2.1

def calculate\_pdf\_and\_cdf(img):

"""Calculate the PDF and CDF of an image."""

hist, \_ = np.histogram(img.flatten(), bins=256, range=(0, 256))

pdf = hist / np.sum(hist)

cdf = np.cumsum(pdf)

return pdf, cdf

**What’s happening here?**

* **Histogram** (hist): Counts of how many pixels have each intensity value (0 to 255).
  + img.flatten() turns the 2D image into a 1D array.
  + np.histogram(..., bins=256, range=(0,256)) counts pixels per intensity bin.
* **PDF (Probability Density Function)**:
  + The histogram values divided by the total number of pixels.
  + Represents the probability that a pixel has a certain intensity.
* **CDF (Cumulative Distribution Function)**:
  + Cumulative sum of the PDF.
  + CDF(r)=∑i=0rPDF(i)\text{CDF}(r) = \sum\_{i=0}^r \text{PDF}(i)CDF(r)=∑i=0r​PDF(i)
  + Gives the cumulative probability up to intensity r.
  + Important because **histogram equalization/matching** uses CDFs as the transformation functions.

def build\_mapping(cdf\_source, cdf\_target):

"""Create a mapping from source to target intensities."""

mapping = np.zeros(256, dtype=np.uint8)

for src\_intensity in range(256):

diff = np.abs(cdf\_source[src\_intensity] - cdf\_target)

mapping[src\_intensity] = np.argmin(diff)

return mapping

**What’s happening here?**

* The goal: Find a mapping that assigns each intensity value in the **source image** to a new intensity value so that the histogram of the transformed image matches the **target image**.
* For each intensity value r in the source (0 to 255):
  + Take its CDF value: cdf\_source[r]
  + Find the intensity z in the target such that cdf\_target[z] is closest to cdf\_source[r].
    - This is done by finding the minimum absolute difference.
  + Store this mapping: mapping[r] = z

**Why?**

* According to the theory of histogram matching:

 where

* T(r)T(r)T(r) is the CDF of the source image
* G−1(⋅)G^{-1}(\cdot)G−1(⋅) is the inverse of the target image’s CDF

 Since G−1G^{-1}G−1 might not be continuous or directly available, we approximate by finding the closest matching intensity based on the target CDF.

def histogram\_match(source\_img, target\_img):

"""Apply histogram matching from scratch."""

\_, cdf\_source = calculate\_pdf\_and\_cdf(source\_img)

\_, cdf\_target = calculate\_pdf\_and\_cdf(target\_img)

mapping = build\_mapping(cdf\_source, cdf\_target)

matched\_flat = mapping[source\_img.flatten()]

matched\_img = matched\_flat.reshape(source\_img.shape)

return matched\_img

**What’s happening here?**

* Compute the CDF of **source** and **target** images.
* Generate the intensity mapping based on CDFs.
* Apply the mapping to every pixel intensity in the source image.
  + We flatten the source image to a 1D array.
  + Replace each pixel value with its mapped value.
* Reshape the mapped array back to the original image shape.

def plot\_images(images, titles):

plt.figure(figsize=(15, 5))

for i in range(len(images)):

plt.subplot(1, len(images), i + 1)

plt.imshow(images[i], cmap='gray')

plt.title(titles[i])

plt.axis('off')

plt.tight\_layout()

plt.show()

**What’s happening here?**

* Visualizes images side-by-side using Matplotlib.
* This is useful to visually compare the source image, target image, and matched image.

**5. Generating Synthetic Images (Optional)**

If you don’t have source/target images handy, you can generate them.

**Generate a gradient source image:**

def generate\_gradient\_image(width=256, height=256):

gradient = np.tile(np.arange(width, dtype=np.uint8), (height, 1))

return gradient

* Creates a simple image where pixel intensity gradually increases horizontally from 0 to 255.
* Very useful to see how histogram matching affects the image.

**Generate a random target image:**

def generate\_random\_contrast\_image(width=256, height=256):

rng = np.random.default\_rng(seed=42)

img = rng.integers(low=0, high=256, size=(height, width), dtype=np.uint8)

return img

* Creates an image with random intensities, providing a drastically different histogram from the gradient.

**🧮 Summary of the Theory Behind This**

* The **CDF** is a monotonically increasing function from 0 to 1 that represents the cumulative pixel intensity distribution.
* Histogram matching aligns the CDF of the source image to that of the target image, by applying the mapping z=G−1(T(r))z = G^{-1}(T(r))z=G−1(T(r)).
* The process effectively changes the brightness and contrast distribution of the source image to resemble that of the target image.\

2.2

import cv2

import numpy as np

import matplotlib.pyplot as plt

from skimage import data

 **cv2 (OpenCV)**: For image processing tasks (though here we mainly use it for padding and color conversions).

 **numpy**: For numerical operations, arrays, and matrix calculations.

 **matplotlib.pyplot**: For displaying images side-by-side.

 **skimage.data**: Provides sample images (so you don't need to download any).

def add\_noise(image, noise\_type="salt\_pepper", amount=0.05):

noisy = image.copy()

if noise\_type == "salt\_pepper":

# Calculate the number of salt and pepper pixels based on the amount

num\_salt = np.ceil(amount \* image.size \* 0.5).astype(int)

num\_pepper = np.ceil(amount \* image.size \* 0.5).astype(int)

# Randomly pick coordinates for salt (white) noise

coords = [np.random.randint(0, i - 1, num\_salt) for i in image.shape]

noisy[coords[0], coords[1]] = 255

# Randomly pick coordinates for pepper (black) noise

coords = [np.random.randint(0, i - 1, num\_pepper) for i in image.shape]

noisy[coords[0], coords[1]] = 0

elif noise\_type == "gaussian":

# Generate Gaussian noise matrix with mean=0 and std=25

row, col = image.shape

mean = 0

sigma = 25

gauss = np.random.normal(mean, sigma, (row, col)).reshape(row, col)

noisy = image + gauss

noisy = np.clip(noisy, 0, 255).astype(np.uint8)

return noisy

 **Salt and Pepper Noise**: Randomly flips some pixels to black (0) or white (255) to simulate noise.

 **Gaussian Noise**: Adds normally-distributed noise with a given standard deviation (sigma) to pixel values.

def show\_images(images, titles, figsize=(15,10)):

plt.figure(figsize=figsize)

n = len(images)

for i in range(n):

plt.subplot(1, n, i+1)

if len(images[i].shape) == 2:

plt.imshow(images[i], cmap='gray')

else:

plt.imshow(cv2.cvtColor(images[i], cv2.COLOR\_BGR2RGB))

plt.title(titles[i])

plt.axis('off')

plt.show()

 Takes a list of images and their titles.

 Uses matplotlib to plot them in a row.

 Shows grayscale images properly using a colormap.

def gaussian\_kernel(kernel\_size=3, sigma=1):

ax = np.linspace(-(kernel\_size // 2), kernel\_size // 2, kernel\_size)

xx, yy = np.meshgrid(ax, ax)

kernel = np.exp(-(xx\*\*2 + yy\*\*2) / (2 \* sigma\*\*2))

kernel = kernel / np.sum(kernel)

return kernel

 Creates a 2D Gaussian kernel matrix of size kernel\_size x kernel\_size.

 sigma controls the spread (standard deviation).

 Normalizes kernel so the sum of all weights is 1 (to maintain brightness)

def apply\_gaussian\_filter(image, kernel\_size=3, sigma=1):

kernel = gaussian\_kernel(kernel\_size, sigma)

pad = kernel\_size // 2

padded\_img = np.pad(image, pad, mode='edge')

filtered\_img = np.zeros\_like(image)

for i in range(image.shape[0]):

for j in range(image.shape[1]):

roi = padded\_img[i:i+kernel\_size, j:j+kernel\_size]

filtered\_img[i,j] = np.sum(roi \* kernel)

return filtered\_img.astype(np.uint8)

* Pads the image to handle edges (using edge replication).
* For each pixel, extracts a region of interest (ROI) of the kernel size.
* Performs element-wise multiplication of ROI and kernel, sums it up, and sets the filtered pixel.
* This smooths the image by averaging neighboring pixels weighted by the Gaussian.

def apply\_median\_filter(image, kernel\_size=3):

pad = kernel\_size // 2

padded\_img = np.pad(image, pad, mode='edge')

filtered\_img = np.zeros\_like(image)

for i in range(image.shape[0]):

for j in range(image.shape[1]):

roi = padded\_img[i:i+kernel\_size, j:j+kernel\_size]

filtered\_img[i, j] = np.median(roi)

return filtered\_img.astype(np.uint8)

* Similar padding as Gaussian filter.
* For each pixel, replaces the pixel with the median value of the neighborhood.
* Effective for removing salt and pepper noise while preserving edges.

# Load sample grayscale image from skimage

image = data.camera()

image = (image).astype(np.uint8) # ensure uint8 format

# Add salt and pepper noise

noisy\_image = add\_noise(image, noise\_type='salt\_pepper', amount=0.05)

# Apply filters with different parameters

gaussian\_3\_1 = apply\_gaussian\_filter(noisy\_image, kernel\_size=3, sigma=1)

gaussian\_5\_1\_5 = apply\_gaussian\_filter(noisy\_image, kernel\_size=5, sigma=1.5)

median\_3 = apply\_median\_filter(noisy\_image, kernel\_size=3)

median\_5 = apply\_median\_filter(noisy\_image, kernel\_size=5)

# Show results

show\_images(

[image, noisy\_image, gaussian\_3\_1, gaussian\_5\_1\_5, median\_3, median\_5],

['Original', 'Noisy (Salt & Pepper)', 'Gaussian 3x3 σ=1', 'Gaussian 5x5 σ=1.5', 'Median 3x3', 'Median 5x5']

)

* Loads a standard test image (camera) from skimage.
* Adds salt & pepper noise.
* Applies Gaussian filters with different kernel sizes and sigma.
* Applies median filters with different kernel sizes.
* Shows all images for visual comparison.

**What You Learn from This Code**

* How to add noise to images artificially.
* How Gaussian and Median filters work and differ.
* How kernel size and sigma affect Gaussian filtering.
* Median filter’s edge-preserving property in removing salt & pepper noise.
* Visualization techniques to compare image results.

2.3