies[i][1]]
        a = np.cos(theta)
        b = np.sin(theta)
        x0 = a * rho
        y0 = b * rho
        x1 = int(x0 + 1000 * (-b))
        y1 = int(y0 + 1000 * (a))
        x2 = int(x0 - 1000 * (-b))
        y2 = int(y0 - 1000 * (a))

        cv2.line(img, (x1, y1), (x2, y2), (0, 255, 0), 2)

# suppress edges to thin the edges
def non_maximum_suppression(img, angles):
    size = img.shape
    suppressed_edges = np.zeros(size)

    for i in range(1, size[0] - 1):
        for j in range(1, size[1] - 1):
            angle = angles[i, j]

            # Define neighbors based on the gradient angle
            if (0 <= angle < 22.5) or (157.5 <= angle <= 180):
                q = img[i, j - 1]
                r = img[i, j + 1]
            elif (22.5 <= angle < 67.5):
                q = img[i - 1, j - 1]
                r = img[i + 1, j + 1]
            elif (67.5 <= angle < 112.5):
```

```
                q = img[i - 1, j]
                r = img[i + 1, j]
            else:
                q = img[i + 1, j - 1]
                r = img[i - 1, j + 1]

            # Suppress non-maximum values
            if img[i, j] >= q and img[i, j] >= r:
                suppressed_edges[i, j] = img[i, j]

    suppressed_edges = np.multiply(suppressed_edges, 255.0 /
suppressed_edges.max())
    return suppressed_edges


# Read in shapes image and convert to grayscale
image = cv2.imread('mnn4-runway-Ohio.jpg')
cv2.imshow('Original Image', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
image_grayscale = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

# Denoise the image with a Gaussian filter

Gx = gaussian_filter(size=5, sigma=1)
Gy = np.transpose(Gx)

print("gaussian gx ", Gx)
print("gaussian gy ", Gy)

denoised_image = convolution_2d(image_grayscale, Gx )
```

```
denoised_image = convolution_2d(image_grayscale, Gy )

# Derivatives
# 1D derivative filters
dx_filter = np.array([[-1, 0, 1]], dtype=float)
dy_filter = np.array([[-1], [0], [1]], dtype=float)

# Compute derivatives
dx_image = convolution_2d(denoised_image, dx_filter)
dy_image = convolution_2d(denoised_image, dy_filter)



# Compute gradient magnitude image
gradient_magnitude = np.sqrt(dx_image**2 + dy_image**2)
print("Denoised Image:")
print(denoised_image)
print("Gradient Magnitude Image:")
print(gradient_magnitude)

##############   getting the angle to do thining of edges

gradient_angle = np.arctan2(dy_image, dx_image)

#suppresed_edges_image = non_maximum_suppression(gradient_magnitude,
gradient_angle)



# Edges is the binary image returned by Otsu

edges = auto_threshold_otsu(gradient_magnitude)
#edges = auto_threshold_otsu(suppresed_edges_image)
```

```
# Show gradient magnitude
cv2.imshow('Gradient Magnitude', gradient_magnitude.astype(np.uint8))
cv2.waitKey(0)
cv2.destroyAllWindows()

# Show thresholded edges
cv2.imshow('Thresholded Edges- Binary image after Otsu', edges)
cv2.waitKey(0)
cv2.destroyAllWindows()

# Hough Lines
H, rhos, thetas = hough_lines_acc(edges)
indicies, H = hough_peaks(H, 20, nhood_size=3)


print("Selected indices:", indicies)
print("Accumulator function values:", [H[idx] for idx in indicies])
plot_hough_acc(H)


print("Selected indices:", indicies)
print("Accumulator function values:", [H[idx] for idx in indicies])
plot_hough_peaks_on_Accumulator(H, indicies, rhos, thetas)




hough_lines_draw(image, indicies, rhos, thetas)
```

```
# Show image with manual Hough Transform Lines
cv2.imshow('Lines:  Hough Transform', image)
cv2.waitKey(0)
cv2.destroyAllWindows()

# Select the N largest maxima
N = 15
selected_indices = indicies[:N]

# Write the pairs (ρ_j, θ_j, nvotes_j) to a file

with open("output.txt", "w", encoding="utf-8") as file:
    for (rho_index, theta_index) in indicies:
        rho = rhos[rho_index]
        theta = thetas[theta_index]
        nvotes = H[rho_index, theta_index]
        file.write(f"ρ: {rho}, θ: {theta}, nvotes: {nvotes}\n"
```

Hough Circle Code.

```
# -*- coding: utf-8 -*-

import cv2
import numpy as np
from collections import defaultdict



def detect_circles_hough(input_image, edge_image, min_radius, max_radius,
delta_radius, num_thetas, bin_threshold, post_process=True):
    # Get the dimensions of the image
    img_height, img_width = edge_image.shape[:2]

    # Define the range for theta values
    theta_step = int(360 / num_thetas)
    thetas = np.arange(0, 360, step=theta_step)

    # Define the range for radii
    radii = np.arange(min_radius, max_radius, step=delta_radius)

    # Pre-calculate Cos(theta) and Sin(theta)
    cos_thetas = np.cos(np.deg2rad(thetas))
    sin_thetas = np.sin(np.deg2rad(thetas))

    # Generate candidate circles for different radii and thetas
    circle_candidates = []
    for radius in radii:
        for theta_index in range(num_thetas):
```

```
            x_offset = int(radius * cos_thetas[theta_index])
            y_offset = int(radius * sin_thetas[theta_index])
            circle_candidates.append((radius, x_offset, y_offset))

    # Initialize the Hough Accumulator
    accumulator = defaultdict(int)

    # Vote for candidate circles based on edge pixels
    for y in range(img_height):
        for x in range(img_width):
            if edge_image[y][x] != 0:  # White pixel (edge)
                for radius, x_offset, y_offset in circle_candidates:
                    x_center = x - x_offset
                    y_center = y - y_offset
                    accumulator[(x_center, y_center, radius)] += 1  # Vote for the current
candidate

    # Output image with detected circles drawn
    output_image = input_image.copy()

    # Output list of detected circles in the format (x, y, radius, threshold)
    detected_circles = []

    # Sort the accumulator based on the votes for the candidate circles
    for circle, votes in sorted(accumulator.items(), key=lambda item: -item[1]):
        x, y, radius = circle
        current_vote_percentage = votes / num_thetas
        if current_vote_percentage >= bin_threshold:
            detected_circles.append((x, y, radius, current_vote_percentage))

    # Post-process the results to remove nearby duplicate circles
```

```
    if post_process:
        pixel_threshold = 5
        postprocessed_circles = []
        for x, y, radius, threshold in detected_circles:
            if all(
                abs(x - xc) > pixel_threshold or abs(y - yc) > pixel_threshold or abs(radius -
rc) > pixel_threshold
                for xc, yc, rc, _ in postprocessed_circles
            ):
                postprocessed_circles.append((x, y, radius, threshold))
        detected_circles = postprocessed_circles

    # Draw the shortlisted circles on the output image
    for x, y, radius, _ in detected_circles:
        output_image = cv2.circle(output_image, (x, y), radius, (0, 255, 0), 2)

    return output_image, detected_circles

###############

import cv2
import numpy as np
from matplotlib import pyplot as plt



def visualize_accumulator(accumulator, thetas, radii):
    """
    Visualize the Hough accumulator matrix.

    Parameters:
        accumulator (numpy.ndarray): The Hough accumulator matrix.
```

```
        thetas (numpy.ndarray): Array of theta values.
        radii (numpy.ndarray): Array of radius values.
    """
    plt.imshow(accumulator, cmap='viridis', extent=[np.min(thetas), np.max(thetas),
np.min(radii), np.max(radii)],
               aspect='auto', origin='lower')
    plt.colorbar()
    plt.title('Hough Accumulator')
    plt.xlabel('Theta (degrees)')
    plt.ylabel('Radius')
    plt.show()




##########
def main():
    # Set the parameters for circle detection
    min_radius = 25
    max_radius = 200
    delta_radius = 1
    num_thetas = 100
    bin_threshold = 0.4
    min_edge_threshold = 100
    max_edge_threshold = 200

    #input_image = cv2.imread("standard.png")
    input_image = cv2.imread("standard.png")


    # Edge detection on the input image
```

```
    edge_image = cv2.cvtColor(input_image, cv2.COLOR_BGR2GRAY)
    edge_image = cv2.Canny(edge_image, min_edge_threshold, max_edge_threshold)

    cv2.imshow('Edge Image', edge_image)
    cv2.waitKey(0)

    if edge_image is not None:
        print("Detecting Hough Circles Started!")
        detected_circles_image, circles_info = detect_circles_hough(
            input_image, edge_image, min_radius, max_radius, delta_radius, num_thetas,
bin_threshold
        )

        cv2.imshow('Detected Circles', detected_circles_image)
        cv2.waitKey(0)

        # Save the information about detected circles to a file
        with open('circles_info.txt', 'w') as circle_file:
            circle_file.write('x ,\ty ,\tRadius ,\tThreshold \n')
            for circle in circles_info:
                circle_file.write(f"{circle[0]} , {circle[1]} , {circle[2]} , {circle[3]}\n")

        if detected_circles_image is not None:
            cv2.imwrite("detected_circles_image.png", detected_circles_image)
    else:
        print("Error in input image!")

    print("Detecting Hough Circles Complete!")


if __name__ == "__main__":
```

```
    main()


"""
Circle Detection using Hough Transform

Parameters:
    min_radius - Minimum radius of circles to detect. Default is 10.
    max_radius - Maximum radius of circles to detect. Default is 200.
    delta_radius - Step size for varying the radius from min to max. Default is 1.
    num_thetas - Number of steps for theta from 0 to 2PI. Default is 100.
    bin_threshold - Thresholding value in percentage to shortlist candidate circles.
Default is 0.4 i.e., 40%.
    min_edge_threshold - Minimum threshold value for edge detection. Default 100.
    max_edge_threshold - Maximum threshold value for edge detection. Default 200.

Note:
    Adjusting input parameters is necessary for detecting circles of different sizes in
different images.

Output:
    detected_circles_img - Image with the detected circles drawn
    circles_info.txt - List of detected circles in format (x, y, radius, votes)

"""
```