Assignment 1

Nabeel Mohamed	7547
Mohamed Milad	7479
Eyad Allafy	7839

Part I: Applying Image Processing Filters for Image Cartoonifying:

Code Explanation

1. **Image Loading and RGB Conversion**:
    - The cv2.imread function reads the image from the provided path, loading it in BGR format (OpenCV's default).
    - cv2.cvtColor(image, cv2.COLOR_BGR2RGB) converts the image to RGB format for consistent color display in Matplotlib.
2. **Grayscale Conversion**:
    - cv2.cvtColor(image, cv2.COLOR_BGR2GRAY) converts the RGB image to grayscale, which is used for edge detection and other processing steps.
3. **Image Resizing**:
    - cv2.resize(image, (0, 0), fx=0.5, fy=0.5) scales down the image to half its original size for faster processing during the bilateral filtering.
4. **Bilateral Filtering**:
    - The bilateral filter is applied iteratively to smooth the image while preserving edges. This step creates a smooth base that resembles a cartoon style.
    - After filtering, the image is resized back to its original dimensions to match the edges later in the process.

5. **Edge Detection**:
    - A median blur is applied to the grayscale image to remove noise.

- cv2.Laplacian(smoothed_image, cv2.CV_8U, ksize=5) is used to perform edge detection, creating a map of edges that will define the cartoon "outlines".
- A binary thresholding operation is then applied to the edge map, converting it to a mask of clear black edges.

6. **Cartoon Image Creation**:
   - The edges are combined with the bilateral-filtered image using a bitwise AND operation. This produces the final cartoon effect by combining smooth colors with pronounced edges.
7. **Visualization**:
   - The code generates a series of subplots for each step in the process, displaying intermediate outputs to illustrate how the final cartoon effect is created.

Results

1. Original Grayscale and Smoothed Grayscale

- **Original Grayscale** shows the standard grayscale conversion of the input image.
- **Smoothed Grayscale** reduces noise and enhances the softer details while keeping the facial features recognizable.

2. Edge Detection

- The **Edge Detection Output** from the Laplacian operator highlights prominent edges, making a high-contrast outline suitable for cartoon-style borders.
- **Output After Thresholding** refines the edge map, making the lines crisp by converting soft edges to solid black lines on a white background.

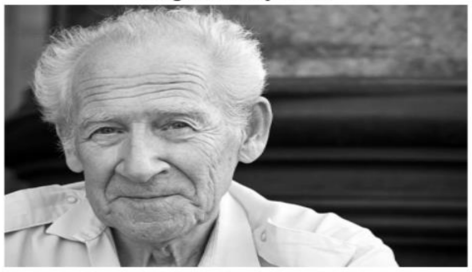
3. Bilateral Filtered Image

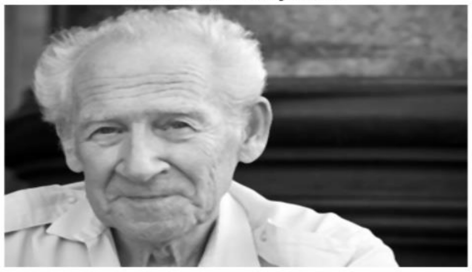
- **Output from Bilateral Filter** is a smooth, low-detail version of the image. The bilateral filter removes fine texture while retaining color boundaries, which is key to the cartoon style.

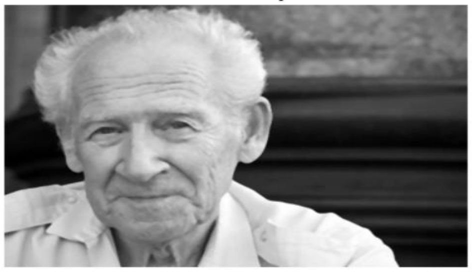
4. Final Cartoonified Image

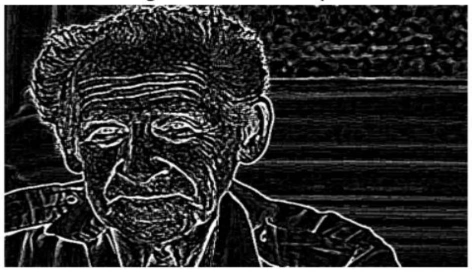
- The **Final Output** combines the smooth color regions from the bilateral filter with the crisp edge outlines. This fusion produces the "cartoonified" effect, where the subject appears as a simplified, high-contrast illustration.

Test Images and Results:

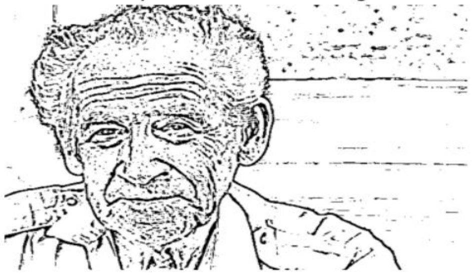

Original Grayscale

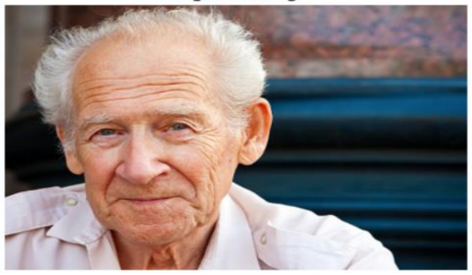

Smoothed Grayscale

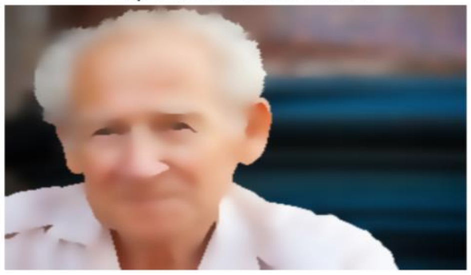

Smoothed Grayscale


Edge Detection Output

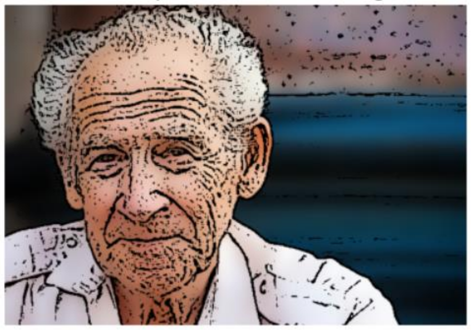
Edge Detection Output


Output After Thresholding


Original Image

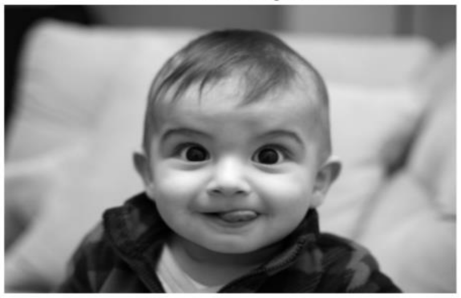

Output from Bilateral Filter

Output from Bilateral Filter

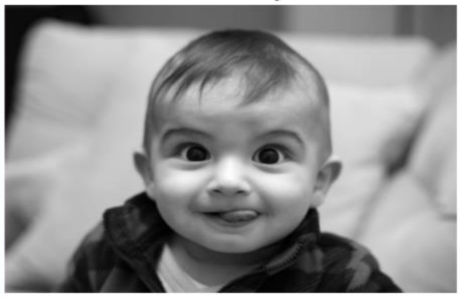

Final Output (Cartoonified Image)

Original Grayscale

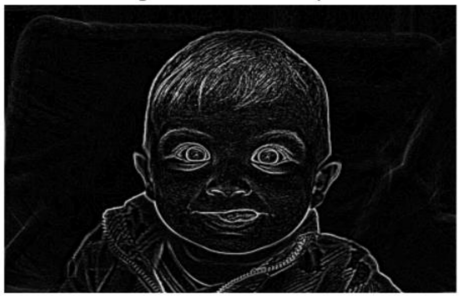
Smoothed Grayscale



Smoothed Grayscale

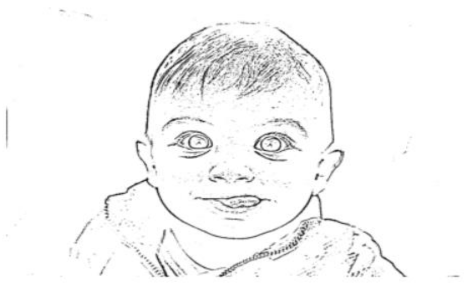


Edge Detection Output

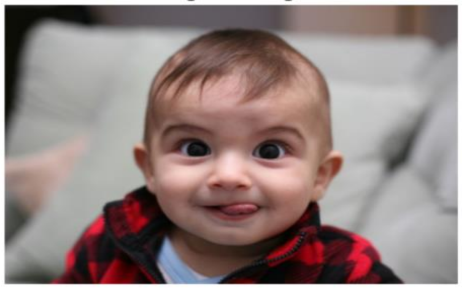


Edge Detection Output

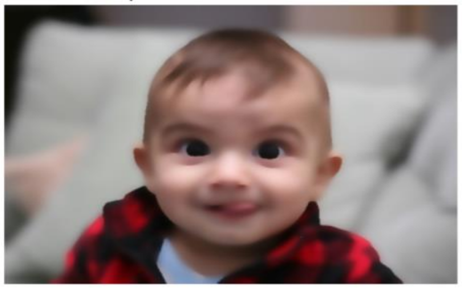
Output After Thresholding



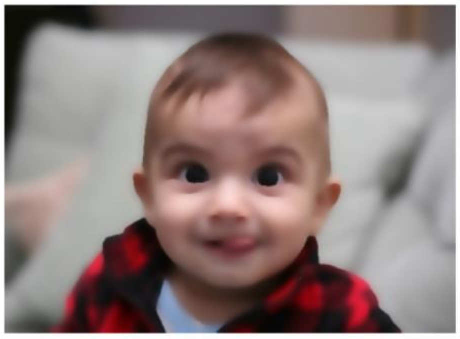
Original Image



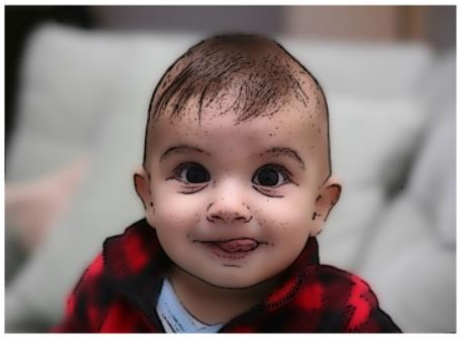
Output from Bilateral Filter



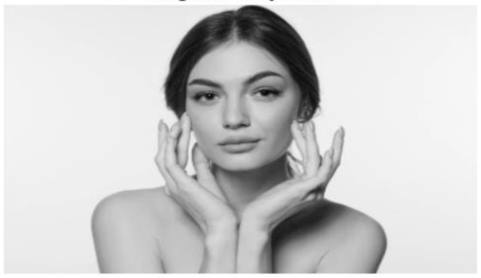
Output from Bilateral Filter



Final Output (Cartoonified Image)



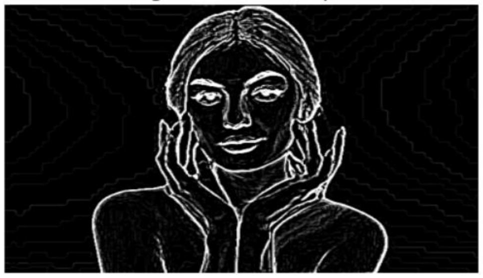
Original Grayscale



Smoothed Grayscale

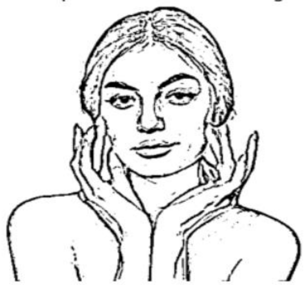
Smoothed Grayscale

Edge Detection Output

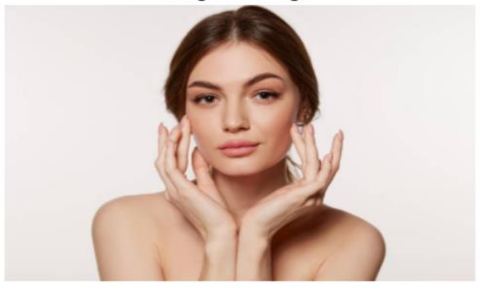


Edge Detection Output

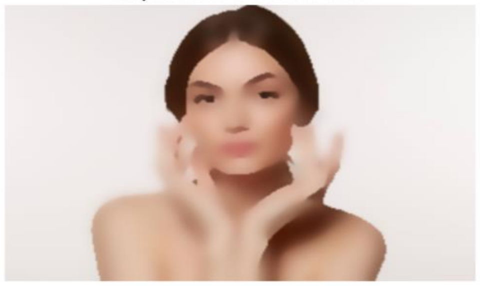
Output After Thresholding



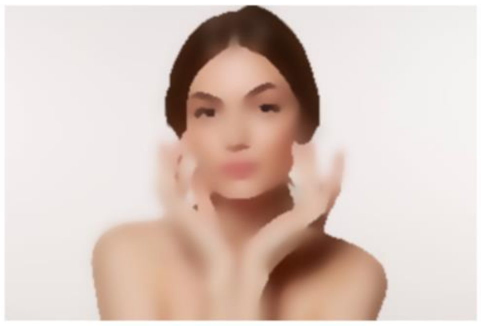
Original Image



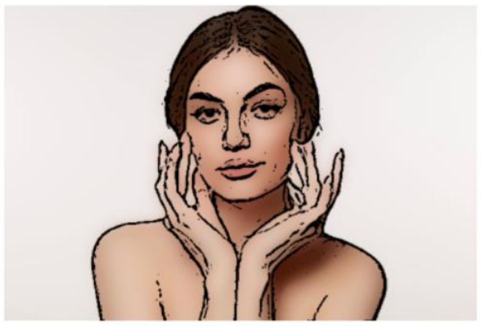
Output from Bilateral Filter



Output from Bilateral Filter



Final Output (Cartoonified Image)



Part II: Road Lane Detection Using Hough Transform:

Code Explanation

1. Importing Libraries

- cv2: Used for image processing, including reading, smoothing, edge detection, and line drawing.
- numpy: Used for array operations, including creating masks and defining polygon points.
- matplotlib.pyplot: Used to display intermediate images at various processing stages.

2. Image Loading and Initial Checks

- The image is loaded using cv2.imread(). If the image path is incorrect or the image fails to load, it prints an error message.

3. Preprocessing the Image

- **Smoothing**: The image is smoothed using cv2.medianBlur(), which applies a median filter with a kernel size of 5 to reduce noise.
- **Grayscale Conversion**: The smoothed image is converted to grayscale, simplifying the image by removing color information.
- **Edge Detection**: The Canny edge detector (cv2.Canny()) is used to identify edges with thresholds of 100 and 200, tuning sensitivity to highlight strong edges.

4. Region of Interest (ROI) Masking

- A polygon is defined to approximate the area of interest where lane lines are typically found.
- A mask is created using np.zeros_like(edges), and cv2.fillPoly() fills the polygon in white, keeping the rest of the mask black. This polygon masks the area where lane lines are most likely to appear.
- cv2.bitwise_and() is applied to keep only the edges within the masked region.

5. Manual Hough Transform

- **Accumulator Array Creation**:
    - The maximum possible rho value is calculated as the diagonal length of the image (rho_max), setting the bounds for possible distances.
    - An accumulator array is created with dimensions (180, 2 * rho_max). This array will count occurrences of (rho, theta) pairs.
- **Populating the Accumulator**:
    - Each edge point in the masked image is processed by iterating over potential theta values (0 to 180 degrees in radians).
    - For each (x, y) point with an edge, the rho value is calculated as x*cos(theta) + y*sin(theta), shifted by rho_max to avoid negative indices.
    - This (rho, theta) pair is then incremented in the accumulator.

- **Non-Maximum Suppression (NMS)**

  Non-Maximum Suppression (NMS) is an essential step added to refine the results obtained from the Hough Transform by retaining only the most significant peaks in the accumulator array, which correspond to the most prominent line detections.

  NMS aims to filter out weaker line responses, ensuring that only the strongest detections remain in the accumulator. This reduces noise and helps focus on the most relevant lines for lane detection.

- **Drawing Lines**:
  - For each cell in the accumulator that exceeds the line threshold (set to 100), a line is drawn.
  - Using `theta` and `rho`, the start and endpoints of the line are calculated with a specified line length of 1000 pixels, and the line is drawn with `cv2.line()` in green on `line_image`.

6. Displaying Results

- The images are displayed in a grid using `matplotlib.pyplot`:
  - **Original Image**: Shows the initial, unprocessed image.
  - **Smoothed Image**: Displays the image after applying median blur.
  - **Edges**: Displays the edges detected by the Canny algorithm.
  - **Masked Edges (ROI)**: Shows edges within the specified ROI.
  - **Line Detection**: Displays the final result with detected lines overlaid on the image.

7. Final Observations

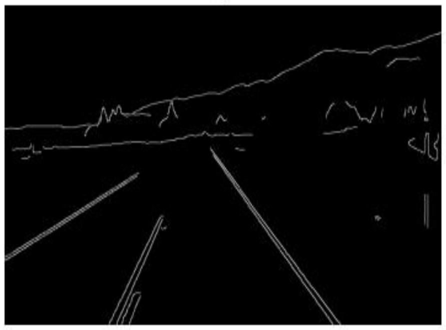
- This code performs lane detection using a custom, manual Hough Transform approach. This method provides flexibility, although it may be computationally intensive.
- The ROI mask helps focus the Hough Transform on relevant areas, improving accuracy by reducing noise from irrelevant regions.

Test Image and Results

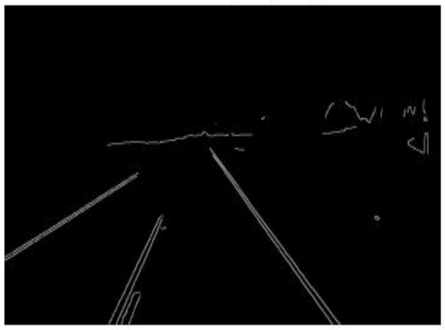
Original image

smoothed image

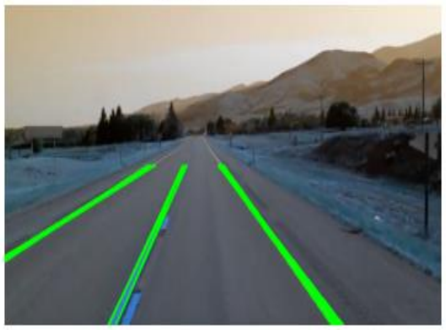
Edges



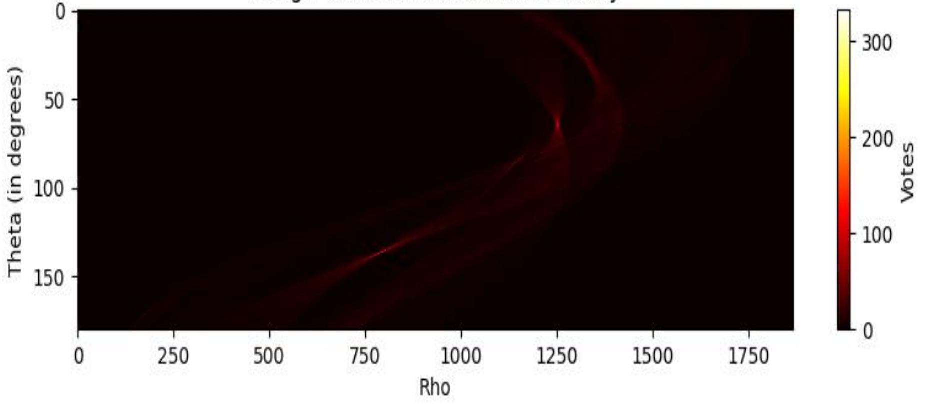
Masked Edges (ROI)



line detection



Hough Transform Accumulator Array


Theta (in degrees)
Rho
Votes