ARISTOTLE UNIVERSITY OF THESSALONIKI

Department of Electrical & Computer Engineering
GE1301 - Digital Image Processing

Hough Transform, Harris Corner Detector and Image Rotation

Prepared by: Nikolaos Andriotis 9472
Instructor: Anastasios Ntelopoulos, Dimitrios Aletras
Date: May 25, 2024

1 Explanation of the Hough Transform Algorithm

Steps of the Algorithm

1. Image Dimensions and Accumulator Initialization

The function begins by extracting the height and width of the input binary image. It then calculates the maximum possible value of ρ based on the image dimensions. The ranges for ρ and θ are created using the provided `d_rho` and `d_theta` resolutions. An accumulator array H is initialized to accumulate votes.

2. Edge Point Detection

The function identifies the indices of the edge points in the binary image using `np.nonzero`. These points are essential for the Hough Transform, as they are the points that contribute to the detection of lines.

3. Hough Transform Voting

For each edge point, the function computes the corresponding ρ for each θ and updates the accumulator array H. This is done using the equation:

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

where x and y are the coordinates of the edge point.

4. Identifying Top Lines

The function then identifies the top `n` lines by finding the highest values in the accumulator array. A temporary copy of H is used to avoid modifying the original accumulator during this process. The highest values are located, and the corresponding ρ and θ values are stored as the top lines.

5. Pixel Association with Top Lines

The function creates a set to track the pixels that belong to the top `n` lines. It iterates over each detected line and checks if the pixels in the binary image lie on these lines using a small tolerance (`d_rho`). If a pixel lies on one of the top lines, it is added to the set.

6. Counting Non-Line Pixels

Finally, the function counts the number of edge pixels that do not belong to any of the top `n` lines by subtracting the set of pixels on the top lines from the set of all edge pixels.

Considerations

Handling Image Boundaries

The function ensures that the computed ρ values fall within the bounds of the accumulator array. This prevents indexing errors and ensures that all votes are correctly counted.

Efficiency

The algorithm uses nested loops to process each edge point and each θ value, which can be computationally intensive for large images and small θ increments. Utilizing efficient array operations and possibly parallel processing can enhance performance.

Flexibility

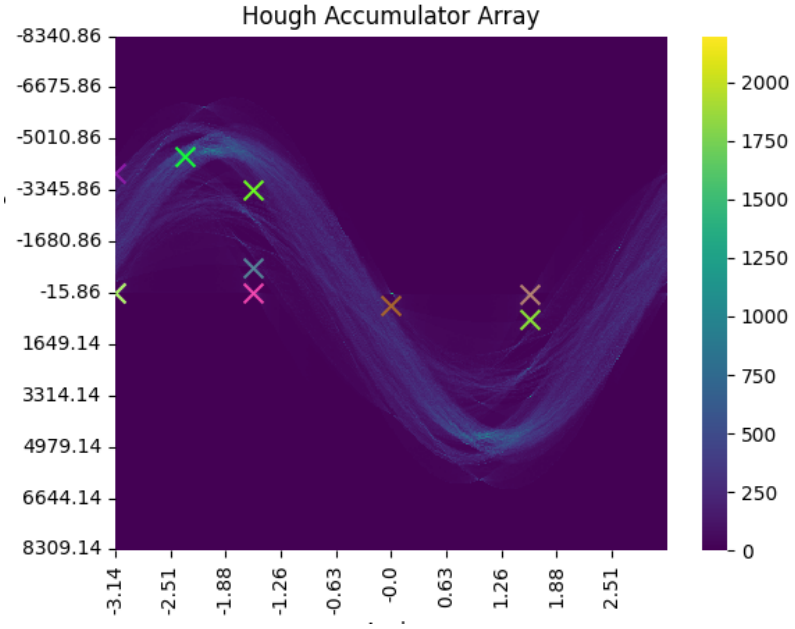
The function can be applied to any binary image and allows for adjusting the resolution of the accumulator array and the number of lines to detect. This makes the function versatile for various applications and image sizes.

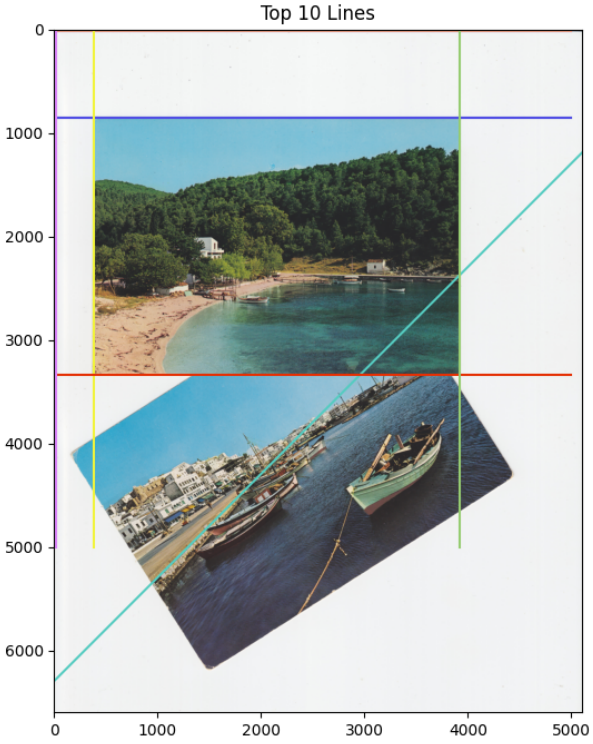
Example Usage

Here is an example of how to use the `my_hough_transform` function:

```
original = io.imread("data/im2.jpg")
gray = color.rgb2gray(original)
binary = feature.canny(gray, sigma=5)
d_rho = 5
d_theta = np.pi / 180
n = 10
H, L, res = my_hough_transform(binary, d_rho, d_theta, n)
print(H)
print(lines)
print(res)
```

Results


Hough Accumulator Array
-8340.86
-6675.86
-5010.86
-3345.86
-1680.86
-15.86
1649.14
3314.14
4979.14
6644.14
8309.14
-3.14
-2.51
-1.88
-1.26
-0.63
-0.0
0.63
1.26
1.88
2.51
2000
1750
1500
1250
1000
750
500
250
0


Top 10 Lines
0
1000
2000
3000
4000
5000
6000
0
1000
2000
3000
4000
5000

2 Explanation of Harris Corner Detection Algorithm

The provided Python code performs Harris corner detection on a given grayscale image. Harris corner detection is a popular method in computer vision for identifying points of interest within an image, specifically corners.

Functions and Their Roles

1. Gaussian Kernel

The function `gaussian_kernel` generates a Gaussian kernel used for smoothing images. Smoothing is a critical step in image processing to reduce noise and enhance the detection of meaningful features.

```
def gaussian_kernel(size: int, sigma: float) -> np.ndarray:
    """
    Generate a Gaussian kernel.
    """
    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)
```

2. Harris Corner Response

The function `harris_response` computes the Harris corner response for an image. This involves several steps: computing image gradients using Sobel operators, smoothing the squared gradients with a Gaussian filter, and calculating the Harris response using the determinant and trace of the structure tensor.

```
def harris_response(img: np.ndarray, k: float, sigma: float) -> np
    .ndarray:
    """
    Compute the Harris corner response for an image.
    """
    # Sobel kernels
    sobel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
    sobel_y = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])

    # Compute the image gradients
    I_x = convolve2d(img, sobel_x, mode="same")
    I_y = convolve2d(img, sobel_y, mode="same")

    # Gaussian kernel
    size = int(4 * sigma)
    gaussian_kernel_ = gaussian_kernel(size, sigma)
```

```
    # Compute the elements of the structure tensor
    I_x2 = convolve2d(I_x**2, gaussian_kernel_, mode="same")
    I_y2 = convolve2d(I_y**2, gaussian_kernel_, mode="same")
    I_xy = convolve2d(I_x * I_y, gaussian_kernel_, mode="same")

    # Compute the Harris response
    det = I_x2 * I_y2 - I_xy**2
    trace = I_x2 + I_y2
    response = det - k * trace**2

    return response
```

3. Corner Locations

The function `corner_locations` identifies the corner locations in the Harris response map by applying a relative threshold. Only points with response values above the threshold are considered as corners.

```
def corner_locations(harris_response: np.ndarray, rel_threshold:
    float) -> np.ndarray:
    """
    Find corner locations in the Harris response map.
    """
    # Compute the threshold
    threshold = rel_threshold * harris_response.max()
    # Find corners by applying a threshold
    corner_locations = harris_response > threshold
    # Find the indices of the corners
    corner_locations = np.argwhere(corner_locations)
    return corner_locations
```

Algorithm Workflow

1. **Read and Preprocess Image:** The algorithm starts by reading an image and converting it to grayscale. Edge detection is performed using the Canny edge detector.
2. **Compute Harris Response:** The `harris_response` function is called to compute the Harris corner response of the grayscale image. This involves gradient computation, Gaussian smoothing, and response calculation.
3. **Detect Corners:** The `corner_locations` function is used to detect corner points by applying a relative threshold to the Harris response map.
4. **Display Results:** The detected corners are plotted on the original image for visualization.

Example Usage

The following example demonstrates how to use the provided functions to detect and visualize corners in an image:

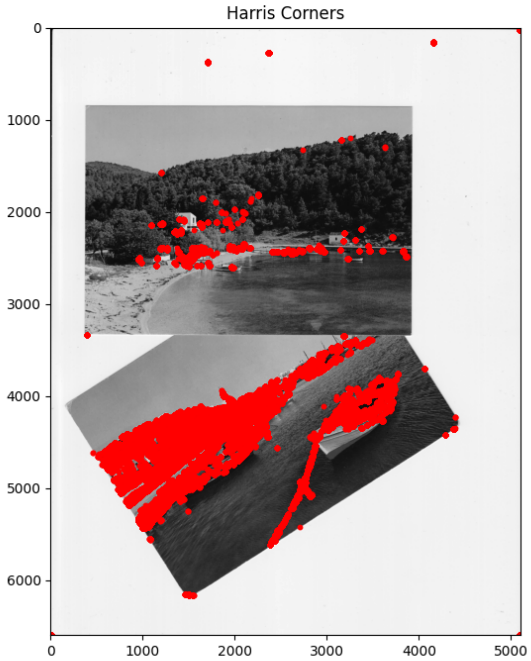
```
print("Reading image...")
original = io.imread("data/im2.jpg")
gray = color.rgb2gray(original)
binary = feature.canny(gray, sigma=5)

k = 0.04
sigma = 3.0
print("Detecting corners...")
response = harris_response(gray, k, sigma)
rel_threshold = 0.1
print("Finding corner locations...")
corners = corner_locations(response, rel_threshold)

plt.figure(figsize=(10, 10))
plt.imshow(gray, cmap="gray")
plt.scatter(corners[:, 1], corners[:, 0], c="r", s=10)
plt.title("Harris Corners")
plt.show()
```

This code reads an image, converts it to grayscale, and applies the Harris corner detection algorithm to find and visualize corners.

Results



3 Explanation of the Image Rotation Algorithm

The provided Python function `my_img_rotation` performs the task of rotating an image by a specified angle using bilinear interpolation. This algorithm is useful in various image processing applications where image alignment, augmentation, or transformation is required.

Steps of the Algorithm

Image Dimensions and Rotation Matrix Calculation

The function begins by extracting the height and width of the input image. It then calculates the elements of the rotation matrix using trigonometric functions. The rotation matrix is defined as:

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

where θ is the rotation angle in radians.

New Bounding Box Dimensions

To ensure that the entire rotated image fits within the new dimensions without cropping, the function calculates the dimensions of the bounding box that can contain the rotated image. This is done by computing the maximum extents of the rotated corners of the original image.

Initialization of Output Image

An output image (`rot_img`) is initialized with a black background (all zeros). The size of this image corresponds to the new bounding box dimensions. The data type and number of channels (grayscale or RGB) are preserved from the original image.

Center Coordinates Calculation

The centers of both the original and the new images are calculated. These centers are used as reference points for the rotation transformation.

Rotation and Bilinear Interpolation

The algorithm iterates over each pixel in the output image. For each pixel, it computes the corresponding coordinates in the original image using the inverse of the rotation transformation. This involves translating the pixel coordinates to the origin, rotating them, and then translating them back to the original image coordinates. Bilinear interpolation is used to compute the pixel values at these coordinates, ensuring smooth transitions and reducing artifacts.

Bilinear interpolation works by taking a weighted average of the four nearest pixels in the original image. This process involves:

- Identifying the four surrounding pixels.
- Calculating the distances from the target coordinates to these surrounding pixels.
- Using these distances to compute the weighted average, which gives the final pixel value for the rotated image.

Considerations

Handling Image Boundaries

The function ensures that pixel coordinates computed during the transformation process are within the bounds of the original image. If the computed coordinates fall outside the original image, they are ignored.

Efficiency

The function uses nested loops to process each pixel, which can be computationally intensive for large images. Bilinear interpolation, while providing smoother results than nearest-neighbor interpolation, also adds to the computational complexity. Optimizations such as vectorized operations or parallel processing can be considered for performance improvements.

Flexibility

The function is designed to handle both grayscale and RGB images. It dynamically adjusts its operations based on the number of channels in the input image.

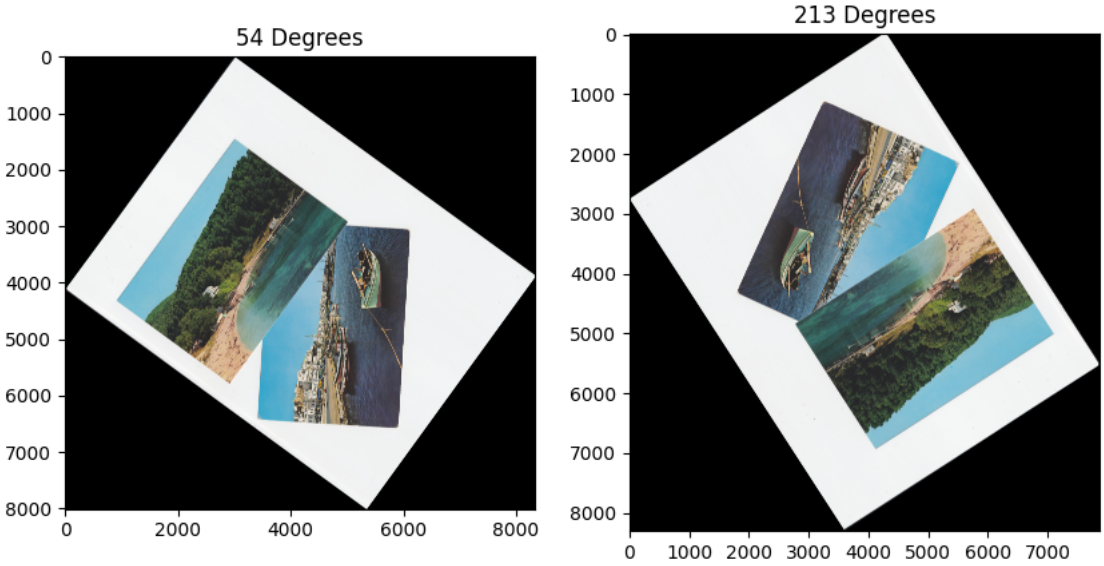
In summary, the `my_img_rotation` function effectively rotates an image by a specified angle, ensuring that the entire image is captured within the new dimensions and that the rotation is smooth and visually appealing. The use of bilinear interpolation enhances the quality of the rotated image by providing smooth transitions between pixel values.

Example usage

```
print("Reading image...")
original = io.imread("data/im2.jpg")
angle_54 = (np.pi / 180) * 54  # 54 degrees in radians
angle_213 = (np.pi / 180) * 213  # 213 degrees in radians

rotated_img_54 = my_img_rotation(original, angle_54)
rotated_img_213 = my_img_rotation(original, angle_213)
plt.figure(figsize=(10, 10))
plt.subplot(1, 2, 1)
plt.imshow(rotated_img_54, cmap="gray")
plt.title("54 Degrees")
plt.subplot(1, 2, 2)
plt.imshow(rotated_img_213, cmap="gray")
plt.title("213 Degrees")
plt.show()
```

Results


54 Degrees
0
1000
2000
3000
4000
5000
6000
7000
8000
0
2000
4000
6000
8000
213 Degrees
0
1000
2000
3000
4000
5000
6000
7000
8000
0
1000
2000
3000
4000
5000
6000
7000