

UCS2523 - Image Processing and Analysis

Assignment - I

Thirumurugan RA
Department of Computer Science and Engineering
Sri Sivasubramaniya Nadar College of Engineering
Chennai, India
thirumurugan2310277@ssn.edu.in

I. INTRODUCTION

This document aims to provide a comprehensive understanding of the digital image processing pipeline.

II. IMAGE ACQUISITION



Fig. 1: This is an image of objects in a table

This image was captured with a **smartphone**

A. Justification of choice of image

- The scene has proper lighting and it also has a good dynamic range of lighting.
- There are many objects in the image - the notebooks, the cloth, the remote, etc. with well defined edges.
- Each object in the image also has their own textures.
- There is also a good foreground (objects) and background (table) distinction since the

III. NOISE SIMULATION

The python libraries and modules used throughout this assignment are: OpenCV, Scikit-Image, Python Imaging Library (PIL), Numpy

The initialization code is given below:

```
1 import cv2
2 from skimage.util import random_noise
3 from PIL import Image
4 from IPython.display import display
5 import numpy as np
6 import matplotlib.pyplot as plt
7
8 img_cv = cv2.imread('Table.jpg')
```

The noises we add are Gaussian noise and salt and pepper noise. The Python code to add these noises to the original image is given below.

```
1 #Add gaussian noise
2 gaussian_noise = random_noise(img_cv, mode='
3 gaussian_img = np.array(255 * gaussian_noise,
4 cv2.imwrite("gaussian_noise_color.jpg",
5 display(Image.fromarray(gaussian_img))
```

```
1 #Add salt and pepper noise
2 salt_pepper_noise = random_noise(img_cv, mode=
3 salt_pepper_img = np.array(255 *
4 cv2.imwrite("salt_and_pepper_noise_color.jpg",
5 display(Image.fromarray(salt_pepper_img))
```

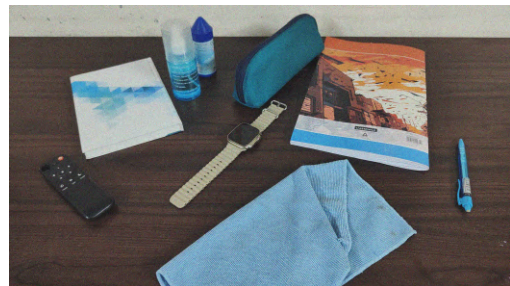


Fig. 2: Image with Gaussian Noise

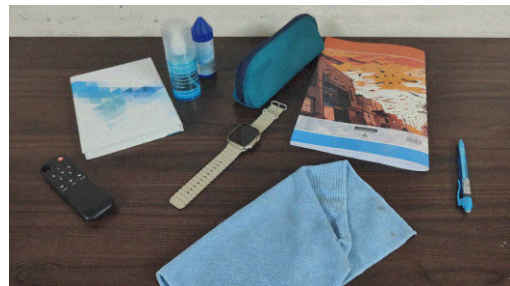


Fig. 3: Image with Salt and Pepper Noise

A. Nature and Impact of Gaussian Noise and Salt-and-Pepper Noise

1) Gaussian Noise:

Nature:

- Gaussian noise refers to random variations in intensity values, drawn from a Gaussian distribution with mean μ and variance σ^2 .
- Each pixel in the image is modified by adding a small random value.

Impact:

- Produces a fine **grainy** texture across the entire image.
- Reduces contrast and blurs edges slightly.
- Degrades overall image quality in a uniform manner.

2) Salt and Pepper Noise:

Nature:

- Salt-and-pepper noise (impulse noise) randomly replaces some pixels with the minimum (black, intensity = 0) or maximum (white, intensity = 255) value.
- The result appears as scattered white and black dots in the image.

Impact:

- Introduces sharp, high-contrast disturbances in the image.
- Strongly degrades edges, fine details, and textures.
- More visually distracting compared to Gaussian noise.

We will continue to use the image with Gaussian noise, since this is the noise that occurs in electronic sensors used in digital cameras

IV. PREPROCESSING AND ENHANCEMENT

The first step in pre-processing is to convert the image with Gaussian noise into grayscale. The code to do that is given below

```
1 # Convert to grayscale
2 gray_img = cv2.cvtColor(gaussian_img, cv2.
   COLOR_RGB2GRAY)
3 cv2.imwrite("grayscale.jpg", gray_img)
4 display(Image.fromarray(gray_img))
```



Fig. 4: Grayscale image with Gaussian Noise

The second step is to resize the grayscale image.

```
1 # Resize gray_img to 512x512
2 gray_img_resize = cv2.resize(gray_img, (512,
   512))
3 print(gray_img_resize.shape)
4 cv2.imwrite("grayscale_resized.jpg",
   gray_img_resize)
5 display(Image.fromarray(gray_img_resize))
```

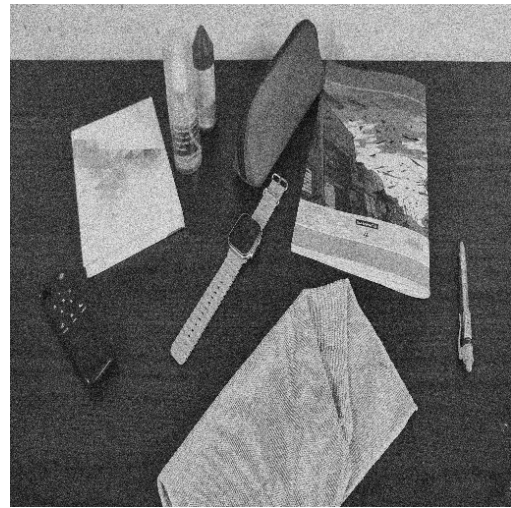


Fig. 5: Resized image

A. Pre-processing techniques justification

- **Conversion to Grayscale:** This is done so that the pre-processing becomes easier. There is no useful data in colour images that help in segmentation, object isolation and feature evaluation.
- **Image Resizing:** We resize the image to a standard size (eg: 512x512) to increase efficiency since computing larger images would take longer time. Resizing also ensures consistency while comparing from a dataset.

Now we can do contrast stretching to enhance the image

```
1 # Apply contrast stretching
2 p2, p98 = np.percentile(gray_img_resize.astype
   (np.float64), (2, 98))
3 contrast_stretched = np.zeros_like(
   gray_img_resize, dtype=np.uint8)
4 cv2.normalize(gray_img_resize,
   contrast_stretched, 0, 255, cv2.
   NORM_MINMAX, dtype=cv2.CV_8U)
5 contrast_stretched = np.clip((gray_img_resize
   - p2) * 255.0 / (p98 - p2), 0, 255).astype
   (np.uint8)
6 cv2.imwrite("contrast_stretched.jpg",
   contrast_stretched)
7 display(Image.fromarray(contrast_stretched))
```

B. Enhancement technique justification

- **Contrast Stretching:**
 - It is an enhancement technique to improve global contrast by linearly rescaling the pixel intensity values. The chosen percentiles (here 2% and 98%) are mapped to 0 and 255 respectively, and all other values are proportionally stretched within this range
 - By applying this we make dark areas darker and bright areas brighter, thereby enhancing visibility of details and textures that may otherwise be obscured.



Fig. 6: Contrast Stretched image

V. NOISE FILTERING AND DENOISING

We will be applying 2 different spatial domain filters - Median filter and Gaussian filter

A. Gaussian Filter

It works by averaging pixel values with their neighbors, but the averaging is weighted according to a Gaussian distribution, so the pixel closer to the center will get more importance than the ones farther.

```
1 # Gaussian Filter
2 gaussian_filter_cs = cv2.GaussianBlur(
3     contrast_stretched.astype(np.uint8), (3,
4     3), 1)
5 cv2.imwrite("gaussian_filter.jpg",
6     gaussian_filter_cs)
7 display(Image.fromarray(gaussian_filter_cs))
```

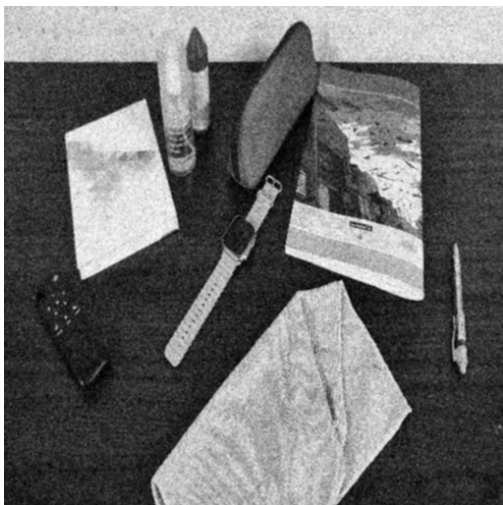


Fig. 7: Image after applying Gaussian Filter

B. Median Filter

It works by replacing each pixel value with the median of the intensity values in its neighborhood. This makes it highly effective in removing salt-and-pepper noise, as the median is less affected by outliers compared to the mean.

```
1 # Median Filter
2 median_filter_cs = cv2.medianBlur(
3     contrast_stretched.astype(np.uint8), 3)
4 cv2.imwrite("median_filter.jpg",
5     median_filter_cs)
6 display(Image.fromarray(median_filter_cs))
```

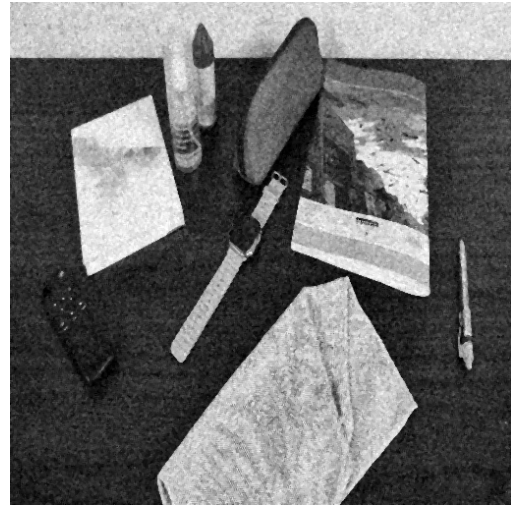


Fig. 8: Image after applying Median Filter

C. Evaluation and Comparison of Performance

Qualitative Comparison:

- **Median Filter:** It managed to reduce some of the noise, but since Gaussian Noise is spread throughout the image, median filter was not effective. The output was slightly cleaner, but many fine details and textures were distorted.
- **Gaussian Filter:** It was much better in handling Gaussian noise. It smoothed out the graininess while still preserving edges and details. A mild blur was there, but still the image was more natural looking than the median filter.

Quantitative Comparison: We will use PSNR (Peak Signal-to-Noise Ratio) and SSIM (Structural Similarity Index Measure) to do Quantitative Comparison of both the filters. First we will look into PSNR Calculation

```
1 # PSNR Calculation
2 psnr_median_cs = cv2.PSNR(contrast_stretched,
3     median_filter_cs)
4 psnr_gaussian_cs = cv2.PSNR(contrast_stretched
5     , gaussian_filter_cs)
6 print(f'PSNR (Gaussian): {psnr_gaussian_cs:.2f} dB')
7 print(f'PSNR (Median): {psnr_median_cs:.2f} dB')
```

Results of PSNR Calculation:

- **PSNR (Gaussian):** 20.10 dB
- **PSNR (Median):** 19.05 dB

SSIM Calculation

```
1 # SSIM Calculation
2 from skimage.metrics import
   structural_similarity as ssim
3 ssim_median_cs = ssim(contrast_stretched,
   median_filter_cs, data_range=255)
4 ssim_gaussian_cs = ssim(contrast_stretched,
   gaussian_filter_cs, data_range=255)
5 print(f'SSIM (Gaussian): {ssim_gaussian_cs:.2f}')
6 print(f'SSIM (Median): {ssim_median_cs:.2f}')
```

Results of SSIM Calculation:

- **SSIM (Gaussian):** 0.46
- **SSIM (Median):** 0.32

Conclusion: Gaussian filter is a better choice because it preserves details and reduces noise. The PSNR and SSIM score for Gaussian filter is also higher compared to Median filter

VI. SEGMENTATION AND OBJECT ISOLATION

We have chosen Edge based technique - Canny Edge Detection.

A. Segmentation method Justification

- **Noise Robustness:** The algorithm applies a Gaussian filter first, which smooths the image and reduces noise, preventing false edge detection.
- **Accurate Edge Linking:** It uses double thresholding and hysteresis. This allows it to detect strong edges and then it keeps weak edges only if they are connected to the strong edges.

This is the code to apply Canny Edge Detection method

```
1 processed_img = gaussian_filter_cs.copy()
2 original_color = cv2.imread('Table.jpg')
3
4 # Apply Canny edge detection
5 canny_edges = cv2.Canny(processed_img,
   threshold1=80, threshold2=290)
6
7 # Display Canny edge result
8 cv2.imwrite("canny_edges.jpg", canny_edges)
9 print("Canny Edge Detection Result (Edges):")
10 display(Image.fromarray(canny_edges))
```

Now we will look into a thresholding based technique, Otsu Thresholding, to compare the 2 and find the best one.

This is the code to apply Otsu thresholding

```
1 processed_img = gaussian_filter_cs.copy()
2 original_color = cv2.imread('Table.jpg')
3
4 # Apply Otsu's thresholding
5 ret, otsu_thresh = cv2.threshold(
6   processed_img, 0, 255, cv2.THRESH_BINARY +
   cv2.THRESH_OTSU
7 )
```

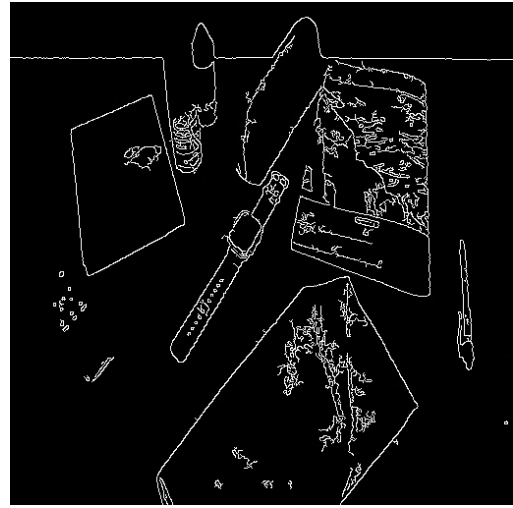


Fig. 9: Image after applying Canny Edge Detection

```
8 # Display Otsu's threshold result
9 cv2.imwrite("otsu_threshold.jpg", otsu_thresh)
10 print("Otsu's Thresholding Result:")
11 display(Image.fromarray(otsu_thresh))
```

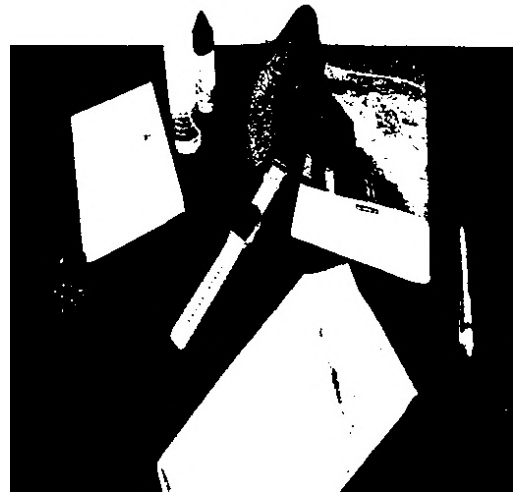


Fig. 10: Image after applying Otsu Thresholding

B. Critical Analysis

For this task, Canny's Edge Detection Algorithm was better for detecting all objects in the image. Otsu's thresholding method failed to find any object that was a similar color to the background, while Canny's method identified the boundaries of almost all objects.

- **Strengths:** Found the outlines of dark objects - specifically the remote, pencil case, and blue pen. It effectively captured the high-frequency details of the folded blue cloth, distinguishing its edges from the table.
- **Weaknesses:** The algorithm's strengths in detection are also its primary weakness for the next step (feature

extraction). The primary weakness is that this method does not perform object isolation. It produces hollow outlines, not solid "masks" of the objects.

VII. FEATURE EXTRACTION AND EVALUATION

- **Area:** It is the total number of pixels inside the object's boundary. It is a simple way to classify objects by size.
- **Centroid:** This is the geometric center of an object, like its center of mass. It's represented as an (x, y) coordinate. It is a simple way to find the object's location.
- **Colour Histogram:** This is a graph that shows the distribution of colors in an object. It counts how many pixels of each color are present. It is a simple and powerful method for object recognition

The code for feature extraction and evaluation is given below

```
1 original_image = cv2.imread('Table.jpg')
2 canny_image = cv2.imread('canny_edges.jpg',
3   cv2.IMREAD_GRAYSCALE)
4 if original_image is not None and canny_image
5   is not None:
6     original_resized = cv2.resize(
7       original_image, (512, 512))
8
9     contours, hierarchy = cv2.findContours(
10      canny_image,
11      cv2.RETR_EXTERNAL,
12      cv2.CHAIN_APPROX_SIMPLE
13    )
14    print(f"Total contours found (before
15      filtering): {len(contours)}")
16
17    min_area = 150
18    filtered_contours = [cnt for cnt in
19      contours if cv2.contourArea(cnt) >
20      min_area]
21    print(f"Contours after filtering (area > {
22      min_area}): {len(filtered_contours)}")
23
24    filtered_contours.sort(key=cv2.contourArea
25      , reverse=True)
26
27    top_contours = filtered_contours[:6]
28    print(f"Selected Top {len(top_contours)}
29      largest contours.")
30
31    top_6_features = []
32    print("\n--- Feature Table for Top 6
33      Objects ---")
34    print("=" * 45)
35    print(f"{'Object ID':<12} | {'Area (pixels
36      )':<15} | {'Centroid (X, Y)':<15}")
37    print("-" * 45)
38
39    for i, cnt in enumerate(top_contours):
40      area = cv2.contourArea(cnt)
41
42      M = cv2.moments(cnt)
43      cx, cy = 0, 0
44      if M["m00"] != 0:
45        cx = int(M["m10"] / M["m00"])
46        cy = int(M["m01"] / M["m00"])
```

```
centroid_str = f"({cx}, {cy})"
print(f"{i + 1:<12} | {area:<15.0f} |
      {centroid_str:<15}")

mask = np.zeros(original_resized.shape
[:2], dtype="uint8")
cv2.drawContours(mask, [cnt], -1, 255,
-1)

hist_b = cv2.calcHist([
  original_resized], [0], mask,
[256], [0, 256])
hist_g = cv2.calcHist([
  original_resized], [1], mask,
[256], [0, 256])
hist_r = cv2.calcHist([
  original_resized], [2], mask,
[256], [0, 256])

features = {
  "id": i + 1,
  "area": area,
  "hist_b": hist_b,
  "hist_g": hist_g,
  "hist_r": hist_r
}
top_6_features.append(features)

print("=" * 45)

print("\nPlotting Color Histograms for Top
6 Objects...")

fig, axes = plt.subplots(3, 2, figsize
=(18, 15))
fig.suptitle('Colour Histograms for the 6
Largest Detected Objects (from Canny)'
, fontsize=20, fontweight='bold')

ax = axes.flatten()

for i, item in enumerate(top_6_features):
  ax[i].plot(item['hist_b'], color='blue')
  ax[i].plot(item['hist_g'], color='green')
  ax[i].plot(item['hist_r'], color='red')

  ax[i].set_title(f"Object {item['id']}:
    Colour Histogram (Area: {item['
    area']:.0f})")
  ax[i].set_xlabel("Pixel Intensity")
  ax[i].set_ylabel("Number of Pixels")
  ax[i].set_xlim([0, 256])

plt.tight_layout(rect=(0, 0.03, 1, 0.96))
plt.savefig('
canny_color_histograms_grid_top6.jpg',
dpi=150)
print("Histogram grid saved as '
canny_color_histograms_grid_top6.jpg'")
plt.show()

result_image = original_resized.copy()
```

```

82 cv2.drawContours(result_image,
83                 filtered_contours, -1, (0, 255, 0), 1)
84
85 print("\nLabeling Top 6 Objects on image:")
86 )
87 for i, cnt in enumerate(top_contours):
88     M = cv2.moments(cnt)
89     cx, cy = 0, 0
90     if M["m00"] != 0:
91         cx = int(M["m10"] / M["m00"])
92         cy = int(M["m01"] / M["m00"])
93
94     label = str(i + 1)
95     cv2.putText(result_image, label, (cx -
96                 10, cy + 10),
97                 cv2.FONT_HERSHEY_SIMPLEX,
98                 0.8,
99                 (0, 0, 255),
100                 2)
101
102 cv2.imwrite("
103         canny_contours_numbered_result.jpg",
104         result_image)
105 print("\nResult saved as '
106         canny_contours_numbered_result.jpg'")
107
108 print("Displaying final numbered image:")
109 display(Image.fromarray(cv2.cvtColor(
110     result_image, cv2.COLOR_BGR2RGB)))

```



Fig. 11: Image after finding the contours

Object	Area (pixels)	Centroid (x, y)
1	74501	(279, 215)
2	1350	(462, 317)
3	1276	(285, 384)
4	331	(61, 310)
5	274	(92, 370)
6	198	(220, 440)

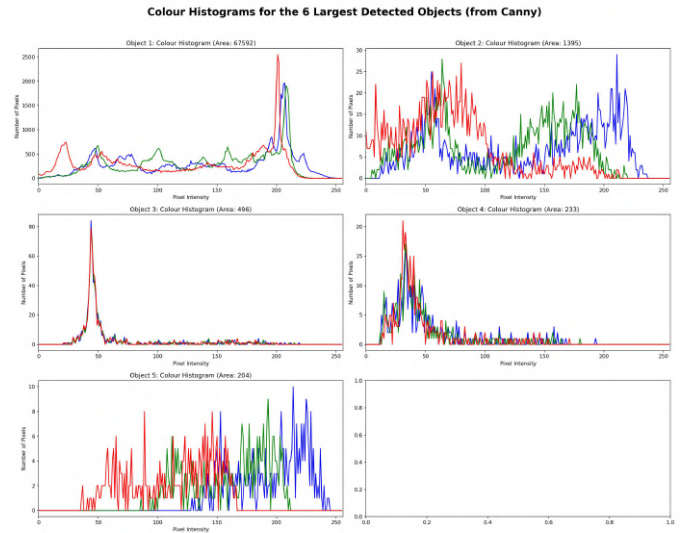


Fig. 12: Colour histograms for the 6 largest detected objects

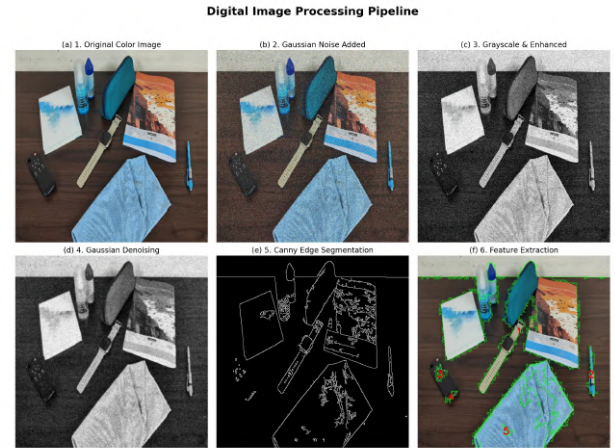


Fig. 13: Digital Image Processing Pipeline

VIII. RESULT VISUALIZATION AND REFLECTION

A. Digital Image Processing Pipeline

This figure displays the progression of the image from its original state through all major processing steps.

B. Evaluation of the Complete Pipeline

1) Areas of Success:

- **Noise Filter Comparison:** The quantitative comparison between the Gaussian and Median filters was a clear success. The Gaussian filter was proven to be the superior choice for handling Gaussian noise, not just visually, but also with higher **PSNR (20.10 dB vs 19.05 dB)** and **SSIM (0.46 vs 0.32)** scores.
- **Edge-Based Segmentation:** The Canny edge detector was far more effective than Otsu's thresholding. Canny successfully identified the boundaries of almost all objects, including the dark remote and pen, which the global Otsu method failed to find.

- **Feature Extraction:** The chosen features were effective for analysis. **Area** provided a simple way to classify objects by size, and the **Centroid** gave a precise (x, y) coordinate for each object's location.

2) Areas of Improvement:

- **Thresholding Failure:** The Otsu's thresholding method was a failure for this image. It uses a single global threshold, which cannot separate dark objects from a dark background. An improvement would be to use **Adaptive Thresholding**, which calculates local thresholds for different regions of the image.
- **"Hollow" Contours:** The primary weakness of the Canny method is that it "produces hollow outlines, not solid 'masks'". This makes direct feature extraction difficult. A key improvement would be to apply **morphological operations** to fill these outlines and create solid masks.
- **Canny's Sensitivity:** The Canny detector was sensitive, picking up high-frequency details from the cloth and notebook art. This created many small, "noisy" contours that had to be filtered out by area. An improvement would be to experiment with a slightly stronger Gaussian blur before Canny to smooth these textures.

3) Learnings:

- **No "One-Size-Fits-All" Algorithm:** The most important lesson was that there is no single "best" method. The Gaussian filter was best for Gaussian noise, but the Median filter is designed to be highly effective against salt-and-pepper noise. Canny succeeded at detection where Otsu failed. The choice of algorithm must match the specific problem.
- **Quantitative Proves Qualitative:** It was one thing to see that the Gaussian filter looked better, but it was much more powerful to prove it with higher PSNR and SSIM scores. This shows that both visual (qualitative) and numerical (quantitative) analysis are required for a complete evaluation.
- **The Pipeline Order is Critical:** The steps are not independent. The decision to run Canny detection on the denoised image was crucial, as Canny's first step is also a Gaussian filter. This shows that a successful pipeline is built by ensuring the output of one step is the ideal input for the next.

The code can be found in this GitHub Repository <https://github.com/thirumuruganra/ipa-assignment>