Lab Instructions - session 4

Noise Modeling and Filtering

1. Image Noise Models and Simulation

Digital images are often corrupted by **noise**, which is any unwanted random variation in pixel values . Noise can arise during image capture (sensor noise, photon statistics), transmission (bit errors), or processing. We commonly model noise as a random process with a specific probability distribution. Three widely-used noise models in imaging are:

Gaussian Noise (additive white Gaussian noise): Noise values follow a normal distribution. This is a typical assumption for sensor read-out noise or thermal noise in images. It is independent of the image signal and affects each pixel independently (often modeled as zero-mean with some variance σ^2).

Salt-and-Pepper Noise (snow noise): Random pixels are flipped to black or white (extreme values), like sprinkling salt and pepper on the image . This typically arises from sharp, sporadic disturbances like faulty sensor elements or transmission errors. It's characterized by sparsely occurring bright and dark pixel defects amidst normal pixels. Only a certain fraction p of pixels are corrupted.

Task 1: Simulating Noise Addition

In this task, you will generate synthetic noise and add it to an image to observe the effects of different noise models. We will use OpenCV (cv2) and NumPy for this.

Mathematical Notes: Salt-and-pepper can be described as a *bimodal* distribution: $P(n=0)=p/2$, $P(n=255)=p/2$, $P(n=\text{no change})=1-p$. You can use np.random.choice for this purpose.

lab4-task1.py

```python
import cv2
import numpy as np

# Load image in grayscale and convert to float [0,1]
I = cv2.imread('lenna.png', cv2.IMREAD_GRAYSCALE)
I = I.astype(np.float32) / 255.0  # shape: (H, W)

# Function to add Gaussian noise
def add_gaussian_noise(img, sigma=0.05):
    # TODO: Generate a noise array using np.random.randn scaled by sigma.
    # Example: noise = np.random.randn(*img.shape) * sigma
    # TODO: Add the noise to the input image.
    # TODO: Clip the resulting values to ensure they remain in the [0,1] range.
    pass

# Function to add salt-and-pepper noise
def add_salt_pepper_noise(img, p=0.02):
    # TODO: Create a copy of the image to modify.
    # TODO: Determine the number of pixels to alter based on the given p
    # TODO: Randomly choose indices for salt (set to 1.0) and pepper (set to 0.0).
    pass

# Generate noisy images using your implementations
gauss_noisy = add_gaussian_noise(I, sigma=0.1)
sp_noisy = add_salt_pepper_noise(I)

# Convert the noisy images back to uint8 for saving or displaying
cv2.imwrite('noisy_gaussian.png', (gauss_noisy * 255).astype(np.uint8))
cv2.imwrite('noisy_saltpepper.png', (sp_noisy * 255).astype(np.uint8))
```

- Vary the noise parameters (`sigma` and p) and observe how the image degradation changes. For Gaussian noise, how does increasing σ affect the image?
- Plot or examine the histogram of the noise for each case to verify it matches the expected distribution.
- Mathematically, if you add two independent Gaussian noises with variances σ_1^2 and σ_2^2 to an image, what is the distribution of the combined noise? What would be its variance?

Task 2: Dynamic "Snow" Noise Simulation

Older analog televisions displayed a dynamic **snow noise** (salt and paper noise) when tuned to a missing channel – essentially random black/white pixel patterns varying over time (this is like salt-and-pepper or Gaussian noise that changes every frame). In this task, you will create an animation of an image corrupted by continually changing noise, and allow user interaction to control noise intensity.

Instructions: Using OpenCV, write a script that:

- Reads an image in grayscale and normalizes it to [0,1] float.
- Continuously in a loop, adds a newly generated Gaussian noise array to the image to produce a noisy frame, and displays it (e.g., using `cv2.imshow`).
- Each loop iteration should use a **different noise realization** (so the noise pattern moves every frame).
- Run at ~30 frames per second (`cv2.waitKey(33)` in the loop).
- Use keyboard controls:
  - Press `'u'` to **increase** the noise standard deviation (make the image noisier).
  - Press `'d'` to **decrease** the noise standard deviation (make it cleaner).
  - Press `'q'` to quit the loop.

You can start from the template below and fill in the missing parts (`pass` statements):

lab4-task2.py

```python
import numpy as np
import cv2

# Load the image in grayscale and normalize to [0,1]
I = cv2.imread('cameraman.jpg', cv2.IMREAD_GRAYSCALE)
I = I.astype(np.float32) / 255.0  # Ensure pixel values are in [0,1]

noise_sigma = 0.05  # initial noise standard deviation

while True:
    # TODO: Create a noise image N using a Gaussian distribution with mean 0 and
variance noise_sigma^2.
    # Hint: Use np.random.randn with the shape of I and multiply by noise_sigma.
    N = np.random.randn(*I.shape) * noise_sigma  # complete if needed

    # TODO: Add the noise to the original image and clip the result to ensure
values remain in [0,1].
```

```
    cv2.imshow('Snow Noise', J)

key = cv2.waitKey(33) & 0xFF

    # TODO: Adjust noise_sigma based on key input:
    if key == ord('u'):

    elif key == ord('d'):

    elif key == ord('q'):
        break

cv2.destroyAllWindows()
```

- What does normalizing the image to [0,1] (the line with `I.astype(np.float32)/255.0`) accomplish in terms of noise addition?
- Why should the noise image be regenerated inside the loop instead of outside? What happens if you create N once before the loop and reuse it every frame?
- Ensure your code never uses a negative `noise_sigma`. Why is a negative standard deviation meaningless for noise generation?

2. Filtering Fundamentals: Spatial Convolution and Frequency Perspective

Now that we can produce noisy images, the next step is to **filter** them to recover a cleaner image. Smoothing filters work by averaging pixels with their neighbors, reducing variance due to noise. The most basic way to do this is via **convolution** with a smoothing kernel.

Image Convolution: In 2D, convolution of an image $I(x,y)$ with a kernel $h(x,y)$ (of size m×n) produces an output $J(x,y)$ given by:

$$J(x,y) = \sum_{u=-a}^{a} \sum_{v=-b}^{b} h(u,v)\, I(x-u,\ y-v),$$

where the kernel size is $(2a+1)\times(2b+1)$ (e.g., for a 3×3 kernel, $a=b=1$). Each output pixel is a weighted sum of neighboring input pixels. If $h(u,v)$ are all positive and sum to 1, this operation smooths the image (local averaging). This is essentially a **low-pass filter** – it passes

low-frequency (smooth) variations and attenuates high-frequency components (like sharp noise or edges) .

2.1 Mean Filter (Box Filter)

The simplest smoothing kernel is a **box filter** – all ones in an $m\times m$ neighborhood, normalized bym^2. This gives an output pixel that is the average of an $m\times m$ patch of the image. For example, a 3×3 mean filter kernel is:$m\times m$ neighborhood, normalized bym^2. This gives an output pixel that is the average of an $m\times m$ patch of the image. For example, a 3×3 mean filter kernel is:$m\times m$ neighborhood, normalized bym^2. This gives an output pixel that is the average of an $m\times m$ patch of the image. For example, a 3×3 mean filter kernel is:

$$h = \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}.$$

Task 3: Implementing a Box Blur

Let's apply a box filter to a noisy image and see the result. OpenCV provides a function `cv2.blur(src, ksize)` that does this, or the more general `cv2.filter2D(src, ddepth, kernel)` for custom kernels. We can also create the kernel manually with NumPy.

Instructions:

- Load an image (you can use the noisy image from Task 1, or a clean image to just see the blurring effect).
- Create a box kernel of size `m×m` (choose `m`, e.g., 5 or 7). This kernel should have all values = 1/(m*m).
- Use `cv2.filter2D` to apply the kernel to the image. Compare with using `cv2.blur` for verification.
- Try different kernel sizes and observe the differences.

lab4-task3.py

```
 import numpy as np
import cv2

# Load an image (using the noisy image from Task 1, or a clean image) in
grayscale
I = cv2.imread('noisy_gaussian.png', cv2.IMREAD_GRAYSCALE)
I = I.astype(np.float32) / 255.0  # Normalize to [0,1]

m = 7  # Filter size (try 3, 5, 7, 11, etc.)

# === TODO: Create an m×m box filter kernel ===
# Use np.ones to create an array of ones and divide by (m*m) to normalize.
kernel = None  # <-- Student to fill in

# === TODO: Apply convolution to blur the image using cv2.filter2D ===
J = None  # <-- Student to fill in

# === TODO: Convert the result to uint8 and save or display it ===
# For example, use cv2.imwrite to save the result.
```

- Try `m=3, 5, 9, 15`. How does the choice of kernel size affect the output?
- Why do we divide the kernel by $(m*m)$? What would happen if we don't?
- Use OpenCV's `cv2.blur(I, (m,m))` to perform the same operation. Confirm it produces the same result as your filter2D approach.
- Suppose you apply two 3×3 mean filters sequentially (one after the other). Is the result different from a single 5×5 filter?

2.2 Gaussian Filter

A **Gaussian filter** uses a kernel shaped by the Gaussian (normal) distribution. In 1D, a Gaussian kernel of width m (odd) and standard deviation has values $G[i] = \frac{1}{\sqrt{2\pi\sigma}} \exp!\left(-\frac{i^2}{2\sigma^2}\right)$. In 2D, the kernel is the outer product of two 1D Gaussians (since a 2D Gaussian function is separable into x and y components). Gaussian filters give more weight to the center pixel and nearest neighbors, and less weight to farther pixels, according to the Gaussian curve. This tends to preserve central detail slightly better than a box filter for the same kernel size.

OpenCV has `cv2.GaussianBlur(src, ksize, sigmaX)` for this. Alternatively, we can build a Gaussian kernel manually or use `cv2.getGaussianKernel`.

Task 4: Implementing Gaussian Blur

We will create a Gaussian filter and apply it, then compare with the box filter results.

Instructions:

- Decide on a kernel size `m` (say 13) and optionally a σ. If σ is not specified, OpenCV will choose one based on `m` .
- Use `cv2.getGaussianKernel(m, sigma)` to get a 1D Gaussian kernel of length m. This returns an m×1 matrix.
- Compute a 2D Gaussian kernel by multiplying the 1D kernel with its transpose (to get an $m \# m$ kernel).
- Apply it with `cv2.filter2D`, or simply call `cv2.GaussianBlur`.
- Experiment with different m and σσ.

lab4-task4.py

```
import numpy as np
import cv2

# Load the noisy image in grayscale and normalize to [0,1]
I = cv2.imread('noisy_gaussian.png',
cv2.IMREAD_GRAYSCALE).astype(np.float32) / 255.0

m = 5  # Filter size (try different values, e.g., 3, 13, 21)

# === TODO: Create a 1D Gaussian kernel using cv2.getGaussianKernel ===
# Use sigma=0 to let OpenCV choose sigma automatically.
g1d = None  # <-- Student to complete

# === TODO: Create a 2D Gaussian kernel by taking the outer product of g1d
with its transpose ===
Gkernel = None  # <-- Student to complete

print("1D Gaussian kernel (m=%d):" % m, g1d.flatten())
print("2D Gaussian kernel sum:", Gkernel.sum())
```

```
# === TODO: Apply the Gaussian filter using cv2.filter2D (or
cv2.GaussianBlur) ===
J_gauss = None  # <-- Student to complete

# === TODO: Convert the result to uint8 and save or display it ===
```

- Observe the printed 1D Gaussian kernel values. Are they roughly centered and symmetric? Do they sum to 1? Try different kernel sizes `m`. Note that if you increase `m`, OpenCV's default σ also increases (roughly proportional to `m`). The blur gets stronger. You can also manually specify a fixed `sigma` independent of `m`.
- Compare the result of a 13×13 Gaussian filter with a 13×13 box filter on the same noisy image. Which one does a better job at reducing noise while preserving detail?

2.3 Median Filter (Non-Linear)

Not all filters are convolution-based linear filters. The **median filter** is a classic example of a non-linear filter that is very effective for **salt-and-pepper noise**. Instead of averaging, the median filter takes an m×m neighborhood and replaces the center pixel with the **median** value of those pixels. This way, outlier pixel values (extremely high or low, like salt or pepper noise) in the neighborhood are discarded in favor of a value that is more representative of the local surroundings.

Key property: A median filter can remove salt-and-pepper noise almost completely if the noise specks are smaller than the filter window, while preserving edges better than a mean filter. This is because an edge consists of a sharp transition – averaging would produce a new intermediate value (blurring the edge), whereas median will pick one of the existing values from either side of the edge, thus the edge (if not too thin) remains at least partially intact.

OpenCV's `cv2.medianBlur(src, ksize)` performs median filtering (ksize should be odd).

Task 5: Removing Salt-and-Pepper Noise with Median Filter

Instructions:

- Take a noisy image with salt-and-pepper noise (for example, `sp_noisy` from Task 1, after converting to uint8 if needed).
- Apply `cv2.medianBlur(sp_noisy, ksize)` with an odd kernel size (try 3, 5, or 7).
- Save or display the result and compare it to using a mean filter on the same input.

lab4-task5.py

```
import cv2
import numpy as np

# Assume sp_noisy_img is a uint8 grayscale image with salt-and-pepper
noise
sp_noisy_img = cv2.imread('noisy_saltpepper.png', cv2.IMREAD_GRAYSCALE)

# Apply median filter with a 5x5 kernel
denoised_med5 =
# Apply median filter with a 3x3 kernel for comparison
denoised_med3 =

cv2.imwrite('denoised_median5.png', denoised_med5)
cv2.imwrite('denoised_median3.png', denoised_med3)
```

- Compare the 3×3 vs 5×5 median filter results on salt-and-pepper noise. A larger window can eliminate larger noise spots, but might start to round off corners or thin lines. What is the smallest window size that completely removes the salt-and-pepper noise in your image?
- Try a mean (box) filter of size 3×3 or 5×5 on the same salt-and-pepper noisy image. Does it remove the noise effectively? What differences do you see compared to median filter?
- Why is the median filter particularly suited for impulse noise?

3. Edge-Preserving Filtering

Bilateral Filter

The **bilateral filter** is a popular edge-preserving smoother. It extends Gaussian smoothing by incorporating **intensity difference** into the weighting. In a bilateral filter, a pixel's neighbors that have a **similar intensity** to the pixel are given more weight than those with very different intensity (which are likely across an edge).

In formula form, for a pixel p, the bilateral filter output is:

$$Ifiltered(p) = \frac{1}{W_p} \sum_{q \in \Omega} I(q)\, f_r(\|I(q) - I(p)\|)\, g_s(\|q - p\|)$$

Parameters for the bilateral filter are typically:

- d (diameter of the filter kernel in pixels, or you can specify a spatial sigma instead),
- `sigmaColor`,
- `sigmaSpace`.

In OpenCV: `cv2.bilateralFilter(src, d, sigmaColor, sigmaSpace)` does the job.

Applying Bilateral Filter

bilateral.py

```
import cv2
import numpy as np

I = cv2.imread('lenna.png')  # load color image
I = I.astype(np.float32)/255.0

# Add noise for testing
noise = np.random.randn(*I.shape) * 0.05
noisy_color = np.clip(I + noise, 0.0, 1.0)

noisy_color_uint8 = (noisy_color * 255).astype(np.uint8)

# Bilateral filter parameters
d = 9  # diameter of pixel neighborhood (if set to >0). If =0, it uses
sigmaSpace to determine.
sigma_color = 0.1
sigma_space = 15

denoised_bilateral = cv2.bilateralFilter(noisy_color_uint8 , d,
sigma_color, sigma_space)
cv2.imwrite('denoised_bilateral.png',
(denoised_bilateral*255).astype(np.uint8))
```

- Change `sigma_color` higher (e.g., 0.3 or 0.5 on [0,1] scale). What happens to the image? (It will blur more aggressively, possibly washing out some edges as if it were closer to a normal Gaussian blur.)
- Change `sigma_space` higher (e.g., 50 pixels). What happens? (The filter considers a larger neighborhood, so it can smooth over wider areas, maybe connecting areas that shouldn't be connected if σ_color allows it.)
- Try a smaller `sigma_color` (like 0.05). Now noise might remain because the filter is very strict about intensity differences – it won't average pixels unless they are extremely similar.

- Compare bilateral filtering time vs a Gaussian of similar kernel size. Bilateral is computationally heavier (it’s doing a lot more work per pixel). For large images, bilateral can be slow.

4. Integrated Challenge

Task 6: Noise + Filter Demo

implement:

- Add a certain amount of Gaussian noise continuously (as in Task 2’s loop).
- Depending on user key press:
    - `'n'`: **No filtering** – just show the noisy image (baseline).
    - `'b'`: **Box filter** – apply averaging filter on the noisy image.
    - `'g'`: **Gaussian filter** – apply Gaussian blur on the noisy image.
    - `'m'`: **Median filter** – apply median filtering on the noisy image.
    - `'l'`: **Bilateral filter** (think **l** for bilateral) – apply bilateral filtering on the noisy image.
- Keys to adjust parameters:
    - `'+'` / `'-'`: Increase or decrease the kernel size `m` for box/gaussian/median filters (make sure to keep it odd and within a reasonable range).
    - `'.'` / `','` (period/comma): Increase or decrease σ_color for the bilateral filter (if relevant).
    - `'u'` / `'d'`: Increase or decrease the noise sigma.
- Display the filtered result in a window named e.g. `"Denoising Demo"`. Print out status messages or parameter values to the console when they change, so the user knows what's happening.
- `'q'` to quit.
- At low noise, any filter might make it look overly smooth – you might prefer mode 'n'.
- At moderate noise, see how median vs bilateral vs Gaussian compare, especially near edges or fine details.
- Find an “optimal” filter and parameter setting for a given noise level subjectively.
- Which filter produces the best visual result for Gaussian noise?
- If you use salt-and-pepper noise instead of Gaussian in this demo, which filter would you switch to?
- How does performance (speed) vary among the filters as you increase `m`?

References

1. **OpenCV: Smoothing Images**