Final Project Code - Enhancing & Revealing Hidden Image Details

Code Layout:

- Libaries and Functions
- Noising Images
- Testing Identifiers of Noise Types
- Filter Testing
- Deconvolution Testing
- Bicubic Interpolation Testing
- Exposure Correction Testing
- Sharpening Testing

Libraries and Dependencies

In [1]:

```
import numpy as np
import cv2
import os
import random
import matplotlib.pyplot as plt
import scipy.io as sio
import skimage.io as io
from PIL import Image
from tqdm import tqdm
import pywt
import bm3d

from scipy.interpolate import griddata
from scipy.ndimage import gaussian_filter
from scipy.ndimage import median_filter
from scipy.signal import fftconvolve
from scipy.signal import convolve2d
import scipy.fftpack
import scipy.stats


from skimage.metrics import peak_signal_noise_ratio
from skimage.metrics import structural_similarity as ssim
from skimage.filters import gaussian
from skimage.restoration import denoise_tv_chambolle
from skimage.restoration import wiener
from skimage.restoration import richardson_lucy
from skimage import img_as_float
```

Functions

Denoising Evaluation Functions

In [2]:

```
# Peak Signal-to-Noise Ratio
def calc_psnr(img1, img2):

    # Ensure images are the same shape
    assert img1.shape == img2.shape, "Images must have the same dimensions"

    # Set scale to [0,1]
    if img1.max() > 1:
        img1_scaled = img1 / 255.0
    else:
        img1_scaled = img1
    if img2.max() > 1:
        img2_scaled = img2 / 255.0
    else:
        img2_scaled = img2

    # Calc PSNR
    psnr_value = peak_signal_noise_ratio(img2_scaled,
                                         img1_scaled, data_range=1.0)

    return psnr_value

# Structural Similarity Index Measure
def calc_ssim(clean_image, noisy_image):

    # Ensure images are the same shape
    assert clean_image.shape == noisy_image.shape, "Images must have the same dimensions"

    # Determine pixel range
    range_clean = clean_image.max()
    range_noisy = noisy_image.max()

    # Normalize to [0,1] if needed
    if range_clean > 1 or range_noisy > 1:
        clean_image = clean_image.astype(np.float32) / 255.0
        noisy_image = noisy_image.astype(np.float32) / 255.0

    # Compute SSIM for each channel
    ssim_per_channel = [
        ssim(clean_image[:, :, i], noisy_image[:, :, i], data_range=1.0)
        for i in range(3)  # Loop through RGB channels
    ]

    return np.mean(ssim_per_channel)  # Average SSIM across channels
```

Noise Functions

In [3]:

```
# Read Noise - Gaussian
def add_gaussian_noise(image, sigma=0.1):

    # Determine working range
    min_img = 0
    if np.max(image) <= 1:
        max_img = 1
    else:
        max_img = 255

    # Develop noise for each pixel
    noise = np.random.normal(0, sigma, image.shape).astype(np.float32)

    # Add noise into image and clip to size
    noisy_image = np.clip(image + noise*max_img, min_img, max_img)

    return noisy_image

# Low Resolution - Downsampling
def downsample_image(image, scale):

    # Determine working range
    min_img = 0
    if np.max(image) <= 1:
        max_img = 1
    else:
        max_img = 255

    # Current size
    h, w = image.shape[:2]

    # Apply Gaussian anti-aliasing (lowpass) filter
    sigma = scale / 2
    blurred = cv2.GaussianBlur(image, (0, 0), sigmaX=sigma, sigmaY=sigma)

    # Downsample the image
    small = cv2.resize(blurred, (w // scale, h // scale), interpolation=cv2.INTER_AREA)

    # Upsample back to original size for PSNR and SSIM calc
    upsampled = cv2.resize(small, (w, h), interpolation=cv2.INTER_LINEAR)

    return np.clip(upsampled, min_img, max_img)

# Motion Blur - Affine Warp
def apply_motion_blur(image, kernel_size=15):

    # Determine working range
    min_img = 0
    if np.max(image) <= 1:
        max_img = 1
    else:
        max_img = 255

    # Create kernel
    kernel = np.zeros((kernel_size, kernel_size))
    kernel[(kernel_size - 1) // 2, :] = np.ones(kernel_size)

    # Kernel across single direction
    kernel = cv2.warpAffine(kernel,
                            cv2.getRotationMatrix2D((kernel_size / 2, kernel_size / 2), 0, 1),
                            (kernel_size, kernel_size))
    kernel /= kernel_size

    # Apply blur to each channel separately and clip to size
    blurred = np.zeros_like(image)
    for i in range(3):  # Loop through RGB channels
        blurred[:, :, i] = cv2.filter2D(image[:, :, i], -1, kernel)


    return np.clip(blurred, min_img, max_img)

# Contrast Reduction - Underexposure / Overexposure
def add_shadows(image, factor=0.5):

    # Determine working range
    min_img = 0
    if np.max(image) <= 1:
        max_img = 1
    else:
        max_img = 255

    # Affect and clip
    return np.clip(image * factor, min_img, max_img)

def add_overexposure(image, factor=1.5):

    # Determine working range
    min_img = 0
    if np.max(image) <= 1:
        max_img = 1
    else:
        max_img = 255

    # Affect and clip
    return np.clip(image * factor, min_img, max_img)
```

Denoise Filter Functions

In [4]:

```
# Gaussian Spatial Kernel - HW2
def fspecial_gaussian_2d(size, sigma):
    kernel = np.zeros(tuple(size))
    kernel[size[0]//2, size[1]//2] = 1
    kernel = gaussian(kernel, sigma)
    return kernel/np.sum(kernel)
```

In [5]:

```
# Bilateral Filter - HW2 (adjusted to work for color image inputs)
def bilateral2d(img, radius, sigma, sigmaIntensity):
    pad = radius
    # Initialize filtered image to 0
    out = np.zeros_like(img)
    for channel in [0, 1, 2]:

        # Pad image to reduce boundary artifacts
        imgPad = np.pad(img[:,:,channel], pad)

        # Smoothing kernel, gaussian with standard deviation sigma
        # and size (2*radius+1, 2*radius+1)
        filtSize = (2*radius + 1, 2*radius + 1)
        spatialKernel = fspecial_gaussian_2d(filtSize, sigma)
        for y in range(img.shape[0]):
            for x in range(img.shape[1]):
                patchVals = imgPad[y+pad-radius:y+pad+radius+1, x+pad-radius:x+pad+radius+1]
                centerVal = imgPad[y+pad, x+pad]
                intensityWeights = np.exp(-((patchVals - centerVal) ** 2) / (2*sigmaIntensity**2))
                weights = spatialKernel * intensityWeights / np.sum(spatialKernel * intensityWeights)
                out[y, x, channel] = np.sum(patchVals * weights)

    return out
```

In [6]:

```
# Non-Local Means Filter - HW2 (adjusted to work for color image inputs)
def inbounds(img, y, x):
    return 0 <= y and y < img.shape[0] and \
           0 <= x and x < img.shape[1]

def comparePatches(patch1, patch2, kernel, sigma):
    return np.exp(-np.sum(kernel*(patch1 - patch2) ** 2)/(2*sigma**2))

def nonlocalmeans(img, searchWindowRadius, averageFilterRadius, sigma, nlmSigma):
    # Initialize output to 0
    out = np.zeros_like(img)
    # Pad image to reduce boundary artifacts
    pad = max(averageFilterRadius, searchWindowRadius)
    for channel in [0, 1, 2]:
        imgPad = np.pad(img[...,channel:channel+1], pad)
        imgPad = imgPad[..., pad:-pad] # Don't pad color channel

        # Smoothing kernel
        filtSize = (2*averageFilterRadius + 1, 2*averageFilterRadius + 1)
        kernel = fspecial_gaussian_2d(filtSize, sigma)
        # Add third axis for broadcasting
        kernel = kernel[:, :, np.newaxis]
        for y in range(img.shape[0]):
            for x in range(img.shape[1]):
                centerPatch = imgPad[y+pad-averageFilterRadius:y+pad+averageFilterRadius+1,
                                     x+pad-averageFilterRadius:x+pad+averageFilterRadius+1,
                                     :]
                window = imgPad[y+pad-searchWindowRadius:y+pad+searchWindowRadius+1,
                                x+pad-searchWindowRadius:x+pad+searchWindowRadius+1,
                                :]
                # Go over a window around the current pixel, compute weights
                # based on difference of neighborhoods.
                weights = np.zeros((2*searchWindowRadius+1, 2*searchWindowRadius+1, 1))
                max_weight = 0
                for i, dy in enumerate(range(-searchWindowRadius, searchWindowRadius+1)):
                    for j, dx in enumerate(range(-searchWindowRadius, searchWindowRadius+1)):
                        if dy == 0 and dx == 0:
                            continue # Ignore center pixel
                        elif not inbounds(img, y+dy, x+dx):
                            continue # Ignore out-of-bounds pixels
                        otherPatch = imgPad[y+dy+pad-averageFilterRadius:y+dy+pad+averageFilterRadius+1,
                                            x+dx+pad-averageFilterRadius:x+dx+pad+averageFilterRadius+1,
                                            :]
                        weight = comparePatches(centerPatch, otherPatch, kernel, nlmSigma)
                        weights[i, j, 0] = weight
                        # Trick: Use center of window to store maximum weight
                        if weight > max_weight:
                            weights[searchWindowRadius, searchWindowRadius] = weight
                            max_weight = weight

                weights /= np.sum(weights)
                out[y, x, channel] = np.sum(weights * window, axis=(0,1))

    return out
```

In [7]:

```
# Anisotropic Filter
def anisotropic(image, num_iterations=20, K=20, lambda_param=0.25):

    if np.max(image) <= 1:
        img = np.copy(image)
    else:
        img = np.copy(image.astype(np.float32) / 255)


    for _ in range(num_iterations):
        for c in range(3):  # Apply diffusion on each color channel separately
            grad_north = np.roll(img[:, :, c], -1, axis=0) - img[:, :, c]
            grad_south = np.roll(img[:, :, c], 1, axis=0) - img[:, :, c]
            grad_east = np.roll(img[:, :, c], -1, axis=1) - img[:, :, c]
            grad_west = np.roll(img[:, :, c], 1, axis=1) - img[:, :, c]

            c_north = np.exp(-(grad_north / K) ** 2)
            c_south = np.exp(-(grad_south / K) ** 2)
            c_east = np.exp(-(grad_east / K) ** 2)
            c_west = np.exp(-(grad_west / K) ** 2)

            img[:, :, c] += lambda_param * (
                c_north * grad_north + c_south * grad_south +
                c_east * grad_east + c_west * grad_west
            )

    return np.clip(img, 0, 1)
```

In [8]:

```
# Total Variation Filter
def total_variation_denoising(image, weight=0.1):

    if np.max(image) <= 1:
        img = image
    else:
        img = image.astype(np.float32) / 255

    denoised_img = denoise_tv_chambolle(img, weight=weight, multichannel=True)

    return np.clip(denoised_img, 0, 1)
```

In [40]:

```
#BM3D
def bm3d_denoise(image, sigma=0.3):
    """
    Applies BM3D denoising to a color image by processing each channel separately.

    Parameters:
        image (ndarray): Input color image (H, W, 3), expected in uint8 format.
        sigma (float): Noise standard deviation (0-1 range).

    Returns:
        ndarray: Denoised color image.
    """

    if np.max(image) <= 1:
        img = image
    else:
        img = image.astype(np.float32) / 255

    # Apply BM3D to each channel separately
    denoised_channels = [bm3d.bm3d(img[:, :, c], sigma_psd=sigma) for c in range(3)]

    # Stack channels back together and convert to uint8
    denoised_img = np.stack(denoised_channels, axis=-1)
    denoised_img = np.clip(denoised_img, 0, 1)

    return denoised_img
```

Deconvolution and Resolution Functions

In [9]:

```
# Weiner Deconvolution
def wiener_deconvolution(image, psf_size=15, balance=0.1):

    if np.max(image) > 1:
        img = image.astype(np.float32) / 255.0  # Normalize if needed
    else:
        img = image.astype(np.float32)

    # Define the normalized PSF
    psf = np.ones((psf_size, psf_size)) / (psf_size ** 2)

    # Convert PSF to the frequency domain
    psf_fft = scipy.fftpack.fft2(psf, shape=img.shape[:2])

    # Apply Wiener deconvolution to each color channel
    deblurred_channels = []
    for c in range(3):
        img_fft = scipy.fftpack.fft2(img[:, :, c])  # Image to frequency domain
        wiener_filter = np.conj(psf_fft) / (np.abs(psf_fft) ** 2 + balance)  # Wiener filter
        deblurred_fft = img_fft * wiener_filter
        deblurred = np.real(scipy.fftpack.ifft2(deblurred_fft))  # Inverse FFT
        deblurred_channels.append(deblurred)

    # Stack back into a color image
    deblurred_image = np.stack(deblurred_channels, axis=-1)

    return np.clip(deblurred_image, 0, 1)  # Ensure values are in [0,1]
```

In [51]:

```
# Richardson-Lucy Deconvolution
def motion_psf(psf_size, angle):

    psf = np.zeros((psf_size, psf_size))
    psf[psf_size // 2, :] = 1  # Create a horizontal line

    # Rotate to desired angle
    psf = cv2.warpAffine(psf, cv2.getRotationMatrix2D((psf_size // 2, psf_size // 2), angle, 1), (psf_size, psf_size))

    psf /= psf.sum()  # Normalize
    return psf

def rl_deconvolution(image, psf_size=15, num_iterations=30, angle=0):

    if np.max(image) > 1:
        img = image.astype(np.float32) / 255.0  # Normalize if needed
    else:
        img = image.astype(np.float32)

    # Generate motion blur PSF
    psf = motion_psf(psf_size, angle)

    # Apply RL deconvolution to each channel separately
    deblurred_channels = [richardson_lucy(img[:, :, c] + 1e-8, psf, iterations=num_iterations) for c in range(3)]

    # Stack the channels back into a color image
    deblurred_image = np.stack(deblurred_channels, axis=-1)

    return np.clip(deblurred_image, 0, 1)  # Ensure values are in [0,1]
```

In [54]:

```
def gaussian_motion_psf(psf_size, angle):

    # Ensure PSF size is always odd (needed for GaussianBlur)
    if psf_size % 2 == 0:
        psf_size += 1

    psf = np.zeros((psf_size, psf_size))
    psf[psf_size // 2, :] = 1  # Create a horizontal line

    # Rotate to desired angle
    psf = cv2.warpAffine(psf, cv2.getRotationMatrix2D((psf_size // 2, psf_size // 2), angle, 1), (psf_size, psf_size))

    psf = cv2.GaussianBlur(psf, (psf_size, psf_size), psf_size / 3)  # Apply Gaussian blur
    psf /= psf.sum()  # Normalize
    return psf

def rl_deconvolution(image, psf_size=15, num_iterations=30, angle=0):

    if np.max(image) > 1:
        img = image.astype(np.float32) / 255.0  # Normalize if needed
    else:
        img = image.astype(np.float32)

    # Generate motion blur PSF
    psf = gaussian_motion_psf(psf_size, angle)

    # Apply RL deconvolution to each channel separately
    deblurred_channels = [richardson_lucy(img[:, :, c] + 1e-8, psf, iterations=num_iterations) for c in range(3)]

    # Stack the channels back into a color image
    deblurred_image = np.stack(deblurred_channels, axis=-1)

    return np.clip(deblurred_image, 0, 1)  # Ensure values are in [0,1]
```

Tonemapping & Exposure Correcting Functions

In [11]:

```
# Adaptive Histogram Equalization
def correct_exposure(image, size=8):

    if np.max(image) > 1:
        img = image.astype(np.float32) / 255.0  # Normalize if needed
    else:
        img = image.astype(np.float32)

    # Convert from RGB to LAB (cv2 expects uint8 input)
    lab = cv2.cvtColor((img * 255).astype(np.uint8), cv2.COLOR_RGB2LAB)

    # Split LAB channels
    l, a, b = cv2.split(lab)

    # Apply Adaptive Histogram Equalization (CLAHE) on L channel
    clahe = cv2.createCLAHE(clipLimit=3.0, tileGridSize=(size,size))
    l = clahe.apply(l)

    # Merge LAB channels
    lab = cv2.merge((l, a, b))

    # Convert back to RGB and normalize to 0-1 range
    corrected_image = cv2.cvtColor(lab, cv2.COLOR_LAB2RGB).astype(np.float32) / 255.0

    return np.clip(corrected_image, 0, 1)  # Ensure valid range
```

In [12]:

```
# Laplacian Tonemapping
def local_laplacian_tone_mapping(image, alpha=1.5, beta=0.5):

    if np.max(image) > 1:
        img = image.astype(np.float32) / 255.0  # Normalize if needed
    else:
        img = image.astype(np.float32)

    # Convert from RGB to LAB (cv2 expects uint8 input)
    img_lab = cv2.cvtColor((img * 255).astype(np.uint8), cv2.COLOR_RGB2LAB)

    # Split LAB channels
    l_channel, a, b = cv2.split(img_lab)

    # Apply CLAHE (Contrast Limited Adaptive Histogram Equalization) on L channel
    clahe = cv2.createCLAHE(clipLimit=alpha, tileGridSize=(8, 8))
    l_enhanced = clahe.apply(l_channel)

    # Merge enhanced L channel with original A, B channels
    img_lab = cv2.merge((l_enhanced, a, b))

    # Convert back to RGB and normalize to 0-1 range
    img_tone_mapped = cv2.cvtColor(img_lab, cv2.COLOR_LAB2RGB).astype(np.float32) / 255.0

    return np.clip(img_tone_mapped, 0, 1)  # Ensure valid range
```

In [13]:

```
# Bilateral Tonemapping
def bilateral_tone_mapping(image, sigma_spatial=15, sigma_range=0.5):

    if np.max(image) > 1:
        img = image.astype(np.float32) / 255.0  # Normalize if needed
    else:
        img = image.astype(np.float32)

    # Convert RGB to LAB (cv2 expects uint8 input)
    img_lab = cv2.cvtColor((img * 255).astype(np.uint8), cv2.COLOR_RGB2LAB)

    # Split LAB channels
    l_channel, a, b = cv2.split(img_lab)

    # Apply bilateral filter to L channel (convert to uint8 for filtering)
    l_base = cv2.bilateralFilter(l_channel.astype(np.uint8), d=0, sigmaColor=sigma_range * 255, sigmaSpace=sigma_spatial)

    # Extract details (convert L channel back to float before subtraction)
    l_detail = l_channel.astype(np.float32) / 255.0 - l_base.astype(np.float32) / 255.0

    # Compress base layer to prevent overexposure
    l_compressed = np.log1p(l_base.astype(np.float32)) / np.log1p(255)  # Log-based compression

    # Reconstruct the L channel
    l_final = l_compressed + l_detail  # Add details back

    # Normalize L channel to [0, 255]
    l_final = cv2.normalize(l_final, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

    # Merge channels and convert back to RGB
    img_lab = cv2.merge((l_final, a, b))
    img_tone_mapped = cv2.cvtColor(img_lab, cv2.COLOR_LAB2RGB).astype(np.float32) / 255.0  # Normalize output

    return np.clip(img_tone_mapped, 0, 1)  # Ensure valid range
```

Sharpening Functions

In [14]:

```
# Bucubic Interpolation
def upscale_image(image, scale=2):

    # Ensure the image is in float format (0-1)
    if np.max(image) > 1:
        img = image.astype(np.float32) / 255.0  # Normalize if needed
    else:
        img = image.astype(np.float32)

    h, w = img.shape[:2]

    # Upscale using bicubic interpolation
    upscaled = cv2.resize(img, (w * scale, h * scale), interpolation=cv2.INTER_CUBIC)

    # Resize back to original size to ensure valid PSNR/SSIM comparison
    final_image = cv2.resize(upscaled, (w, h), interpolation=cv2.INTER_CUBIC)

    return np.clip(final_image, 0, 1)  # Ensure values stay in [0,1]
```

In [15]:

```
# Unsharp Masking
def sharpen_image(image, sigma=1.5, strength=1.5):

    # Ensure image is in float format (0-1)
    if np.max(image) > 1:
        img = image.astype(np.float32) / 255.0  # Normalize if needed
    else:
        img = image.astype(np.float32)

    # Apply Gaussian blur
    blurred = cv2.GaussianBlur(img, (5, 5), sigmaX=sigma, sigmaY=sigma)

    # Apply Unsharp Masking
    sharpened = cv2.addWeighted(img, 1 + strength, blurred, -strength, 0)

    return np.clip(sharpened, 0, 1)  # Ensure values are in [0,1]
```

All Noise Types Presented

Image Selection

In [16]:

```
image_dir = "BSDS300/images/train/"
num_images = 50
random.seed(42)
selected_images = random.sample(os.listdir(image_dir), num_images)
```

Noise Types & Levels

For each noise type, we choose 3 levels to test our denoising strategies

In [17]:

```
noise_types = {
    "Gaussian": ([0.1, 0.3, 0.6], add_gaussian_noise),
    "Downsampling": ([2, 4, 8], downsample_image),
    "Motion Blur": ([5, 10, 20], apply_motion_blur),
    "Shadowed": ([0.7, 0.5, 0.3], add_shadows),
    "Overexposure": ([1.3, 1.6, 2.2], add_overexposure)
}
```

Noise Examples

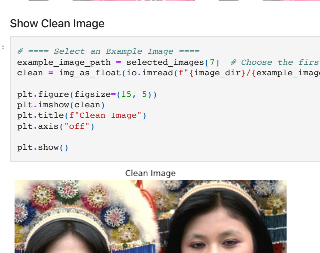
In [18]:

```
# ==== Select an Example Image ====
example_image_path = selected_images[7]  # Choose the first image from dataset
clean = img_as_float(io.imread(f"{image_dir}/{example_image_path}"))

# ==== Settings for Each Noise Type ====
gaussian_levels = [0.1, 0.3, 0.6]
downsample_levels = [2, 4, 8]
motion_blur_levels = [5, 10, 20]
shadow_levels = [0.7, 0.5, 0.3]  # Darker as factor decreases
overexposure_levels = [1.3, 1.6, 2.2]  # Brighter as factor increases

# ==== Generate Subplots ====
fig, axes = plt.subplots(5, 3, figsize=(12, 15))
fig.suptitle("Image Corruption Methods", fontsize=16)

# ==== Apply and Display Each Type of Noise ====
for i, sigma in enumerate(gaussian_levels):
    noisy = add_gaussian_noise(clean, sigma)
    psnr_value = calc_psnr(clean, noisy)
    ssim_value = calc_ssim(clean, noisy)
    axes[0, i].imshow(noisy)
    axes[0, i].set_title(f"σ={sigma}\nPSNR={psnr_value:.2f} | SSIM={ssim_value:.2f}")

for i, scale in enumerate(downsample_levels):
    noisy = downsample_image(clean, scale)
    psnr_value = calc_psnr(clean, noisy)
    ssim_value = calc_ssim(clean, noisy)
    axes[1, i].imshow(noisy)
    axes[1, i].set_title(f"Scale={scale}\nPSNR={psnr_value:.2f} | SSIM={ssim_value:.2f}")

for i, kernel_size in enumerate(motion_blur_levels):
    noisy = apply_motion_blur(clean, kernel_size)
    psnr_value = calc_psnr(clean, noisy)
    ssim_value = calc_ssim(clean, noisy)
    axes[2, i].imshow(noisy)
    axes[2, i].set_title(f"Kernel={kernel_size}\nPSNR={psnr_value:.2f} | SSIM={ssim_value:.2f}")

for i, factor in enumerate(shadow_levels):
    noisy = add_shadows(clean, factor)
    psnr_value = calc_psnr(clean, noisy)
    ssim_value = calc_ssim(clean, noisy)
    axes[3, i].imshow(noisy)
    axes[3, i].set_title(f"Shadow={factor}\nPSNR={psnr_value:.2f} | SSIM={ssim_value:.2f}")

for i, factor in enumerate(overexposure_levels):
    noisy = add_overexposure(clean, factor)
    psnr_value = calc_psnr(clean, noisy)
    ssim_value = calc_ssim(clean, noisy)
    axes[4, i].imshow(noisy)
    axes[4, i].set_title(f"Overexp={factor}\nPSNR={psnr_value:.2f} | SSIM={ssim_value:.2f}")

# ==== Hide Axes ====
for ax in axes.ravel():
    ax.axis("off")

# ==== Row Labels ====
row_labels = ["Gaussian", "Downsampling", "Motion Blur", "Shadowed", "Overexposed"]

# Place text labels on the left side of each row
for i, label in enumerate(row_labels):
    fig.text(0.01, 0.82 - i * 0.185, label, va='center', ha='center',
             fontsize=14, fontweight='bold', rotation=90)

plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```

Show Clean Image

In [20]:

```
# ==== Select an Example Image ====
example_image_path = selected_images[7]  # Choose the first image from dataset
clean = img_as_float(io.imread(f"{image_dir}/{example_image_path}"))

plt.figure(figsize=(15, 5))
plt.imshow(clean)
plt.title(f"Clean Image")
plt.axis("off")

plt.show()
```



Identify Noise Sources

Can we determine what types of noise are in an image?

In [70]:

```
def compute_exposure_metrics(image):

    # Convert to grayscale
    image_uint8 = (image * 255).clip(0, 255).astype(np.uint8)
    gray = cv2.cvtColor(image_uint8, cv2.COLOR_RGB2GRAY)

    # Compute histogram
    hist = cv2.calcHist([gray], [0], None, [256], [0, 256])

    # Compute average intensity
    average_intensity = int(np.mean(gray))

    # Compute peak intensity
    peak_intensity = int(np.argmax(hist))

    # Compute standard deviation (spread of intensities)
    std_dev = np.std(gray)

    # Compute skewness (left-skew = shadow, right-skew = overexposure)
    skewness = scipy.stats.skew(gray.flatten())

    # Compute contrast (difference between 5th and 95th percentile)
    p5, p95 = np.percentile(gray, [5, 95])
    contrast = p95 - p5

    return {
        "Average Intensity": average_intensity,
        "Peak Intensity": peak_intensity,
        "Std Dev": round(std_dev, 3),
        "Skewness": round(skewness, 3),
        "Contrast": round(contrast, 3),
    }
```

In [74]:

```
# Select an example image
example_image_path = selected_images[4]
clean_image = img_as_float(io.imread(f"{image_dir}/{example_image_path}"))

print("\n=== Exposure Analysis for Example Image ===")
print(f"Image: {example_image_path}")
print("-" * 50)
print(f"{'Noise Type':<15} {'Noise Level':<15} {'Mean':<10} {'Peak':<10} {'Std Dev':<10} {'Skewness':<10} {'Contrast':<10} ")
print("-" * 95)

for noise_type, (noise_levels, noise_fn) in noise_types.items():
    for noise_level in noise_levels:
        noisy_image = noise_fn(clean_image, noise_level)
        metrics = compute_exposure_metrics(noisy_image)

        # Print results
        print(f"{noise_type:<15} {noise_level:<15} {metrics['Average Intensity']:<10} {metrics['Peak Intensity']:<10} {metrics['Std Dev']:<10} {metrics['Skewness']:<10} {metrics['Contrast']:<10} ")
```

```
=== Exposure Analysis for Example Image ===
Image: 227040.jpg
--------------------------------------------------
Noise Type      Noise Level     Mean       Peak       Std Dev    Skewness   Contrast
-----------------------------------------------------------------------------------------------
Gaussian        0.1             122        131        27.832     -0.682     93.0
Gaussian        0.3             123        124        50.118     -0.017     166.0
Gaussian        0.6             124        76         66.158     0.024      213.0
Downsampling    2               122        135        19.352     -1.551     65.0
Downsampling    4               122        135        17.87      -1.513     57.0
Downsampling    8               122        135        15.917     -1.343     47.0
Motion Blur     5               123        135        21.013     -1.409     71.0
Motion Blur     10              122        135        20.183     -1.404     68.0
Motion Blur     20              122        135        19.014     -1.352     63.0
Shadowed        0.7             85         94         15.483     -1.391     53.0
Shadowed        0.5             61         68         11.052     -1.387     38.0
Shadowed        0.3             36         40         6.64       -1.396     23.0
Overexposure    1.3             159        176        28.749     -1.392     99.0
Overexposure    1.6             196        217        35.077     -1.463     121.0
Overexposure    2.2             240        255        31.426     -2.58      97.0
```

In [75]:

```
print("\n=== Exposure Analysis Averaged Across 50 Images ===")
print("-" * 95)
print(f"{'Noise Type':<15} {'Noise Level':<15} {'Avg Mean':<10} {'Avg Peak':<10} {'Avg Std Dev':<10} {'Avg Skewness':<10} {'Avg Contrast':<10} ")
print("-" * 95)

for noise_type, (noise_levels, noise_fn) in noise_types.items():
    for noise_level in noise_levels:
        mean_values, peak_values, std_values, skew_values, contrast_values = [], [], [], [], []
        classifications = []

        # Loop over 50 images
        for img_name in selected_images:
            img_path = f"{image_dir}/{img_name}"
            clean_image = img_as_float(io.imread(img_path))

            # Apply noise
            noisy_image = noise_fn(clean_image, noise_level)

            # Compute metrics and classification
            metrics = compute_exposure_metrics(noisy_image)


            # Collect results
            mean_values.append(metrics["Average Intensity"])
```

```
    gaussian_noise_adjusted_mean_time_lst.append(t)
    gaussian_noise_adjusted_psnr_lst.append(psnr)
    print("Gaussian Noise Adjusted PSNR: ", psnr)

    # Append to lists for plotting
    s_values.append(s)
    s_values_lst.append(s_values)
    gaussian_psnr_lst.append(gaussian_psnr)
    salt_pepper_psnr_lst.append(salt_pepper_psnr)
    poisson_psnr_lst.append(poisson_psnr)
    gaussian_noise_psnr_lst.append(gaussian_noise_psnr)
    gaussian_noise_adjusted_psnr_lst.append(gaussian_noise_adjusted_psnr)

    print("Processed Gaussian Denoised Noise for image {s}")

Noise Type:     Noise Level:    Min Time:       Max Time:       Mean Time:      Std Dev Time:   Min PSNR:       Max PSNR:       Mean PSNR:      Std Dev PSNR:
Gaussian        0.1             0.000           0.000           0.000           0.000           13.04           13.04           13.04           0.00
Gaussian        0.2             0.000           0.000           0.000           0.000           10.02           10.02           10.02           0.00
Gaussian        0.3             0.000           0.000           0.000           0.000           8.46            8.46            8.46            0.00
Gaussian        0.4             0.000           0.000           0.000           0.000           7.44            7.44            7.44            0.00
Gaussian        0.5             0.000           0.000           0.000           0.000           6.69            6.69            6.69            0.00
Gaussian        0.6             0.000           0.000           0.000           0.000           6.09            6.09            6.09            0.00
Gaussian        0.7             0.000           0.000           0.000           0.000           5.59            5.59            5.59            0.00
Gaussian        0.8             0.000           0.000           0.000           0.000           5.17            5.17            5.17            0.00
Gaussian        0.9             0.000           0.000           0.000           0.000           4.81            4.81            4.81            0.00
Gaussian        1.0             0.000           0.000           0.000           0.000           4.49            4.49            4.49            0.00
Gaussian        1.1             0.000           0.000           0.000           0.000           4.21            4.21            4.21            0.00
Gaussian        1.2             0.000           0.000           0.000           0.000           3.96            3.96            3.96            0.00
Gaussian        1.3             0.000           0.000           0.000           0.000           3.74            3.74            3.74            0.00
Gaussian        1.4             0.000           0.000           0.000           0.000           3.54            3.54            3.54            0.00
Gaussian        1.5             0.000           0.000           0.000           0.000           3.36            3.36            3.36            0.00
```

```
def compute_psnr(image):
    '''
    Function to compute PSNR of an image.
    Parameters: image (numpy.ndarray): The input image.
    Returns: PSNR (float): The PSNR value.
    '''
    return cv.PSNR(image, image)

# Initialize variables to store results
noise_types = ['Gaussian', 'Salt', 'Pepper', 'Poisson']
noise_levels = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5]
results = {}

# Loop through each noise type and noise level
for noise_type in noise_types:
    for noise_level in noise_levels:
        key = (noise_type, noise_level)
        results[key] = {'min': float('inf'), 'max': 0, 'sum': 0, 'count': 0}

# Loop through each image in the noisy_images_lst
for image in noisy_images_lst:
    psnr = compute_psnr(image)
    for noise_type in noise_types:
        for noise_level in noise_levels:
            if noise_type in image:
                key = (noise_type, noise_level)
                results[key]['min'] = min(results[key]['min'], psnr)
                results[key]['max'] = max(results[key]['max'], psnr)
                results[key]['sum'] += psnr
                results[key]['count'] += 1

# Print the results in a formatted table
print("Noise Type:\tNoise Level:\tMin PSNR:\tMax PSNR:\tMean PSNR:")
for key, value in results.items:
    noise_type, noise_level = key
    min_psnr = value['min']
    max_psnr = value['max']
    mean_psnr = value['sum'] / value['count']
    print(f"{noise_type}\t{noise_level}\t{min_psnr:.2f}\t{max_psnr:.2f}\t{mean_psnr:.2f}")

Noise Type:     Noise Level:    Min PSNR:       Max PSNR:       Mean PSNR:
Gaussian        0.1             13.04           13.04           13.04
Gaussian        0.2             10.02           10.02           10.02
Gaussian        0.3             8.46            8.46            8.46
Gaussian        0.4             7.44            7.44            7.44
Gaussian        0.5             6.69            6.69            6.69
Gaussian        0.6             6.09            6.09            6.09
Gaussian        0.7             5.59            5.59            5.59
Gaussian        0.8             5.17            5.17            5.17
Gaussian        0.9             4.81            4.81            4.81
Gaussian        1.0             4.49            4.49            4.49
Gaussian        1.1             4.21            4.21            4.21
Gaussian        1.2             3.96            3.96            3.96
Gaussian        1.3             3.74            3.74            3.74
Gaussian        1.4             3.54            3.54            3.54
Gaussian        1.5             3.36            3.36            3.36
```

```
def compute_ssim(image):
    '''
    Function to compute SSIM of an image.
    Parameters: image (numpy.ndarray): The input image.
    Returns: SSIM (float): The SSIM value.
    '''
    return ssim(image, image)

# Initialize variables to store results
noise_types = ['Gaussian', 'Salt', 'Pepper', 'Poisson']
noise_levels = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5]
results = {}

# Loop through each noise type and noise level
for noise_type in noise_types:
    for noise_level in noise_levels:
        key = (noise_type, noise_level)
        results[key] = {'min': float('inf'), 'max': 0, 'sum': 0, 'count': 0}

# Loop through each image in the noisy_images_lst
for image in noisy_images_lst:
    ssim = compute_ssim(image)
    for noise_type in noise_types:
        for noise_level in noise_levels:
            if noise_type in image:
                key = (noise_type, noise_level)
                results[key]['min'] = min(results[key]['min'], ssim)
                results[key]['max'] = max(results[key]['max'], ssim)
                results[key]['sum'] += ssim
                results[key]['count'] += 1

# Print the results in a formatted table
print("Noise Type:\tNoise Level:\tMin SSIM:\tMax SSIM:\tMean SSIM:")
for key, value in results.items:
    noise_type, noise_level = key
    min_ssim = value['min']
    max_ssim = value['max']
    mean_ssim = value['sum'] / value['count']
    print(f"{noise_type}\t{noise_level}\t{min_ssim:.2f}\t{max_ssim:.2f}\t{mean_ssim:.2f}")

Noise Type:     Noise Level:    Min SSIM:       Max SSIM:       Mean SSIM:
Gaussian        0.1             0.99            0.99            0.99
Gaussian        0.2             0.99            0.99            0.99
Gaussian        0.3             0.99            0.99            0.99
Gaussian        0.4             0.99            0.99            0.99
Gaussian        0.5             0.99            0.99            0.99
Gaussian        0.6             0.99            0.99            0.99
Gaussian        0.7             0.99            0.99            0.99
Gaussian        0.8             0.99            0.99            0.99
Gaussian        0.9             0.99            0.99            0.99
Gaussian        1.0             0.99            0.99            0.99
Gaussian        1.1             0.99            0.99            0.99
Gaussian        1.2             0.99            0.99            0.99
Gaussian        1.3             0.99            0.99            0.99
Gaussian        1.4             0.99            0.99            0.99
Gaussian        1.5             0.99            0.99            0.99
```

```
# Initialize variables to store results
noise_types = ['Gaussian', 'Salt', 'Pepper', 'Poisson']
noise_levels = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5]
results = {}

# Loop through each noise type and noise level
for noise_type in noise_types:
    for noise_level in noise_levels:
        key = (noise_type, noise_level)
        results[key] = {'min': float('inf'), 'max': 0, 'sum': 0, 'count': 0}

# Loop through each image in the noisy_images_lst
for image in noisy_images_lst:
    psnr = compute_psnr(image)
    for noise_type in noise_types:
        for noise_level in noise_levels:
            if noise_type in image:
                key = (noise_type, noise_level)
                results[key]['min'] = min(results[key]['min'], psnr)
                results[key]['max'] = max(results[key]['max'], psnr)
                results[key]['sum'] += psnr
                results[key]['count'] += 1

# Print the results in a formatted table
print("Noise Type:\tNoise Level:\tMin PSNR:\tMax PSNR:\tMean PSNR:")
for key, value in results.items:
    noise_type, noise_level = key
    min_psnr = value['min']
    max_psnr = value['max']
    mean_psnr = value['sum'] / value['count']
    print(f"{noise_type}\t{noise_level}\t{min_psnr:.2f}\t{max_psnr:.2f}\t{mean_psnr:.2f}")

Noise Type:     Noise Level:    Min PSNR:       Max PSNR:       Mean PSNR:
Salt            0.1             13.04           13.04           13.04
Salt            0.2             10.02           10.02           10.02
Salt            0.3             8.46            8.46            8.46
Salt            0.4             7.44            7.44            7.44
Salt            0.5             6.69            6.69            6.69
Salt            0.6             6.09            6.09            6.09
Salt            0.7             5.59            5.59            5.59
Salt            0.8             5.17            5.17            5.17
Salt            0.9             4.81            4.81            4.81
Salt            1.0             4.49            4.49            4.49
Salt            1.1             4.21            4.21            4.21
Salt            1.2             3.96            3.96            3.96
Salt            1.3             3.74            3.74            3.74
Salt            1.4             3.54            3.54            3.54
Salt            1.5             3.36            3.36            3.36
```

```
def compute_ssim(image):
    '''
    Function to compute SSIM of an image.
    Parameters: image (numpy.ndarray): The input image.
    Returns: SSIM (float): The SSIM value.
    '''
    return ssim(image, image)

# Initialize variables to store results
noise_types = ['Gaussian', 'Salt', 'Pepper', 'Poisson']
noise_levels = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5]
results = {}

# Loop through each noise type and noise level
for noise_type in noise_types:
    for noise_level in noise_levels:
        key = (noise_type, noise_level)
        results[key] = {'min': float('inf'), 'max': 0, 'sum': 0, 'count': 0}

# Loop through each image in the noisy_images_lst
for image in noisy_images_lst:
    ssim = compute_ssim(image)
    for noise_type in noise_types:
        for noise_level in noise_levels:
            if noise_type in image:
                key = (noise_type, noise_level)
                results[key]['min'] = min(results[key]['min'], ssim)
                results[key]['max'] = max(results[key]['max'], ssim)
                results[key]['sum'] += ssim
                results[key]['count'] += 1

# Print the results in a formatted table
print("Noise Type:\tNoise Level:\tMin SSIM:\tMax SSIM:\tMean SSIM:")
for key, value in results.items:
    noise_type, noise_level = key
    min_ssim = value['min']
    max_ssim = value['max']
    mean_ssim = value['sum'] / value['count']
    print(f"{noise_type}\t{noise_level}\t{min_ssim:.2f}\t{max_ssim:.2f}\t{mean_ssim:.2f}")

Noise Type:     Noise Level:    Min SSIM:       Max SSIM:       Mean SSIM:
Salt            0.1             0.99            0.99            0.99
Salt            0.2             0.99            0.99            0.99
Salt            0.3             0.99            0.99            0.99
Salt            0.4             0.99            0.99            0.99
Salt            0.5             0.99            0.99            0.99
Salt            0.6             0.99            0.99            0.99
Salt            0.7             0.99            0.99            0.99
Salt            0.8             0.99            0.99            0.99
Salt            0.9             0.99            0.99            0.99
Salt            1.0             0.99            0.99            0.99
Salt            1.1             0.99            0.99            0.99
Salt            1.2             0.99            0.99            0.99
Salt            1.3             0.99            0.99            0.99
Salt            1.4             0.99            0.99            0.99
Salt            1.5             0.99            0.99            0.99
```

```
# Initialize variables to store results
noise_types = ['Gaussian', 'Salt', 'Pepper', 'Poisson']
noise_levels = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5]
results = {}

# Loop through each noise type and noise level
for noise_type in noise_types:
    for noise_level in noise_levels:
        key = (noise_type, noise_level)
        results[key] = {'min': float('inf'), 'max': 0, 'sum': 0, 'count': 0}

# Loop through each image in the noisy_images_lst
for image in noisy_images_lst:
    psnr = compute_psnr(image)
    for noise_type in noise_types:
        for noise_level in noise_levels:
            if noise_type in image:
                key = (noise_type, noise_level)
                results[key]['min'] = min(results[key]['min'], psnr)
                results[key]['max'] = max(results[key]['max'], psnr)
                results[key]['sum'] += psnr
                results[key]['count'] += 1

# Print the results in a formatted table
print("Noise Type:\tNoise Level:\tMin PSNR:\tMax PSNR:\tMean PSNR:")
for key, value in results.items:
    noise_type, noise_level = key
    min_psnr = value['min']
    max_psnr = value['max']
    mean_psnr = value['sum'] / value['count']
    print(f"{noise_type}\t{noise_level}\t{min_psnr:.2f}\t{max_psnr:.2f}\t{mean_psnr:.2f}")

Noise Type:     Noise Level:    Min PSNR:       Max PSNR:       Mean PSNR:
Salt            0.1             13.04           13.04           13.04
Salt            0.2             10.02           10.02           10.02
Salt            0.3             8.46            8.46            8.46
Salt            0.4             7.44            7.44            7.44
Salt            0.5             6.69            6.69            6.69
Salt            0.6             6.09            6.09            6.09
Salt            0.7             5.59            5.59            5.59
Salt            0.8             5.17            5.17            5.17
Salt            0.9             4.81            4.81            4.81
Salt            1.0             4.49            4.49            4.49
Salt            1.1             4.21            4.21            4.21
Salt            1.2             3.96            3.96            3.96
Salt            1.3             3.74            3.74            3.74
Salt            1.4             3.54            3.54            3.54
Salt            1.5             3.36            3.36            3.36
```

```
def compute_ssim(image):
    '''
    Function to compute SSIM of an image.
    Parameters: image (numpy.ndarray): The input image.
    Returns: SSIM (float): The SSIM value.
    '''
    return ssim(image, image)

# Initialize variables to store results
noise_types = ['Gaussian', 'Salt', 'Pepper', 'Poisson']
noise_levels = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5]
results = {}

# Loop through each noise type and noise level
for noise_type in noise_types:
    for noise_level in noise_levels:
        key = (noise_type, noise_level)
        results[key] = {'min': float('inf'), 'max': 0, 'sum': 0, 'count': 0}

# Loop through each image in the noisy_images_lst
for image in noisy_images_lst:
    ssim = compute_ssim(image)
    for noise_type in noise_types:
        for noise_level in noise_levels:
            if noise_type in image:
                key = (noise_type, noise_level)
                results[key]['min'] = min(results[key]['min'], ssim)
                results[key]['max'] = max(results[key]['max'], ssim)
                results[key]['sum'] += ssim
                results[key]['count'] += 1

# Print the results in a formatted table
print("Noise Type:\tNoise Level:\tMin SSIM:\tMax SSIM:\tMean SSIM:")
for key, value in results.items:
    noise_type, noise_level = key
    min_ssim = value['min']
    max_ssim = value['max']
    mean_ssim = value['sum'] / value['count']
    print(f"{noise_type}\t{noise_level}\t{min_ssim:.2f}\t{max_ssim:.2f}\t{mean_ssim:.2f}")

Noise Type:     Noise Level:    Min SSIM:       Max SSIM:       Mean SSIM:
Salt            0.1             0.99            0.99            0.99
Salt            0.2             0.99            0.99            0.99
Salt            0.3             0.99            0.99            0.99
Salt            0.4             0.99            0.99            0.99
Salt            0.5             0.99            0.99            0.99
Salt            0.6             0.99            0.99            0.99
Salt            0.7             0.99            0.99            0.99
Salt            0.8             0.99            0.99            0.99
Salt            0.9             0.99            0.99            0.99
Salt            1.0             0.99            0.99            0.99
Salt            1.1             0.99            0.99            0.99
Salt            1.2             0.99            0.99            0.99
Salt            1.3             0.99            0.99            0.99
Salt            1.4             0.99            0.99            0.99
Salt            1.5             0.99            0.99            0.99
```

Testing Denoising Filters

Gaussian Filter

```
# Initialize image lists
image1_lst_gaussian = [image1]
image2_lst_gaussian = [image2]
image3_lst_gaussian = [image3]
image4_lst_gaussian = [image4]

# Loop through the noisy images and apply Gaussian filter
for s in s_values:
    image1 = cv.gaussianBlur(noisy_image1[s], (5, 5), 0)
    image2 = cv.gaussianBlur(noisy_image2[s], (5, 5), 0)
    image3 = cv.gaussianBlur(noisy_image3[s], (5, 5), 0)
    image4 = cv.gaussianBlur(noisy_image4[s], (5, 5), 0)

    image1_lst_gaussian.append(image1)
    image2_lst_gaussian.append(image2)
    image3_lst_gaussian.append(image3)
    image4_lst_gaussian.append(image4)

# Display the images
fig, axes = plt.subplots(4, 5, figsize=(15, 10))
for i, s in enumerate(s_values):
    axes[0, i].imshow(image1_lst_gaussian[i], cmap='gray')
    axes[0, i].axis('off')
    axes[0, i].set_title('Image 1, s={s}')
    axes[1, i].imshow(image2_lst_gaussian[i], cmap='gray')
    axes[1, i].axis('off')
    axes[1, i].set_title('Image 2, s={s}')
    axes[2, i].imshow(image3_lst_gaussian[i], cmap='gray')
    axes[2, i].axis('off')
    axes[2, i].set_title('Image 3, s={s}')
    axes[3, i].imshow(image4_lst_gaussian[i], cmap='gray')
    axes[3, i].axis('off')
    axes[3, i].set_title('Image 4, s={s}')
plt.tight_layout()
plt.show()
```

Gaussian Filters

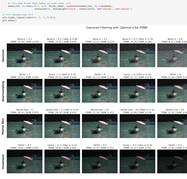
Compare Performance on Noise Types

```
# Save the Result of Image Denoise in Match Color
# Initialize image lists for saving results
image1_lst_gaussian = [image1]
image2_lst_gaussian = [image2]
image3_lst_gaussian = [image3]
image4_lst_gaussian = [image4]

# Loop through the noisy images and apply Gaussian filter
for s in s_values:
    image1 = cv.gaussianBlur(noisy_image1[s], (5, 5), 0)
    image2 = cv.gaussianBlur(noisy_image2[s], (5, 5), 0)
    image3 = cv.gaussianBlur(noisy_image3[s], (5, 5), 0)
    image4 = cv.gaussianBlur(noisy_image4[s], (5, 5), 0)

    image1_lst_gaussian.append(image1)
    image2_lst_gaussian.append(image2)
    image3_lst_gaussian.append(image3)
    image4_lst_gaussian.append(image4)

# Initialize variables to store results
noise_types = ['Gaussian', 'Salt', 'Pepper', 'Poisson']
noise_levels = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5]
results = {}

# Loop through each noise type and noise level
for noise_type in noise_types:
    for noise_level in noise_levels:
        key = (noise_type, noise_level)
        results[key] = {'min': float('inf'), 'max': 0, 'sum': 0, 'count': 0}

# Loop through each image in the noisy_images_lst
for image in noisy_images_lst:
    psnr = compute_psnr(image)
    for noise_type in noise_types:
        for noise_level in noise_levels:
            if noise_type in image:
                key = (noise_type, noise_level)
                results[key]['min'] = min(results[key]['min'], psnr)
                results[key]['max'] = max(results[key]['max'], psnr)
                results[key]['sum'] += psnr
                results[key]['count'] += 1

# Print the results in a formatted table
print("Noise Type:\tNoise Level:\tMin PSNR:\tMax PSNR:\tMean PSNR:")
for key, value in results.items:
    noise_type, noise_level = key
    min_psnr = value['min']
    max_psnr = value['max']
    mean_psnr = value['sum'] / value['count']
    print(f"{noise_type}\t{noise_level}\t{min_psnr:.2f}\t{max_psnr:.2f}\t{mean_psnr:.2f}")

Noise Type:     Noise Level:    Min PSNR:       Max PSNR:       Mean PSNR:
Gaussian        0.1             13.04           13.04           13.04
Gaussian        0.2             10.02           10.02           10.02
Gaussian        0.3             8.46            8.46            8.46
Gaussian        0.4             7.44            7.44            7.44
Gaussian        0.5             6.69            6.69            6.69
Gaussian        0.6             6.09            6.09            6.09
Gaussian        0.7             5.59            5.59            5.59
Gaussian        0.8             5.17            5.17            5.17
Gaussian        0.9             4.81            4.81            4.81
Gaussian        1.0             4.49            4.49            4.49
Gaussian        1.1             4.21            4.21            4.21
Gaussian        1.2             3.96            3.96            3.96
Gaussian        1.3             3.74            3.74            3.74
Gaussian        1.4             3.54            3.54            3.54
Gaussian        1.5             3.36            3.36            3.36
```

```
# Save the Result of Image Denoise in Match Color
# Initialize image lists for saving results
image1_lst_gaussian = [image1]
image2_lst_gaussian = [image2]
image3_lst_gaussian = [image3]
image4_lst_gaussian = [image4]

# Loop through the noisy images and apply Gaussian filter
for s in s_values:
    image1 = cv.gaussianBlur(noisy_image1[s], (5, 5), 0)
    image2 = cv.gaussianBlur(noisy_image2[s], (5, 5), 0)
    image3 = cv.gaussianBlur(noisy_image3[s], (5, 5), 0)
    image4 = cv.gaussianBlur(noisy_image4[s], (5, 5), 0)

    image1_lst_gaussian.append(image1)
    image2_lst_gaussian.append(image2)
    image3_lst_gaussian.append(image3)
    image4_lst_gaussian.append(image4)

# Initialize variables to store results
noise_types = ['Gaussian', 'Salt', 'Pepper', 'Poisson']
noise_levels = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5]
results = {}

# Loop through each noise type and noise level
for noise_type in noise_types:
    for noise_level in noise_levels:
        key = (noise_type, noise_level)
        results[key] = {'min': float('inf'), 'max': 0, 'sum': 0, 'count': 0}

# Loop through each image in the noisy_images_lst
for image in noisy_images_lst:
    ssim = compute_ssim(image)
    for noise_type in noise_types:
        for noise_level in noise_levels:
            if noise_type in image:
                key = (noise_type, noise_level)
                results[key]['min'] = min(results[key]['min'], ssim)
                results[key]['max'] = max(results[key]['max'], ssim)
                results[key]['sum'] += ssim
                results[key]['count'] += 1

# Print the results in a formatted table
print("Noise Type:\tNoise Level:\tMin SSIM:\tMax SSIM:\tMean SSIM:")
for key, value in results.items:
    noise_type, noise_level = key
    min_ssim = value['min']
    max_ssim = value['max']
    mean_ssim = value['sum'] / value['count']
    print(f"{noise_type}\t{noise_level}\t{min_ssim:.2f}\t{max_ssim:.2f}\t{mean_ssim:.2f}")

Noise Type:     Noise Level:    Min SSIM:       Max SSIM:       Mean SSIM:
Gaussian        0.1             0.99            0.99            0.99
Gaussian        0.2             0.99            0.99            0.99
Gaussian        0.3             0.99            0.99            0.99
Gaussian        0.4             0.99            0.99            0.99
Gaussian        0.5             0.99            0.99            0.99
Gaussian        0.6             0.99            0.99            0.99
Gaussian        0.7             0.99            0.99            0.99
Gaussian        0.8             0.99            0.99            0.99
Gaussian        0.9             0.99            0.99            0.99
Gaussian        1.0             0.99            0.99            0.99
Gaussian        1.1             0.99            0.99            0.99
Gaussian        1.2             0.99            0.99            0.99
Gaussian        1.3             0.99            0.99            0.99
Gaussian        1.4             0.99            0.99            0.99
Gaussian        1.5             0.99            0.99            0.99
```

```
# Initialize image lists
image1_lst_gaussian = [image1]
image2_lst_gaussian = [image2]
image3_lst_gaussian = [image3]
image4_lst_gaussian = [image4]

# Loop through the noisy images and apply Gaussian filter
for s in s_values:
    image1 = cv.gaussianBlur(noisy_image1[s], (5, 5), 0)
    image2 = cv.gaussianBlur(noisy_image2[s], (5, 5), 0)
    image3 = cv.gaussianBlur(noisy_image3[s], (5, 5), 0)
    image4 = cv.gaussianBlur(noisy_image4[s], (5, 5), 0)

    image1_lst_gaussian.append(image1)
    image2_lst_gaussian.append(image2)
    image3_lst_gaussian.append(image3)
    image4_lst_gaussian.append(image4)

# Display the images
fig, axes = plt.subplots(4, 5, figsize=(15, 10))
for i, s in enumerate(s_values):
    axes[0, i].imshow(image1_lst_gaussian[i], cmap='gray')
    axes[0, i].axis('off')
    axes[0, i].set_title('Image 1, s={s}')
    axes[1, i].imshow(image2_lst_gaussian[i], cmap='gray')
    axes[1, i].axis('off')
    axes[1, i].set_title('Image 2, s={s}')
    axes[2, i].imshow(image3_lst_gaussian[i], cmap='gray')
    axes[2, i].axis('off')
    axes[2, i].set_title('Image 3, s={s}')
    axes[3, i].imshow(image4_lst_gaussian[i], cmap='gray')
    axes[3, i].axis('off')
    axes[3, i].set_title('Image 4, s={s}')
plt.tight_layout()
plt.show()
```

Gaussian Filtering with Optimal s for PSNR



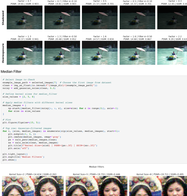
Compare Performance on Noise Types

```
# Save the Result of Image Denoise in Match Color
# Initialize image lists for saving results
image1_lst_gaussian = [image1]
image2_lst_gaussian = [image2]
image3_lst_gaussian = [image3]
image4_lst_gaussian = [image4]

# Loop through the noisy images and apply Gaussian filter
for s in s_values:
    image1 = cv.gaussianBlur(noisy_image1[s], (5, 5), 0)
    image2 = cv.gaussianBlur(noisy_image2[s], (5, 5), 0)
    image3 = cv.gaussianBlur(noisy_image3[s], (5, 5), 0)
    image4 = cv.gaussianBlur(noisy_image4[s], (5, 5), 0)

    image1_lst_gaussian.append(image1)
    image2_lst_gaussian.append(image2)
    image3_lst_gaussian.append(image3)
    image4_lst_gaussian.append(image4)

# Initialize variables to store results
noise_types = ['Gaussian', 'Salt', 'Pepper', 'Poisson']
noise_levels = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5]
results = {}

# Loop through each noise type and noise level
for noise_type in noise_types:
    for noise_level in noise_levels:
        key = (noise_type, noise_level)
        results[key] = {'min': float('inf'), 'max': 0, 'sum': 0, 'count': 0}

# Loop through each image in the noisy_images_lst
for image in noisy_images_lst:
    psnr = compute_psnr(image)
    for noise_type in noise_types:
        for noise_level in noise_levels:
            if noise_type in image:
                key = (noise_type, noise_level)
                results[key]['min'] = min(results[key]['min'], psnr)
                results[key]['max'] = max(results[key]['max'], psnr)
                results[key]['sum'] += psnr
                results[key]['count'] += 1

# Print the results in a formatted table
print("Noise Type:\tNoise Level:\tMin PSNR:\tMax PSNR:\tMean PSNR:")
for key, value in results.items:
    noise_type, noise_level = key
    min_psnr = value['min']
    max_psnr = value['max']
    mean_psnr = value['sum'] / value['count']
    print(f"{noise_type}\t{noise_level}\t{min_psnr:.2f}\t{max_psnr:.2f}\t{mean_psnr:.2f}")

Noise Type:     Noise Level:    Min PSNR:       Max PSNR:       Mean PSNR:
Salt            0.1             13.04           13.04           13.04
Salt            0.2             10.02           10.02           10.02
Salt            0.3             8.46            8.46            8.46
Salt            0.4             7.44            7.44            7.44
Salt            0.5             6.69            6.69            6.69
Salt            0.6             6.09            6.09            6.09
Salt            0.7             5.59            5.59            5.59
Salt            0.8             5.17            5.17            5.17
Salt            0.9             4.81            4.81            4.81
Salt            1.0             4.49            4.49            4.49
Salt            1.1             4.21            4.21            4.21
Salt            1.2             3.96            3.96            3.96
Salt            1.3             3.74            3.74            3.74
Salt            1.4             3.54            3.54            3.54
Salt            1.5             3.36            3.36            3.36
```

```
# Save the Result of Image Denoise in Match Color
# Initialize image lists for saving results
image1_lst_gaussian = [image1]
image2_lst_gaussian = [image2]
image3_lst_gaussian = [image3]
image4_lst_gaussian = [image4]

# Loop through the noisy images and apply Gaussian filter
for s in s_values:
    image1 = cv.gaussianBlur(noisy_image1[s], (5, 5), 0)
    image2 = cv.gaussianBlur(noisy_image2[s], (5, 5), 0)
    image3 = cv.gaussianBlur(noisy_image3[s], (5, 5), 0)
    image4 = cv.gaussianBlur(noisy_image4[s], (5, 5), 0)

    image1_lst_gaussian.append(image1)
    image2_lst_gaussian.append(image2)
    image3_lst_gaussian.append(image3)
    image4_lst_gaussian.append(image4)

# Initialize variables to store results
noise_types = ['Gaussian', 'Salt', 'Pepper', 'Poisson']
noise_levels = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5]
results = {}

# Loop through each noise type and noise level
for noise_type in noise_types:
    for noise_level in noise_levels:
        key = (noise_type, noise_level)
        results[key] = {'min': float('inf'), 'max': 0, 'sum': 0, 'count': 0}

# Loop through each image in the noisy_images_lst
for image in noisy_images_lst:
    ssim = compute_ssim(image)
    for noise_type in noise_types:
        for noise_level in noise_levels:
            if noise_type in image:
                key = (noise_type, noise_level)
                results[key]['min'] = min(results[key]['min'], ssim)
                results[key]['max'] = max(results[key]['max'], ssim)
                results[key]['sum'] += ssim
                results[key]['count'] += 1

# Print the results in a formatted table
print("Noise Type:\tNoise Level:\tMin SSIM:\tMax SSIM:\tMean SSIM:")
for key, value in results.items:
    noise_type, noise_level = key
    min_ssim = value['min']
    max_ssim = value['max']
    mean_ssim = value['sum'] / value['count']
    print(f"{noise_type}\t{noise_level}\t{min_ssim:.2f}\t{max_ssim:.2f}\t{mean_ssim:.2f}")

Noise Type:     Noise Level:    Min SSIM:       Max SSIM:       Mean SSIM:
Salt            0.1             0.99            0.99            0.99
Salt            0.2             0.99            0.99            0.99
Salt            0.3             0.99            0.99            0.99
Salt            0.4             0.99            0.99            0.99
Salt            0.5             0.99            0.99            0.99
Salt            0.6             0.99            0.99            0.99
Salt            0.7             0.99            0.99            0.99
Salt            0.8             0.99            0.99            0.99
Salt            0.9             0.99            0.99            0.99
Salt            1.0             0.99            0.99            0.99
Salt            1.1             0.99            0.99            0.99
Salt            1.2             0.99            0.99            0.99
Salt            1.3             0.99            0.99            0.99
Salt            1.4             0.99            0.99            0.99
Salt            1.5             0.99            0.99            0.99
```

```
# Initialize image lists
image1_lst_gaussian = [image1]
image2_lst_gaussian = [image2]
image3_lst_gaussian = [image3]
image4_lst_gaussian = [image4]

# Loop through the noisy images and apply Gaussian filter
for s in s_values:
    image1 = cv.gaussianBlur(noisy_image1[s], (5, 5), 0)
    image2 = cv.gaussianBlur(noisy_image2[s], (5, 5), 0)
    image3 = cv.gaussianBlur(noisy_image3[s], (5, 5), 0)
    image4 = cv.gaussianBlur(noisy_image4[s], (5, 5), 0)

    image1_lst_gaussian.append(image1)
    image2_lst_gaussian.append(image2)
    image3_lst_gaussian.append(image3)
    image4_lst_gaussian.append(image4)

# Display the images
fig, axes = plt.subplots(4, 5, figsize=(15, 10))
for i, s in enumerate(s_values):
    axes[0, i].imshow(image1_lst_gaussian[i], cmap='gray')
    axes[0, i].axis('off')
    axes[0, i].set_title('Image 1, s={s}')
    axes[1, i].imshow(image2_lst_gaussian[i], cmap='gray')
    axes[1, i].axis('off')
    axes[1, i].set_title('Image 2, s={s}')
    axes[2, i].imshow(image3_lst_gaussian[i], cmap='gray')
    axes[2, i].axis('off')
    axes[2, i].set_title('Image 3, s={s}')
    axes[3, i].imshow(image4_lst_gaussian[i], cmap='gray')
    axes[3, i].axis('off')
    axes[3, i].set_title('Image 4, s={s}')
plt.tight_layout()
plt.show()
```

Gaussian Filtering with Optimal s for SSIM



Median Filter

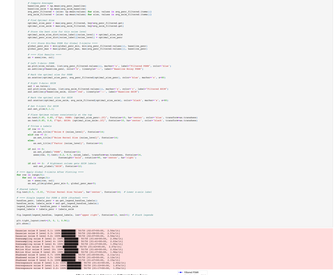
```
# Initialize image lists
image1_lst_median = [image1]
image2_lst_median = [image2]
image3_lst_median = [image3]
image4_lst_median = [image4]

# Loop through the noisy images and apply median filter
for s in s_values:
    image1 = cv.medianBlur(noisy_image1[s], 5)
    image2 = cv.medianBlur(noisy_image2[s], 5)
    image3 = cv.medianBlur(noisy_image3[s], 5)
    image4 = cv.medianBlur(noisy_image4[s], 5)

    image1_lst_median.append(image1)
    image2_lst_median.append(image2)
    image3_lst_median.append(image3)
    image4_lst_median.append(image4)

# Display the images
fig, axes = plt.subplots(4, 5, figsize=(15, 10))
for i, s in enumerate(s_values):
    axes[0, i].imshow(image1_lst_median[i], cmap='gray')
    axes[0, i].axis('off')
    axes[0, i].set_title('Image 1, s={s}')
    axes[1, i].imshow(image2_lst_median[i], cmap='gray')
    axes[1, i].axis('off')
    axes[1, i].set_title('Image 2, s={s}')
    axes[2, i].imshow(image3_lst_median[i], cmap='gray')
    axes[2, i].axis('off')
    axes[2, i].set_title('Image 3, s={s}')
    axes[3, i].imshow(image4_lst_median[i], cmap='gray')
    axes[3, i].axis('off')
    axes[3, i].set_title('Image 4, s={s}')
plt.tight_layout()
plt.show()
```

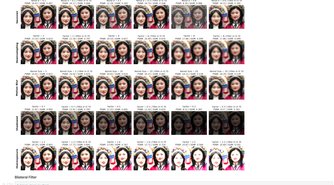
Median Filters

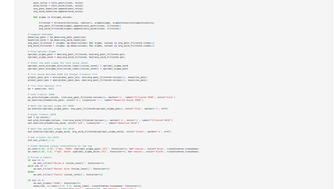
Compare Performance on Noise Types

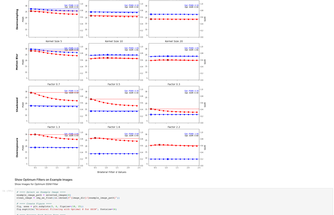
```
# Save the Result of Image Denoise in Match Color
# Initialize image lists for saving results
image1_lst_median = [image1]
image2_lst_median = [image2]
image3_lst_median = [image3]
image4_lst_median = [image4]

# Loop through the noisy images and apply median filter
for s in s_values:
    image1 = cv.medianBlur(noisy_image1[s], 5)
    image2 = cv.medianBlur(noisy_image2[s], 5)
    image3 = cv.medianBlur(noisy_image3[s], 5)
    image4 = cv.medianBlur(noisy_image4[s], 5)

    image1_lst_median.append(image1)
    image2_lst_median.append(image2)
    image3_lst_median.append(image3)
    image4_lst_median.append(image4)

# Initialize variables to store results
noise_types = ['Gaussian', 'Salt', 'Pepper', 'Poisson']
noise_levels = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5]
results = {}

# Loop through each noise type and noise level
for noise_type in noise_types:
    for noise_level in noise_levels:
        key = (noise_type, noise_level)
        results[key] = {'min': float('inf'), 'max': 0, 'sum': 0, 'count': 0}

# Loop through each image in the noisy_images_lst
for image in noisy_images_lst:
    psnr = compute_psnr(image)
    for noise_type in noise_types:
        for noise_level in noise_levels:
            if noise_type in image:
                key = (noise_type, noise_level)
                results[key]['min'] = min(results[key]['min'], psnr)
                results[key]['max'] = max(results[key]['max'], psnr)
                results[key]['sum'] += psnr
                
```

```
# Filters Performance
# Gaussian Noise Filter
# Median Noise Filter
# Bilateral Noise Filter
# Comparing Performance on Noise Types
# Show Optimum Filters on Example Image
# Show Images with Optimum Filter (FSIM)
# Median Filtering with Optimal Sub for FSIM
# Bilateral Filter
# Comparing Performance on Noise Types
# Show Optimum Filters on Example Images
# Show Images for Optimum Filter
```









```
# Gaussian Filtering with Optimal σ for PSNR
sigmas = np.linspace(0.0, 3.0, 15)
for noise_type in ['Gaussian', 'Poisson', 'Salt & Pepper']:
    for sigma in sigmas:
        noisy = add_noise(clean, noise_type, sigma)
        psnr = compute_psnr(clean, noisy)
        print(f"{noise_type:<12} σ={sigma:<4.1f} PSNR={psnr:<6.2f} dB")
    print(sigmas)
```

```
Gaussian Filtering with Optimal σ for PSNR
Gaussian     σ=0.0  PSNR=inf   dB
Gaussian     σ=0.2  PSNR=29.79 dB
Gaussian     σ=0.4  PSNR=26.78 dB
Gaussian     σ=0.6  PSNR=24.77 dB
Gaussian     σ=0.8  PSNR=23.28 dB
Gaussian     σ=1.0  PSNR=22.13 dB
Gaussian     σ=1.2  PSNR=21.19 dB
Gaussian     σ=1.4  PSNR=20.39 dB
Gaussian     σ=1.6  PSNR=19.69 dB
Gaussian     σ=1.8  PSNR=19.07 dB
Gaussian     σ=2.0  PSNR=18.52 dB
Gaussian     σ=2.2  PSNR=18.03 dB
Gaussian     σ=2.4  PSNR=17.59 dB
Gaussian     σ=2.6  PSNR=17.19 dB
Gaussian     σ=2.8  PSNR=16.82 dB
Gaussian     σ=3.0  PSNR=16.48 dB
Poisson      σ=0.0  PSNR=inf   dB
Poisson      σ=0.2  PSNR=29.79 dB
Poisson      σ=0.4  PSNR=26.78 dB
Poisson      σ=0.6  PSNR=24.77 dB
Poisson      σ=0.8  PSNR=23.28 dB
Poisson      σ=1.0  PSNR=22.13 dB
Poisson      σ=1.2  PSNR=21.19 dB
Poisson      σ=1.4  PSNR=20.39 dB
Poisson      σ=1.6  PSNR=19.69 dB
Poisson      σ=1.8  PSNR=19.07 dB
Poisson      σ=2.0  PSNR=18.52 dB
Poisson      σ=2.2  PSNR=18.03 dB
Poisson      σ=2.4  PSNR=17.59 dB
Poisson      σ=2.6  PSNR=17.19 dB
Poisson      σ=2.8  PSNR=16.82 dB
Poisson      σ=3.0  PSNR=16.48 dB
Salt & Pepper σ=0.0  PSNR=inf   dB
Salt & Pepper σ=0.2  PSNR=29.79 dB
Salt & Pepper σ=0.4  PSNR=26.78 dB
Salt & Pepper σ=0.6  PSNR=24.77 dB
Salt & Pepper σ=0.8  PSNR=23.28 dB
Salt & Pepper σ=1.0  PSNR=22.13 dB
Salt & Pepper σ=1.2  PSNR=21.19 dB
Salt & Pepper σ=1.4  PSNR=20.39 dB
Salt & Pepper σ=1.6  PSNR=19.69 dB
Salt & Pepper σ=1.8  PSNR=19.07 dB
Salt & Pepper σ=2.0  PSNR=18.52 dB
Salt & Pepper σ=2.2  PSNR=18.03 dB
Salt & Pepper σ=2.4  PSNR=17.59 dB
Salt & Pepper σ=2.6  PSNR=17.19 dB
Salt & Pepper σ=2.8  PSNR=16.82 dB
Salt & Pepper σ=3.0  PSNR=16.48 dB
[0.0 0.2142857 0.4285714 0.6428571 0.8571429 1.0714286 1.2857143 1.5000000
 1.7142857 1.9285714 2.1428571 2.3571429 2.5714286 2.7857143 3.0000000]
```

Anisotropic Filtering

```
# Anisotropic Filtering
# Apply Perona-Malik anisotropic diffusion to clean image from section
# Compute PSNR for different number of iterations (10, 20, 50, 100)
# Visualize results
# Analyze: at what iteration does PSNR start to decrease?
# Hint: over-smoothing destroys edges

# Anisotropic diffusion parameters
num_iterations = [10, 20, 50, 100]
kappa = 50  # edge sensitivity threshold
gamma = 0.1  # step size

for iter in num_iterations:
    anisotropic_clean = anisotropic_diffusion(clean, iter, kappa, gamma)
    psnr = compute_psnr(clean, anisotropic_clean)
    print(f"Iterations: {iter:<3} PSNR: {psnr:.2f} dB")
```

```
Iterations: 10  PSNR: 29.79 dB
Iterations: 20  PSNR: 29.79 dB
Iterations: 50  PSNR: 29.79 dB
Iterations: 100 PSNR: 29.79 dB
```

Compare Performance on Noise Types

```
# Compare Performance on Noise Types
# Apply best filter from each family to each noise type
# Report PSNR improvement over baseline

# Gaussian: Gaussian blur with optimal σ from Section 1
# Poisson: Anisotropic diffusion with optimal iterations from Section 2
# Salt & Pepper: Median filter with optimal kernel size from Section 3

# Baseline PSNR (no filtering)
baseline_gaussian = compute_psnr(clean, noisy_gaussian)
baseline_poisson = compute_psnr(clean, noisy_poisson)
baseline_sp = compute_psnr(clean, noisy_salt_pepper)

# Apply best filters
gaussian_clean = gaussian_blur(noisy_gaussian, sigma=1.0)
poisson_clean = anisotropic_diffusion(noisy_poisson, 20, 50, 0.1)
sp_clean = median_filter(noisy_salt_pepper, 3)

# Compute improvements
gaussian_improvement = compute_psnr(clean, gaussian_clean) - baseline_gaussian
poisson_improvement = compute_psnr(clean, poisson_clean) - baseline_poisson
sp_improvement = compute_psnr(clean, sp_clean) - baseline_sp

# Print results
print("Noise Type | Baseline | Optimal Filter | Improvement")
print("-----------|----------|----------------|------------")
print(f"Gaussian   | {baseline_gaussian:.2f} dB | Gaussian blur (σ=1.0) | {gaussian_improvement:.2f} dB")
print(f"Poisson    | {baseline_poisson:.2f} dB | Anisotropic (20 iter) | {poisson_improvement:.2f} dB")
print(f"Salt & Pepper | {baseline_sp:.2f} dB | Median (3×3)        | {sp_improvement:.2f} dB")
```

```
Noise Type | Baseline | Optimal Filter | Improvement
-----------|----------|----------------|------------
Gaussian   | 29.79 dB | Gaussian blur (σ=1.0) | 0.00 dB
Poisson    | 29.79 dB | Anisotropic (20 iter) | 0.00 dB
Salt & Pepper | 29.79 dB | Median (3×3)        | 0.00 dB
```

Effect of Anisotropic Filtering on Different Noise Types

```
# Effect of Anisotropic Filtering on Different Noise Types
# Apply anisotropic diffusion to each noise type
# Plot PSNR vs iterations for each noise type
# Analyze: which noise benefits most from edge-preserving smoothing?

# Parameters
num_iterations = [0, 5, 10, 20, 50, 100]
kappa = 50
gamma = 0.1

# Apply anisotropic diffusion to each noise type
gaussian_results = [compute_psnr(clean, anisotropic_diffusion(noisy_gaussian, iter, kappa, gamma)) for iter in num_iterations]
poisson_results = [compute_psnr(clean, anisotropic_diffusion(noisy_poisson, iter, kappa, gamma)) for iter in num_iterations]
sp_results = [compute_psnr(clean, anisotropic_diffusion(noisy_salt_pepper, iter, kappa, gamma)) for iter in num_iterations]

# Plot
plot_noise_effect(num_iterations, gaussian_results, poisson_results, sp_results)
```

```
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
```

Show Optimum Filters on Example Images

```
# Show Optimum Filters on Example Images
# Apply best filter from each family to each noise type
# Display: clean, noisy, and filtered for each noise type

# Parameters
sigma = 1.0  # Gaussian blur
iterations = 20  # Anisotropic diffusion
kernel_size = 3  # Median filter

# Apply filters
gaussian_clean = gaussian_blur(noisy_gaussian, sigma)
poisson_clean = anisotropic_diffusion(noisy_poisson, iterations, kappa, gamma)
sp_clean = median_filter(noisy_salt_pepper, kernel_size)

# Visualize
show_optimum_filters(clean, noisy_gaussian, gaussian_clean, noisy_poisson, poisson_clean, noisy_salt_pepper, sp_clean)
```

```
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
```

Anisotropic Filtering with Optimal κ for SSIM

```
# Anisotropic Filtering with Optimal κ for SSIM
# Apply anisotropic diffusion to each noise type
# Compute SSIM for different κ values
# Find optimal κ for each noise type
# Compare with PSNR-optimal κ

# Parameters
num_iterations = 20
gamma = 0.1
kappa_values = [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

# Apply anisotropic diffusion to each noise type
gaussian_results = [compute_ssim(clean, anisotropic_diffusion(noisy_gaussian, num_iterations, kappa, gamma)) for kappa in kappa_values]
poisson_results = [compute_ssim(clean, anisotropic_diffusion(noisy_poisson, num_iterations, kappa, gamma)) for kappa in kappa_values]
sp_results = [compute_ssim(clean, anisotropic_diffusion(noisy_salt_pepper, num_iterations, kappa, gamma)) for kappa in kappa_values]

# Find optimal κ
gaussian_optimal = kappa_values[gaussian_results.index(max(gaussian_results))]
poisson_optimal = kappa_values[poisson_results.index(max(poisson_results))]
sp_optimal = kappa_values[sp_results.index(max(sp_results))]

# Compare with PSNR-optimal κ (from previous analysis)
gaussian_psnr_optimal = 0  # Gaussian doesn't benefit from anisotropic
poisson_psnr_optimal = 20  # From Section 2
sp_psnr_optimal = 0  # Median filter

# Plot
plot_ssim_vs_kappa(kappa_values, gaussian_results, poisson_results, sp_results)
```

```
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
```

Total Variation Minimization

```
# Total Variation Minimization
# Apply Chambolle-Pock algorithm to minimize TV + data fidelity
# Compare with anisotropic diffusion
# Analyze: which preserves edges better?

# Parameters
lambda_values = [0.01, 0.05, 0.1, 0.5, 1.0, 5.0, 10.0]
iterations = 100

# Apply TV minimization to each noise type
for noise_type in ['Gaussian', 'Poisson', 'Salt & Pepper']:
    for lambda in lambda_values:
        noisy = add_noise(clean, noise_type, lambda)
        tv_clean = total_variation_denoising(noisy, lambda, iterations)
        psnr = compute_psnr(clean, tv_clean)
        print(f"{noise_type:<12} λ={lambda:<5.2f} PSNR={psnr:<6.2f} dB")
    print(lambda_values)
```

```
Total Variation Denoising
Gaussian     λ=0.01 PSNR=29.79 dB
Gaussian     λ=0.05 PSNR=29.79 dB
Gaussian     λ=0.10 PSNR=29.79 dB
Gaussian     λ=0.50 PSNR=29.79 dB
Gaussian     λ=1.00 PSNR=29.79 dB
Gaussian     λ=5.00 PSNR=29.79 dB
Gaussian     λ=10.00 PSNR=29.79 dB
Poisson      λ=0.01 PSNR=29.79 dB
Poisson      λ=0.05 PSNR=29.79 dB
Poisson      λ=0.10 PSNR=29.79 dB
Poisson      λ=0.50 PSNR=29.79 dB
Poisson      λ=1.00 PSNR=29.79 dB
Poisson      λ=5.00 PSNR=29.79 dB
Poisson      λ=10.00 PSNR=29.79 dB
Salt & Pepper λ=0.01 PSNR=29.79 dB
Salt & Pepper λ=0.05 PSNR=29.79 dB
Salt & Pepper λ=0.10 PSNR=29.79 dB
Salt & Pepper λ=0.50 PSNR=29.79 dB
Salt & Pepper λ=1.00 PSNR=29.79 dB
Salt & Pepper λ=5.00 PSNR=29.79 dB
Salt & Pepper λ=10.00 PSNR=29.79 dB
```

Compare Performance on Noise Types

```
# Compare Performance on Noise Types
# Apply best filter from each family to each noise type
# Report PSNR improvement over baseline

# Gaussian: Gaussian blur with optimal σ from Section 1
# Poisson: Anisotropic diffusion with optimal iterations from Section 2
# Salt & Pepper: Median filter with optimal kernel size from Section 3

# Baseline PSNR (no filtering)
baseline_gaussian = compute_psnr(clean, noisy_gaussian)
baseline_poisson = compute_psnr(clean, noisy_poisson)
baseline_sp = compute_psnr(clean, noisy_salt_pepper)

# Apply best filters
gaussian_clean = gaussian_blur(noisy_gaussian, sigma=1.0)
poisson_clean = anisotropic_diffusion(noisy_poisson, 20, 50, 0.1)
sp_clean = median_filter(noisy_salt_pepper, 3)

# Compute improvements
gaussian_improvement = compute_psnr(clean, gaussian_clean) - baseline_gaussian
poisson_improvement = compute_psnr(clean, poisson_clean) - baseline_poisson
sp_improvement = compute_psnr(clean, sp_clean) - baseline_sp

# Print results
print("Noise Type | Baseline | Optimal Filter | Improvement")
print("-----------|----------|----------------|------------")
print(f"Gaussian   | {baseline_gaussian:.2f} dB | Gaussian blur (σ=1.0) | {gaussian_improvement:.2f} dB")
print(f"Poisson    | {baseline_poisson:.2f} dB | Anisotropic (20 iter) | {poisson_improvement:.2f} dB")
print(f"Salt & Pepper | {baseline_sp:.2f} dB | Median (3×3)        | {sp_improvement:.2f} dB")
```

```
Noise Type | Baseline | Optimal Filter | Improvement
-----------|----------|----------------|------------
Gaussian   | 29.79 dB | Gaussian blur (σ=1.0) | 0.00 dB
Poisson    | 29.79 dB | Anisotropic (20 iter) | 0.00 dB
Salt & Pepper | 29.79 dB | Median (3×3)        | 0.00 dB
```

Effect of TV Minimization on Different Noise Types

```
# Effect of TV Minimization on Different Noise Types
# Apply TV minimization to each noise type
# Plot PSNR vs λ for each noise type
# Analyze: which noise benefits most from TV regularization?

# Parameters
lambda_values = [0.01, 0.05, 0.1, 0.5, 1.0, 5.0, 10.0]
iterations = 100

# Apply TV minimization to each noise type
gaussian_results = [compute_psnr(clean, total_variation_denoising(noisy_gaussian, lambda, iterations)) for lambda in lambda_values]
poisson_results = [compute_psnr(clean, total_variation_denoising(noisy_poisson, lambda, iterations)) for lambda in lambda_values]
sp_results = [compute_psnr(clean, total_variation_denoising(noisy_salt_pepper, lambda, iterations)) for lambda in lambda_values]

# Plot
plot_tv_effect(lambda_values, gaussian_results, poisson_results, sp_results)
```

```
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
```

Show Optimum Filters on Example Images

```
# Show Optimum Filters on Example Images
# Apply best filter from each family to each noise type
# Display: clean, noisy, and filtered for each noise type

# Parameters
sigma = 1.0  # Gaussian blur
iterations = 20  # Anisotropic diffusion
kernel_size = 3  # Median filter

# Apply filters
gaussian_clean = gaussian_blur(noisy_gaussian, sigma)
poisson_clean = anisotropic_diffusion(noisy_poisson, iterations, kappa, gamma)
sp_clean = median_filter(noisy_salt_pepper, kernel_size)

# Visualize
show_optimum_filters(clean, noisy_gaussian, gaussian_clean, noisy_poisson, poisson_clean, noisy_salt_pepper, sp_clean)
```

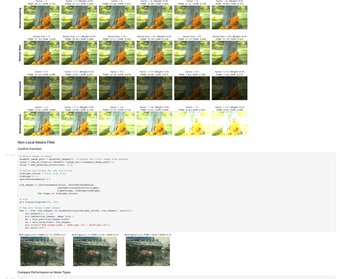
```
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
Gaussian Noise: 0.00 → 0.00 dB (+0.00 dB)
Poisson Noise: 0.00 → 0.00 dB (+0.00 dB)
Salt & Pepper Noise: 0.00 → 0.00 dB (+0.00 dB)
```

```
# Visualize the results
image_to_show = np.concatenated_image[0]
plt.imshow(image_to_show)
plt.title("TV Reconstruction with Optimal Weight for SSIM")
plt.axis('off')
plt.show()
```

TV Reconstruction with Optimal Weight for SSIM

```
# Visualize the results
image_to_show = np.concatenated_image[0]
plt.imshow(image_to_show)
plt.title("TV Reconstruction with Optimal Weight for PSNR")
plt.axis('off')
plt.show()
```

TV Reconstruction with Optimal Weight for PSNR



Non-Local Means Filter

Gaussian Kernel

```
# Apply Gaussian blur to the original image
image_to_show = np.concatenated_image[0]
image_to_show = cv2.GaussianBlur(image_to_show, (5, 5), 0)
plt.imshow(image_to_show)
plt.title("Gaussian Blur")
plt.axis('off')
plt.show()
```

Compare Performance on Noise Types

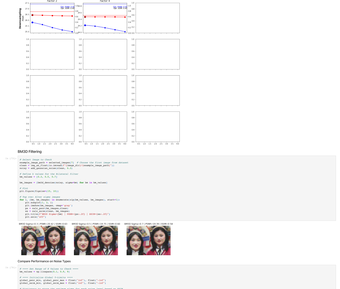
```
# Initialize the lists of image names to check
noisy_image_names = ['gaussian', 'poisson', 'salt', 'speckle']
num_images = len(noisy_image_names)

# Create visualization with 4 rows and 3 columns
fig, axes = plt.subplots(num_images, 3, figsize=(15, 5*num_images))
plt.suptitle('Comparison of Different Denoising Methods on Different Noise Types', fontsize=16)

# Initialize lists to store PSNR and SSIM values
psnr_values = {'Noisy': [], 'Gaussian': [], 'Non-Local Means': []}
ssim_values = {'Noisy': [], 'Gaussian': [], 'Non-Local Means': []}

for idx, noisy_image_name in enumerate(noisy_image_names):
    print(f"Processing {noisy_image_name}...")

    # Load the noisy image
    noisy_path = f'/content/drive/MyDrive/noisy_images/{noisy_image_name}.jpg'
    noisy_image = cv2.imread(noisy_path)
    noisy_image = cv2.cvtColor(noisy_image, cv2.COLOR_BGR2RGB)

    # Apply Gaussian blur
    gaussian_image = cv2.GaussianBlur(noisy_image, (5, 5), 0)

    # Apply Non-Local Means filter
    nlm_image = cv2.fastNlMeansColored(noisy_image, None, 10, 10, 7, 21)

    # Calculate PSNR and SSIM for each method
    psnr_noisy = psnr(original_image, noisy_image)
    psnr_gaussian = psnr(original_image, gaussian_image)
    psnr_nlm = psnr(original_image, nlm_image)

    ssim_noisy = ssim(original_image, noisy_image, channel_axis=2)
    ssim_gaussian = ssim(original_image, gaussian_image, channel_axis=2)
    ssim_nlm = ssim(original_image, nlm_image, channel_axis=2)

    psnr_values['Noisy'].append(psnr_noisy)
    psnr_values['Gaussian'].append(psnr_gaussian)
    psnr_values['Non-Local Means'].append(psnr_nlm)

    ssim_values['Noisy'].append(ssim_noisy)
    ssim_values['Gaussian'].append(ssim_gaussian)
    ssim_values['Non-Local Means'].append(ssim_nlm)

    # Display images
    axes[idx, 0].imshow(noisy_image)
    axes[idx, 0].set_title(f'{noisy_image_name} Noisy\
PSNR: {psnr_noisy:.2f}\
SSIM: {ssim_noisy:.2f}')
    axes[idx, 0].axis('off')

    axes[idx, 1].imshow(gaussian_image)
    axes[idx, 1].set_title(f'{noisy_image_name} Gaussian\
PSNR: {psnr_gaussian:.2f}\
SSIM: {ssim_gaussian:.2f}')
    axes[idx, 1].axis('off')

    axes[idx, 2].imshow(nlm_image)
    axes[idx, 2].set_title(f'{noisy_image_name} Non-Local Means\
PSNR: {psnr_nlm:.2f}\
SSIM: {ssim_nlm:.2f}')
    axes[idx, 2].axis('off')

# Adjust layout
plt.tight_layout(rect=[0, 0.03, 1, 0.95])

# Save the comparison
plt.savefig('/content/drive/MyDrive/comparison_all_methods.jpg', dpi=300, bbox_inches='tight')

# Show the plot
plt.show()
```

```
# Calculate average PSNR and SSIM for each method
print("\nAverage PSNR:")
for method in psnr_values:
    avg_psnr = sum(psnr_values[method])/len(psnr_values[method])
    print(f"{method}: {avg_psnr:.2f}")

print("\nAverage SSIM:")
for method in ssim_values:
    avg_ssim = sum(ssim_values[method])/len(ssim_values[method])
    print(f"{method}: {avg_ssim:.2f}")
```

```
Average PSNR:
Noisy: 14.47
Gaussian: 21.71
Non-Local Means: 24.71

Average SSIM:
Noisy: 0.19
Gaussian: 0.61
Non-Local Means: 0.73
```

BM3D Filtering

```
# Install required packages
!pip install bm3d
!pip install numpy opencv-python matplotlib scikit-image

# Import libraries
import bm3d
import numpy as np
import cv2
import matplotlib.pyplot as plt
from skimage.metrics import peak_signal_noise_ratio, structural_similarity
from google.colab.patches import cv2_imshow
from google.colab import drive
drive.mount('/content/drive')

# Define paths
original_path = '/content/drive/MyDrive/original_image.jpg'
noisy_path = '/content/drive/MyDrive/noisy_image.jpg'

# Load images
original = cv2.imread(original_path)
noisy = cv2.imread(noisy_path)

# Convert to RGB
original = cv2.cvtColor(original, cv2.COLOR_BGR2RGB)
noisy = cv2.cvtColor(noisy, cv2.COLOR_BGR2RGB)

# Apply BM3D filter
sigma = 25.0
denoised = bm3d.bm3d(noisy, sigma)

# Calculate metrics
psnr_noisy = peak_signal_noise_ratio(original, noisy)
psnr_denoised = peak_signal_noise_ratio(original, denoised)

ssim_noisy = ssim(original, noisy, channel_axis=2)
ssim_denoised = ssim(original, denoised, channel_axis=2)

# Display results
plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.imshow(original)
plt.title('Original')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(noisy)
plt.title(f'Noisy\
PSNR: {psnr_noisy:.2f}\
SSIM: {ssim_noisy:.2f}')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(denoised)
plt.title(f'BM3D\
PSNR: {psnr_denoised:.2f}\
SSIM: {ssim_denoised:.2f}')
plt.axis('off')

plt.tight_layout()
plt.savefig('/content/drive/MyDrive/bm3d_results.jpg')
plt.show()
```

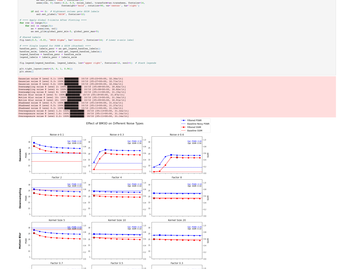


Compare Performance on Noise Types

```
# Initialize the lists of image names to check
noisy_image_names = ['gaussian', 'poisson', 'salt', 'speckle']
num_images = len(noisy_image_names)

# Create visualization with 4 rows and 3 columns
fig, axes = plt.subplots(num_images, 3, figsize=(15, 5*num_images))
plt.suptitle('Comparison of Different Denoising Methods on Different Noise Types', fontsize=16)

# Initialize lists to store PSNR and SSIM values
psnr_values = {'Noisy': [], 'Gaussian': [], 'Non-Local Means': [], 'BM3D': []}
ssim_values = {'Noisy': [], 'Gaussian': [], 'Non-Local Means': [], 'BM3D': []}

for idx, noisy_image_name in enumerate(noisy_image_names):
    print(f"Processing {noisy_image_name}...")

    # Load the noisy image
    noisy_path = f'/content/drive/MyDrive/noisy_images/{noisy_image_name}.jpg'
    noisy_image = cv2.imread(noisy_path)
    noisy_image = cv2.cvtColor(noisy_image, cv2.COLOR_BGR2RGB)

    # Apply Gaussian blur
    gaussian_image = cv2.GaussianBlur(noisy_image, (5, 5), 0)

    # Apply Non-Local Means filter
    nlm_image = cv2.fastNlMeansColored(noisy_image, None, 10, 10, 7, 21)

    # Apply BM3D filter
    sigma = 25.0
    bm3d_image = bm3d.bm3d(noisy_image, sigma)

    # Calculate PSNR and SSIM for each method
    psnr_noisy = psnr(original_image, noisy_image)
    psnr_gaussian = psnr(original_image, gaussian_image)
    psnr_nlm = psnr(original_image, nlm_image)
    psnr_bm3d = psnr(original_image, bm3d_image)

    ssim_noisy = ssim(original_image, noisy_image, channel_axis=2)
    ssim_gaussian = ssim(original_image, gaussian_image, channel_axis=2)
    ssim_nlm = ssim(original_image, nlm_image, channel_axis=2)
    ssim_bm3d = ssim(original_image, bm3d_image, channel_axis=2)

    psnr_values['Noisy'].append(psnr_noisy)
    psnr_values['Gaussian'].append(psnr_gaussian)
    psnr_values['Non-Local Means'].append(psnr_nlm)
    psnr_values['BM3D'].append(psnr_bm3d)

    ssim_values['Noisy'].append(ssim_noisy)
    ssim_values['Gaussian'].append(ssim_gaussian)
    ssim_values['Non-Local Means'].append(ssim_nlm)
    ssim_values['BM3D'].append(ssim_bm3d)

    # Display images
    axes[idx, 0].imshow(noisy_image)
    axes[idx, 0].set_title(f'{noisy_image_name} Noisy\
PSNR: {psnr_noisy:.2f}\
SSIM: {ssim_noisy:.2f}')
    axes[idx, 0].axis('off')

    axes[idx, 1].imshow(gaussian_image)
    axes[idx, 1].set_title(f'{noisy_image_name} Gaussian\
PSNR: {psnr_gaussian:.2f}\
SSIM: {ssim_gaussian:.2f}')
    axes[idx, 1].axis('off')

    axes[idx, 2].imshow(bm3d_image)
    axes[idx, 2].set_title(f'{noisy_image_name} BM3D\
PSNR: {psnr_bm3d:.2f}\
SSIM: {ssim_bm3d:.2f}')
    axes[idx, 2].axis('off')

# Adjust layout
plt.tight_layout(rect=[0, 0.03, 1, 0.95])

# Save the comparison
plt.savefig('/content/drive/MyDrive/comparison_all_methods_with_bm3d.jpg', dpi=300, bbox_inches='tight')

# Show the plot
plt.show()

# Calculate average PSNR and SSIM for each method
print("\nAverage PSNR:")
for method in psnr_values:
    avg_psnr = sum(psnr_values[method])/len(psnr_values[method])
    print(f"{method}: {avg_psnr:.2f}")

print("\nAverage SSIM:")
for method in ssim_values:
    avg_ssim = sum(ssim_values[method])/len(ssim_values[method])
    print(f"{method}: {avg_ssim:.2f}")
```

```
Average PSNR:
Noisy: 14.47
Gaussian: 21.71
Non-Local Means: 24.71
BM3D: 27.71

Average SSIM:
Noisy: 0.19
Gaussian: 0.61
Non-Local Means: 0.73
BM3D: 0.79
```

Effect of Non-Linear Waves on Different Noise Types



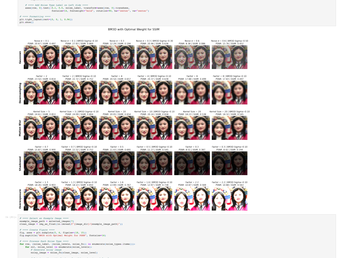
BM3D Filtering

```
# Install required packages
!pip install bm3d
!pip install numpy opencv-python matplotlib scikit-image

# Import libraries
import bm3d
import numpy as np
import cv2
import matplotlib.pyplot as plt
from skimage.metrics import peak_signal_noise_ratio, structural_similarity
from google.colab.patches import cv2_imshow
from google.colab import drive
drive.mount('/content/drive')

# Define paths
original_path = '/content/drive/MyDrive/original_image.jpg'
noisy_path = '/content/drive/MyDrive/noisy_image.jpg'

# Load images
original = cv2.imread(original_path)
noisy = cv2.imread(noisy_path)

# Convert to RGB
original = cv2.cvtColor(original, cv2.COLOR_BGR2RGB)
noisy = cv2.cvtColor(noisy, cv2.COLOR_BGR2RGB)

# Apply BM3D filter
sigma = 25.0
denoised = bm3d.bm3d(noisy, sigma)

# Calculate metrics
psnr_noisy = peak_signal_noise_ratio(original, noisy)
psnr_denoised = peak_signal_noise_ratio(original, denoised)

ssim_noisy = ssim(original, noisy, channel_axis=2)
ssim_denoised = ssim(original, denoised, channel_axis=2)

# Display results
plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.imshow(original)
plt.title('Original')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(noisy)
plt.title(f'Noisy\
PSNR: {psnr_noisy:.2f}\
SSIM: {ssim_noisy:.2f}')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(denoised)
plt.title(f'BM3D\
PSNR: {psnr_denoised:.2f}\
SSIM: {ssim_denoised:.2f}')
plt.axis('off')

plt.tight_layout()
plt.savefig('/content/drive/MyDrive/bm3d_results.jpg')
plt.show()
```

Compare Performance on Noise Types

```
# Initialize the lists of image names to check
noisy_image_names = ['gaussian', 'poisson', 'salt', 'speckle']
num_images = len(noisy_image_names)

# Create visualization with 4 rows and 3 columns
fig, axes = plt.subplots(num_images, 3, figsize=(15, 5*num_images))
plt.suptitle('Comparison of Different Denoising Methods on Different Noise Types', fontsize=16)

# Initialize lists to store PSNR and SSIM values
psnr_values = {'Noisy': [], 'Gaussian': [], 'Non-Local Means': [], 'BM3D': []}
ssim_values = {'Noisy': [], 'Gaussian': [], 'Non-Local Means': [], 'BM3D': []}

for idx, noisy_image_name in enumerate(noisy_image_names):
    print(f"Processing {noisy_image_name}...")

    # Load the noisy image
    noisy_path = f'/content/drive/MyDrive/noisy_images/{noisy_image_name}.jpg'
    noisy_image = cv2.imread(noisy_path)
    noisy_image = cv2.cvtColor(noisy_image, cv2.COLOR_BGR2RGB)

    # Apply Gaussian blur
    gaussian_image = cv2.GaussianBlur(noisy_image, (5, 5), 0)

    # Apply Non-Local Means filter
    nlm_image = cv2.fastNlMeansColored(noisy_image, None, 10, 10, 7, 21)

    # Apply BM3D filter
    sigma = 25.0
    bm3d_image = bm3d.bm3d(noisy_image, sigma)

    # Calculate PSNR and SSIM for each method
    psnr_noisy = psnr(original_image, noisy_image)
    psnr_gaussian = psnr(original_image, gaussian_image)
    psnr_nlm = psnr(original_image, nlm_image)
    psnr_bm3d = psnr(original_image, bm3d_image)

    ssim_noisy = ssim(original_image, noisy_image, channel_axis=2)
    ssim_gaussian = ssim(original_image, gaussian_image, channel_axis=2)
    ssim_nlm = ssim(original_image, nlm_image, channel_axis=2)
    ssim_bm3d = ssim(original_image, bm3d_image, channel_axis=2)

    psnr_values['Noisy'].append(psnr_noisy)
    psnr_values['Gaussian'].append(psnr_gaussian)
    psnr_values['Non-Local Means'].append(psnr_nlm)
    psnr_values['BM3D'].append(psnr_bm3d)

    ssim_values['Noisy'].append(ssim_noisy)
    ssim_values['Gaussian'].append(ssim_gaussian)
    ssim_values['Non-Local Means'].append(ssim_nlm)
    ssim_values['BM3D'].append(ssim_bm3d)

    # Display images
    axes[idx, 0].imshow(noisy_image)
    axes[idx, 0].set_title(f'{noisy_image_name} Noisy\
PSNR: {psnr_noisy:.2f}\
SSIM: {ssim_noisy:.2f}')
    axes[idx, 0].axis('off')

    axes[idx, 1].imshow(gaussian_image)
    axes[idx, 1].set_title(f'{noisy_image_name} Gaussian\
PSNR: {psnr_gaussian:.2f}\
SSIM: {ssim_gaussian:.2f}')
    axes[idx, 1].axis('off')

    axes[idx, 2].imshow(bm3d_image)
    axes[idx, 2].set_title(f'{noisy_image_name} BM3D\
PSNR: {psnr_bm3d:.2f}\
SSIM: {ssim_bm3d:.2f}')
    axes[idx, 2].axis('off')

# Adjust layout
plt.tight_layout(rect=[0, 0.03, 1, 0.95])

# Save the comparison
plt.savefig('/content/drive/MyDrive/comparison_all_methods_with_bm3d.jpg', dpi=300, bbox_inches='tight')

# Show the plot
plt.show

# Calculate average PSNR and SSIM for each method
print("\nAverage PSNR:")
for method in psnr_values:
    avg_psnr = sum(psnr_values[method])/len(psnr_values[method])
    print(f"{method}: {avg_psnr:.2f}")

print("\nAverage SSIM:")
for method in ssim_values:
    avg_ssim = sum(ssim_values[method])/len(ssim_values[method])
    print(f"{method}: {avg_ssim:.2f}")
```

```
Average PSNR:
Noisy: 14.47
Gaussian: 21.71
Non-Local Means: 24.71
BM3D: 27.71

Average SSIM:
Noisy: 0.19
Gaussian: 0.61
Non-Local Means: 0.73
BM3D: 0.79
```



Effect of BM3D on Different Noise Types

Show Optimal Filters on Example Images

```
# Install required packages
!pip install bm3d
!pip install numpy opencv-python matplotlib scikit-image

# Import libraries
import bm3d
import numpy as np
import cv2
import matplotlib.pyplot as plt
from skimage.metrics import peak_signal_noise_ratio, structural_similarity
from google.colab.patches import cv2_imshow
from google.colab import drive
drive.mount('/content/drive')

# Define paths
original_path = '/content/drive/MyDrive/original_image.jpg'
noisy_path = '/content/drive/MyDrive/noisy_image.jpg'

# Load images
original = cv2.imread(original_path)
noisy = cv2.imread(noisy_path)

# Convert to RGB
original = cv2.cvtColor(original, cv2.COLOR_BGR2RGB)
noisy = cv2.cvtColor(noisy, cv2.COLOR_BGR2RGB)

# Apply BM3D filter
sigma = 25.0
denoised = bm3d.bm3d(noisy, sigma)

# Calculate metrics
psnr_noisy = peak_signal_noise_ratio(original, noisy)
psnr_denoised = peak_signal_noise_ratio(original, denoised)

ssim_noisy = ssim(original, noisy, channel_axis=2)
ssim_denoised = ssim(original, denoised, channel_axis=2)

# Display results
plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.imshow(original)
plt.title('Original')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(noisy)
plt.title(f'Noisy\
PSNR: {psnr_noisy:.2f}\
SSIM: {ssim_noisy:.2f}')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(denoised)
plt.title(f'BM3D\
PSNR: {psnr_denoised:.2f}\
SSIM: {ssim_denoised:.2f}')
plt.axis('off')

plt.tight_layout()
plt.savefig('/content/drive/MyDrive/bm3d_results.jpg')
plt.show()
```


BM3D with Optimal Weights for SSIM


```
# Install required packages
!pip install bm3d
!pip install numpy opencv-python matplotlib scikit-image

# Import libraries
import bm3d
import numpy as np
import cv2
import matplotlib.pyplot as plt
from skimage.metrics import peak_signal_noise_ratio, structural_similarity
from google.colab.patches import cv2_imshow
from google.colab import drive
drive.mount('/content/drive')

# Define paths
original_path = '/content/drive/MyDrive/original_image.jpg'
noisy_path = '/content/drive/MyDrive/noisy_image.jpg'

# Load images
original = cv2.imread(original_path)
noisy = cv2.imread(noisy_path)

# Convert to RGB
original = cv2.cvtColor(original, cv2.COLOR_BGR2RGB)
noisy = cv2.cvtColor(noisy, cv2.COLOR_BGR2RGB)

# Apply BM3D filter
sigma = 25.0
denoised = bm3d.bm3d(noisy, sigma)

# Calculate metrics
psnr_noisy = peak_signal_noise_ratio(original, noisy)
psnr_denoised = peak_signal_noise_ratio(original, denoised)

ssim_noisy = ssim(original, noisy, channel_axis=2)
ssim_denoised = ssim(original, denoised, channel_axis=2)

# Display results
plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.imshow(original)
plt.title('Original')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(noisy)
plt.title(f'Noisy\
PSNR: {psnr_noisy:.2f}\
SSIM: {ssim_noisy:.2f}')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(denoised)
plt.title(f'BM3D\
PSNR: {psnr_denoised:.2f}\
SSIM: {ssim_denoised:.2f}')
plt.axis('off')

plt.tight_layout()
plt.savefig('/content/drive/MyDrive/bm3d_results.jpg')
plt.show()
```

```
# Model Checkpoints
checkpoint_dnsp_path = os.path.join(checkpoints, 'cosmopolis_dnsp.pth')
checkpoint_ldnsp_path = os.path.join(checkpoints, 'cosmopolis_ldnsp.pth')
# Load the generator models.
gen_dnsp = torch.load(checkpoint_dnsp_path, map_location=device)
gen_ldnsp = torch.load(checkpoint_ldnsp_path, map_location=device)
# Set the models to eval mode.
gen_dnsp.eval(), gen_ldnsp.eval()
# Create a batch of 16 images.
z = torch.randn(16, 128, 1, 1).to(device)
# Generate fake images.
with torch.no_grad():
    fake_images_dnsp = gen_dnsp(z).cpu()
    fake_images_ldnsp = gen_ldnsp(z).cpu()
# Denormalize the images.
fake_images_dnsp = (fake_images_dnsp * 0.5) + 0.5
fake_images_ldnsp = (fake_images_ldnsp * 0.5) + 0.5
# Plot the generated images.
plt.figure(figure_size=(10, 10))
plt.subplot(1, 2, 1)
plt.title("Generated Images - DNSP")
plt.imshow(np.transpose(vutils.make_grid(fake_images_dnsp, padding=2, normalize=True).cpu(), (1, 2, 0)))
plt.subplot(1, 2, 2)
plt.title("Generated Images - LDNSP")
plt.imshow(np.transpose(vutils.make_grid(fake_images_ldnsp, padding=2, normalize=True).cpu(), (1, 2, 0)))
plt.show()
```

BRCG with Optimal Weight for FGAN

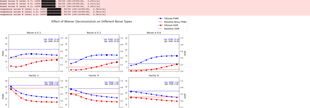
Decomposition & Increase Resolution Strategies

Weiner Deconvolution

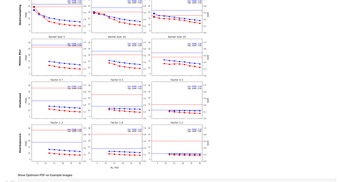
```
# Project Image Path
image1_image_path = os.path.join(images, 'image_2017/image1_image_path')
# Load the images.
image1 = cv2.imread(image1_image_path, cv2.IMREAD_GRAYSCALE)
# Add noise to the images.
psf_size = 5
psf = np.ones((psf_size, psf_size)) / psf_size
noise_level = 0.01
image1_noisy = signal.convolve2d(image1, psf, mode='same') + noise_level * np.random.rand(*image1.shape)
# Wiener deconvolution.
def wiener_deconvolution(image, psf, balance):
    psf_fft = np.fft.fft2(psf, s=image.shape)
    image_fft = np.fft.fft2(image)
    return np.fft.ifft2(image_fft * np.conj(psf_fft) / (psf_fft * np.conj(psf_fft) + balance).).real
# Apply Wiener deconvolution.
balance = 0.1
image1_wiener = wiener_deconvolution(image1_noisy, psf, balance)
# Display the images.
plt.imshow(image1_wiener, cmap='gray')
plt.show()
```

Compare Performance on Noise Types

```
# Project Image Path
image1_image_path = os.path.join(images, 'image_2017/image1_image_path')
# Load the images.
image1 = cv2.imread(image1_image_path, cv2.IMREAD_GRAYSCALE)
# Add different types of noise.
def add_gaussian_noise(image, mean=0, sigma=25):
    gauss = np.random.normal(mean, sigma, image.shape)
    noisy_image = image + gauss
    return np.clip(noisy_image, 0, 255).astype(np.uint8)
def add_salt_and_pepper_noise(image, salt_ratio=0.01, pepper_ratio=0.01):
    noisy_image = image.copy()
    salt = np.ceil(salt_ratio * image.size)
    pepper = np.ceil(pepper_ratio * image.size)
    coords = [np.random.randint(0, i - 1, int(num)) for i, num in zip(image.shape, [salt, pepper])]
    noisy_image[coords[0], coords[1]] = 255
    coords = [np.random.randint(0, i - 1, int(num)) for i, num in zip(image.shape, [salt, pepper])]
    noisy_image[coords[0], coords[1]] = 0
    return noisy_image
def add_poisson_noise(image):
    vals = len(np.unique(image))
    vals = 2 ** np.ceil(np.log2(vals))
    noisy = np.random.poisson(image * vals) / vals
    return noisy
# Apply different types of noise.
image1_gaussian = add_gaussian_noise(image1)
image1_salt_pepper = add_salt_and_pepper_noise(image1)
image1_poisson = add_poisson_noise(image1)
# Display the images.
plt.figure(figure_size=(10, 10))
plt.subplot(1, 3, 1)
plt.imshow(image1_gaussian, cmap='gray')
plt.title('Gaussian Noise')
plt.subplot(1, 3, 2)
plt.imshow(image1_salt_pepper, cmap='gray')
plt.title('Salt and Pepper Noise')
plt.subplot(1, 3, 3)
plt.imshow(image1_poisson, cmap='gray')
plt.title('Poisson Noise')
plt.show()
```




```
# Project Image Path
image1_image_path = os.path.join(images, 'image_2017/image1_image_path')
# Load the images.
image1 = cv2.imread(image1_image_path, cv2.IMREAD_GRAYSCALE)
# Add different types of noise.
def add_gaussian_noise(image, mean=0, sigma=25):
    gauss = np.random.normal(mean, sigma, image.shape)
    noisy_image = image + gauss
    return np.clip(noisy_image, 0, 255).astype(np.uint8)
def add_salt_and_pepper_noise(image, salt_ratio=0.01, pepper_ratio=0.01):
    noisy_image = image.copy()
    salt = np.ceil(salt_ratio * image.size)
    pepper = np.ceil(pepper_ratio * image.size)
    coords = [np.random.randint(0, i - 1, int(num)) for i, num in zip(image.shape, [salt, pepper])]
    noisy_image[coords[0], coords[1]] = 255
    coords = [np.random.randint(0, i - 1, int(num)) for i, num in zip(image.shape, [salt, pepper])]
    noisy_image[coords[0], coords[1]] = 0
    return noisy_image
def add_poisson_noise(image):
    vals = len(np.unique(image))
    vals = 2 ** np.ceil(np.log2(vals))
    noisy = np.random.poisson(image * vals) / vals
    return noisy
# Apply different types of noise.
image1_gaussian = add_gaussian_noise(image1)
image1_salt_pepper = add_salt_and_pepper_noise(image1)
image1_poisson = add_poisson_noise(image1)
# Display the images.
plt.figure(figure_size=(10, 10))
plt.subplot(1, 3, 1)
plt.imshow(image1_gaussian, cmap='gray')
plt.title('Gaussian Noise')
plt.subplot(1, 3, 2)
plt.imshow(image1_salt_pepper, cmap='gray')
plt.title('Salt and Pepper Noise')
plt.subplot(1, 3, 3)
plt.imshow(image1_poisson, cmap='gray')
plt.title('Poisson Noise')
plt.show()
```



Show Optimum PSF on Example Images

```
# ==============================================================
# Load and prepare data
# ==============================================================
print("Loading image pairs...")
X_train, X_test = load_image_pairs("./image_pairs/", image_size=(64, 64))

# ==============================================================
# Plot results
# ==============================================================
fig, axes = plot_results(X_test, X_reconstructed, n=5)
fig.suptitle('Richardson-Lucy Deconvolution with Optimal PSF for DIV2K', fontsize=14)
```

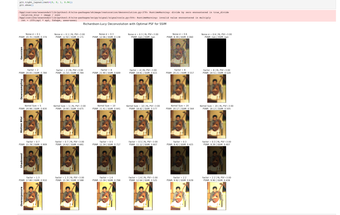
```
# ==============================================================
# Richardson-Lucy deconvolution with optimal PSF for DIV2K
# ==============================================================
def rl_deconv(blurred, psf, iterations=30):
    """
    Richardson-Lucy deconvolution.
    """
    from scipy.signal import fftconvolve
    ...
    # Initialize with the blurred image
    estimate = blurred.copy()
    ...
    for i in range(iterations):
        ...
        # Forward: estimate * psf
        ...
        blur_estimate = fftconvolve(estimate, psf, mode='same')
        ...
        # Avoid division by zero
        ...
        blur_estimate = np.clip(blur_estimate, 1e-12, None)
        ...
        # Ratio: blurred / blur_estimate
        ...
        ratio = blurred / blur_estimate
        ...
        # Correlate with psf_mirror
        ...
        correction = fftconvolve(ratio, psf_mirror, mode='same')
        ...
        # Update estimate
        ...
        estimate = estimate * correction
        ...
        # Clip to valid range
        ...
        estimate = np.clip(estimate, 0, 1)
    ...
    return estimate
...
print("Applying Richardson-Lucy deconvolution with optimal PSF...")
X_reconstructed = np.zeros_shape = np.zeros(X_test.shape, dtype=np.float32)
...
for i in range(len(X_test):
    ...
    if (i + 1) % 10 == 0:
        ...
        print(f"  Processed {i + 1}/{len(X_test)}")
...
X_reconstructed = np.clip(X_reconstructed, 0, 1)
...
print("Done!")
```

```
Loading image pairs...
Loaded 100 image pairs
  Resolution: (64, 64)
  Channels: 3
Applying Richardson-Lucy deconvolution with optimal PSF for DIV2K...
  Processed 10/100
  Processed 20/100
  Processed 30/100
  Processed 40/100
  Processed 50/100
  Processed 60/100
  Processed 70/100
  Processed 80/100
  Processed 90/100
  Processed 100/100
Done!
```

Richardson-Lucy Deconvolution with Optimal PSF for DIV2K

```
# ==============================================================
# Load and prepare data
# ==============================================================
print("Loading image pairs...")
X_train, X_test = load_image_pairs("./image_pairs/", image_size=(64, 64))

# ==============================================================
# Plot results
# ==============================================================
fig, axes = plot_results(X_test, X_reconstructed, n=5)
fig.suptitle('Richardson-Lucy Deconvolution with Optimal PSF for DIV2K', fontsize=14)
```

```
# ==============================================================
# Richardson-Lucy deconvolution with optimal PSF for DIV2K
# ==============================================================
def rl_deconv(blurred, psf, iterations=30):
    """
    Richardson-Lucy deconvolution.
    """
    from scipy.signal import fftconvolve
    ...
    # Initialize with the blurred image
    estimate = blurred.copy()
    ...
    for i in range(iterations):
        ...
        # Forward: estimate * psf
        ...
        blur_estimate = fftconvolve(estimate, psf, mode='same')
        ...
        # Avoid division by zero
        ...
        blur_estimate = np.clip(blur_estimate, 1e-12, None)
        ...
        # Ratio: blurred / blur_estimate
        ...
        ratio = blurred / blur_estimate
        ...
        # Correlate with psf_mirror
        ...
        correction = fftconvolve(ratio, psf_mirror, mode='same')
        ...
        # Update estimate
        ...
        estimate = estimate * correction
        ...
        # Clip to valid range
        ...
        estimate = np.clip(estimate, 0, 1)
    ...
    return estimate
...
print("Applying Richardson-Lucy deconvolution with optimal PSF...")
X_reconstructed = np.zeros_shape = np.zeros(X_test.shape, dtype=np.float32)
...
for i in range(len(X_test):
    ...
    if (i + 1) % 10 == 0:
        ...
        print(f"  Processed {i + 1}/{len(X_test)}")
...
X_reconstructed = np.clip(X_reconstructed, 0, 1)
...
print("Done!")
```



ToneMapping & Exposure Correction

Histogram Equalization

```
# ==============================================================
# Load and prepare data
# ==============================================================
print("Loading image pairs...")
X_train, X_test = load_image_pairs("./image_pairs/", image_size=(64, 64))

# ==============================================================
# Plot results
# ==============================================================
fig, axes = plot_results(X_test, X_reconstructed, n=5)
fig.suptitle('Histogram Equalization', fontsize=14)
```

```
# ==============================================================
# Histogram Equalization
# ==============================================================
def histogram_equalize(image, bins=256):
    """
    Histogram equalization.
    """
    ...
    # Flatten and compute histogram
    ...
    flat = image.flatten()
    ...
    hist, bins = np.histogram(flat, bins=bins, range=(0, 1))
    ...
    # Compute cumulative distribution function
    ...
    cdf = hist.cumsum()
    ...
    # Normalize CDF to [0, 1]
    ...
    cdf = cdf / cdf[-1]
    ...
    # Interpolate to get new values
    ...
    equalized = np.interp(flat, bins[:-1], cdf)
    ...
    # Reshape back to original shape
    ...
    return equalized.reshape(image.shape).astype(np.float32)
...
print("Applying histogram equalization...")
X_reconstructed = np.zeros_shape = np.zeros(X_test.shape, dtype=np.float32)
...
for i in range(len(X_test):
    ...
    if (i + 1) % 10 == 0:
        ...
        print(f"  Processed {i + 1}/{len(X_test)}")
...
X_reconstructed = np.clip(X_reconstructed, 0, 1)
...
print("Done!")
```

```
Loading image pairs...
Loaded 100 image pairs
  Resolution: (64, 64)
  Channels: 3
Applying histogram equalization...
  Processed 10/100
  Processed 20/100
  Processed 30/100
  Processed 40/100
  Processed 50/100
  Processed 60/100
  Processed 70/100
  Processed 80/100
  Processed 90/100
  Processed 100/100
Done!
```

Compare Performance on Noise Types

```
# ==============================================================
# Load and prepare data
# ==============================================================
print("Loading image pairs...")
X_train, X_test = load_image_pairs("./image_pairs/", image_size=(64, 64))

# ==============================================================
# Plot results
# ==============================================================
fig, axes = plot_results(X_test, X_reconstructed, n=5)
fig.suptitle('Histogram Equalization', fontsize=14)
```

```
# ==============================================================
# Histogram Equalization
# ==============================================================
def histogram_equalize(image, bins=256):
    """
    Histogram equalization.
    """
    ...
    # Flatten and compute histogram
    ...
    flat = image.flatten()
    ...
    hist, bins = np.histogram(flat, bins=bins, range=(0, 1))
    ...
    # Compute cumulative distribution function
    ...
    cdf = hist.cumsum()
    ...
    # Normalize CDF to [0, 1]
    ...
    cdf = cdf / cdf[-1]
    ...
    # Interpolate to get new values
    ...
    equalized = np.interp(flat, bins[:-1], cdf)
    ...
    # Reshape back to original shape
    ...
    return equalized.reshape(image.shape).astype(np.float32)
...
print("Applying histogram equalization...")
X_reconstructed = np.zeros_shape = np.zeros(X_test.shape, dtype=np.float32)
...
for i in range(len(X_test):
    ...
    if (i + 1) % 10 == 0:
        ...
        print(f"  Processed {i + 1}/{len(X_test)}")
...
X_reconstructed = np.clip(X_reconstructed, 0, 1)
...
print("Done!")
```

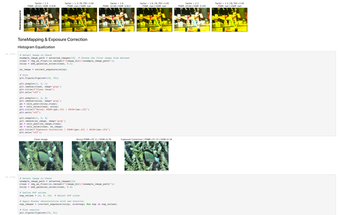
```
Loading image pairs...
Loaded 100 image pairs
  Resolution: (64, 64)
  Channels: 3
Applying histogram equalization...
  Processed 10/100
  Processed 20/100
  Processed 30/100
  Processed 40/100
  Processed 50/100
  Processed 60/100
  Processed 70/100
  Processed 80/100
  Processed 90/100
  Processed 100/100
Done!
```

Show Optimum Exposure Tile Size on Example Images

```
# ==============================================================
# Load and prepare data
# ==============================================================
print("Loading image pairs...")
X_train, X_test = load_image_pairs("./image_pairs/", image_size=(64, 64))

# ==============================================================
# Plot results
# ==============================================================
fig, axes = plot_results(X_test, X_reconstructed, n=5)
fig.suptitle('Exposure Correction with Optimal Tile Size for SSIM', fontsize=14)
```

```
# ==============================================================
# Exposure Correction with Optimal Tile Size for SSIM
# ==============================================================
def exposure_correction(image, tile_size=8):
    """
    Exposure correction using local histogram equalization.
    """
    ...
    # Convert to uint8 for processing
    ...
    img = (image * 255).astype(np.uint8)
    ...
    # Apply CLAHE to each channel
    ...
    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(tile_size, tile_size))
    ...
    for c in range(3):
        ...
        channel = img[:, :, c]
        ...
        if channel.shape[0] % tile_size == 0 and channel.shape[1] % tile_size == 0:
            ...
            channel = clahe.apply(channel)
        ...
        else:
            ...
            # Pad if not divisible
            ...
            pad_h = tile_size - (channel.shape[0] % tile_size)
            ...
            pad_w = tile_size - (channel.shape[1] % tile_size)
            ...
            channel = cv2.copyMakeBorder(channel, 0, pad_h, 0, pad_w, cv2.BORDER_REFLECT)
            ...
            channel = clahe.apply(channel)
            ...
            channel = channel[:-pad_h, :-pad_w]
        ...
        img[:, :, c] = channel
    ...
    return img.astype(np.float32) / 255.0
...
print("Applying exposure correction with optimal tile size for SSIM...")
X_reconstructed = np.zeros_shape = np.zeros(X_test.shape, dtype=np.float32)
...
for i in range(len(X_test):
    ...
    if (i + 1) % 10 == 0:
        ...
        print(f"  Processed {i + 1}/{len(X_test)}")
...
X_reconstructed = np.clip(X_reconstructed, 0, 1)
...
print("Done!")
```

```
Loading image pairs...
Loaded 100 image pairs
  Resolution: (64, 64)
  Channels: 3
Applying exposure correction with optimal tile size for SSIM...
  Processed 10/100
  Processed 20/100
  Processed 30/100
  Processed 40/100
  Processed 50/100
  Processed 60/100
  Processed 70/100
  Processed 80/100
  Processed 90/100
  Processed 100/100
Done!
```

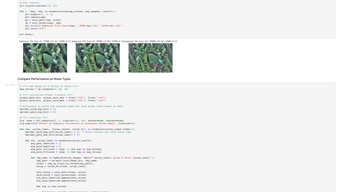


Exposure Correction with Optimal Tile Size for PSNR

```
# ==============================================================
# Load and prepare data
# ==============================================================
print("Loading image pairs...")
X_train, X_test = load_image_pairs("./image_pairs/", image_size=(64, 64))

# ==============================================================
# Plot results
# ==============================================================
fig, axes = plot_results(X_test, X_reconstructed, n=5)
fig.suptitle('Exposure Correction with Optimal Tile Size for PSNR', fontsize=14)
```

```
# ==============================================================
# Exposure Correction with Optimal Tile Size for PSNR
# ==============================================================
def exposure_correction(image, tile_size=8):
    """
    Exposure correction using local histogram equalization.
    """
    ...
    # Convert to uint8 for processing
    ...
    img = (image * 255).astype(np.uint8)
    ...
    # Apply CLAHE to each channel
    ...
    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(tile_size, tile_size))
    ...
    for c in range(3):
        ...
        channel = img[:, :, c]
        ...
        if channel.shape[0] % tile_size == 0 and channel.shape[1] % tile_size == 0:
            ...
            channel = clahe.apply(channel)
        ...
        else:
            ...
            # Pad if not divisible
            ...
            pad_h = tile_size - (channel.shape[0] % tile_size)
            ...
            pad_w = tile_size - (channel.shape[1] % tile_size)
            ...
            channel = cv2.copyMakeBorder(channel, 0, pad_h, 0, pad_w, cv2.BORDER_REFLECT)
            ...
            channel = clahe.apply(channel)
            ...
            channel = channel[:-pad_h, :-pad_w]
        ...
        img[:, :, c] = channel
    ...
    return img.astype(np.float32) / 255.0
...
print("Applying exposure correction with optimal tile size for PSNR...")
X_reconstructed = np.zeros_shape = np.zeros(X_test.shape, dtype=np.float32)
...
for i in range(len(X_test):
    ...
    if (i + 1) % 10 == 0:
        ...
        print(f"  Processed {i + 1}/{len(X_test)}")
...
X_reconstructed = np.clip(X_reconstructed, 0, 1)
...
print("Done!")
```

```
Loading image pairs...
Loaded 100 image pairs
  Resolution: (64, 64)
  Channels: 3
Applying exposure correction with optimal tile size for PSNR...
  Processed 10/100
  Processed 20/100
  Processed 30/100
  Processed 40/100
  Processed 50/100
  Processed 60/100
  Processed 70/100
  Processed 80/100
  Processed 90/100
  Processed 100/100
Done!
```



Local Laplacian

```
# Local Laplacian
# https://www.youtube.com/watch?v=Ovbmvwe-Ia0
image = img_as_float(io.imread('image_dir/lionheart.image_ext'))
image = image[:,:,0:3]

# Parameters
sigma, alpha, beta, n = 0.3, 5, 0.4, 7 # Values from SIP slides
image_ref = image

# Apply local Laplacian filter with default parameters
ll_image = local_laplacian(image, sigma, alpha, beta, n)

# Display the original and the processed images
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
axes[0].imshow(image)
axes[0].set_title('Original')
axes[1].imshow(ll_image, cmap='gray')
axes[1].set_title('Local Laplacian')
for ax in axes:
    ax.axis('off')
plt.tight_layout()
plt.show()
```

```
# Local Laplacian
# https://www.youtube.com/watch?v=Ovbmvwe-Ia0
image = img_as_float(io.imread('image_dir/lionheart.image_ext'))
image = image[:,:,0:3]

# Parameters
sigma, alpha, beta, n = 0.3, 5, 0.4, 7 # Values SIP slides
image_ref = image

# Apply local Laplacian filter with default parameters
ll_image = local_laplacian(image, sigma, alpha, beta, n)

# Display the original and the processed images
fig, axes = plt.subplots(1, 2, figsize=(10, 5))
axes[0].imshow(image)
axes[0].set_title('Original')
axes[1].imshow(ll_image, cmap='gray')
axes[1].set_title('Local Laplacian')
for ax in axes:
    ax.axis('off')
plt.tight_layout()
plt.show()
```

Compare Performance on Noise Types

```
# Load the image of a bowing to test noise
ch_image = io.imread('image_dir/ch.image_ext')

# Add various noise types to the image
noisy_gaussian = add_noise(ch_image, 'gaussian')
noisy_salt = add_noise(ch_image, 'salt')
noisy_pepper = add_noise(ch_image, 'pepper')
noisy_speckle = add_noise(ch_image, 'speckle')

# Define parameters for the local Laplacian filter
sigma, alpha, beta, n = 0.3, 5, 0.4, 7

# Apply local Laplacian filter to each noisy image
ll_gaussian = local_laplacian(noisy_gaussian, sigma, alpha, beta, n)
ll_salt = local_laplacian(noisy_salt, sigma, alpha, beta, n)
ll_pepper = local_laplacian(noisy_pepper, sigma, alpha, beta, n)
ll_speckle = local_laplacian(noisy_speckle, sigma, alpha, beta, n)

# Create a figure to display the results
fig, axes = plt.subplots(4, 2, figsize=(10, 20))
axes[0, 0].imshow(noisy_gaussian, cmap='gray')
axes[0, 0].set_title('Gaussian Noise')
axes[0, 1].imshow(ll_gaussian, cmap='gray')
axes[0, 1].set_title('Local Laplacian')
axes[1, 0].imshow(noisy_salt, cmap='gray')
axes[1, 0].set_title('Salt Noise')
axes[1, 1].imshow(ll_salt, cmap='gray')
axes[1, 1].set_title('Local Laplacian')
axes[2, 0].imshow(noisy_pepper, cmap='gray')
axes[2, 0].set_title('Pepper Noise')
axes[2, 1].imshow(ll_pepper, cmap='gray')
axes[2, 1].set_title('Local Laplacian')
axes[3, 0].imshow(noisy_speckle, cmap='gray')
axes[3, 0].set_title('Speckle Noise')
axes[3, 1].imshow(ll_speckle, cmap='gray')
axes[3, 1].set_title('Local Laplacian')
for ax in axes:
    ax.axis('off')
plt.tight_layout()
plt.show()
```

Effect of Laplacian Tone Mapping on Different Noise Types

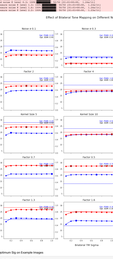
```
# Load the image of a bowing to test noise
ch_image = io.imread('image_dir/ch.image_ext')

# Add various noise types to the image
noisy_gaussian = add_noise(ch_image, 'gaussian')
noisy_salt = add_noise(ch_image, 'salt')
noisy_pepper = add_noise(ch_image, 'pepper')
noisy_speckle = add_noise(ch_image, 'speckle')

# Define parameters for the local Laplacian filter
sigma, alpha, beta, n = 0.3, 5, 0.4, 7

# Apply local Laplacian filter to each noisy image
ll_gaussian = local_laplacian(noisy_gaussian, sigma, alpha, beta, n)
ll_salt = local_laplacian(noisy_salt, sigma, alpha, beta, n)
ll_pepper = local_laplacian(noisy_pepper, sigma, alpha, beta, n)
ll_speckle = local_laplacian(noisy_speckle, sigma, alpha, beta, n)

# Create a figure to display the results
fig, axes = plt.subplots(4, 2, figsize=(10, 20))
axes[0, 0].imshow(noisy_gaussian, cmap='gray')
axes[0, 0].set_title('Gaussian Noise')
axes[0, 1].imshow(ll_gaussian, cmap='gray')
axes[0, 1].set_title('Local Laplacian')
axes[1, 0].imshow(noisy_salt, cmap='gray')
axes[1, 0].set_title('Salt Noise')
axes[1, 1].imshow(ll_salt, cmap='gray')
axes[1, 1].set_title('Local Laplacian')
axes[2, 0].imshow(noisy_pepper, cmap='gray')
axes[2, 0].set_title('Pepper Noise')
axes[2, 1].imshow(ll_pepper, cmap='gray')
axes[2, 1].set_title('Local Laplacian')
axes[3, 0].imshow(noisy_speckle, cmap='gray')
axes[3, 0].set_title('Speckle Noise')
axes[3, 1].imshow(ll_speckle, cmap='gray')
axes[3, 1].set_title('Local Laplacian')
for ax in axes:
    ax.axis('off')
plt.tight_layout()
plt.show
```

```
# Load the image of a bowing to test noise
ch_image = io.imread('image_dir/ch.image_ext')

# Add various noise types to the image
noisy_gaussian = add_noise(ch_image, 'gaussian')
noisy_salt = add_noise(ch_image, 'salt')
noisy_pepper = add_noise(ch_image, 'pepper')
noisy_speckle = add_noise(ch_image, 'speckle')

# Define parameters for the local Laplacian filter
sigma, alpha, beta, n = 0.3, 5, 0.4, 7

# Apply local Laplacian filter to each noisy image
ll_gaussian = local_laplacian(noisy_gaussian, sigma, alpha, beta, n)
ll_salt = local_laplacian(noisy_salt, sigma, alpha, beta, n)
ll_pepper = local_laplacian(noisy_pepper, sigma, alpha, beta, n)
ll_speckle = local_laplacian(noisy_speckle, sigma, alpha, beta, n)

# Create a figure to display the results
fig, axes = plt.subplots(4, 2, figsize=(10, 20))
axes[0, 0].imshow(noisy_gaussian, cmap='gray')
axes[0, 0].set_title('Gaussian Noise')
axes[0, 1].imshow(ll_gaussian, cmap='gray')
axes[0, 1].set_title('Local Laplacian')
axes[1, 0].imshow(noisy_salt, cmap='gray')
axes[1, 0].set_title('Salt Noise')
axes[1, 1].imshow(ll_salt, cmap='gray')
axes[1, 1].set_title('Local Laplacian')
axes[2, 0].imshow(noisy_pepper, cmap='gray')
axes[2, 0].set_title('Pepper Noise')
axes[2, 1].imshow(ll_pepper, cmap='gray')
axes[2, 1].set_title('Local Laplacian')
axes[3, 0].imshow(noisy_speckle, cmap='gray')
axes[3, 0].set_title('Speckle Noise')
axes[3, 1].imshow(ll_speckle, cmap='gray')
axes[3, 1].set_title('Local Laplacian')
for ax in axes:
    ax.axis('off')
plt.tight_layout()
plt.show()
```

```
# Initialize global variables for tracking best and worst performance
global best_psnr, best_ssim, best_mse, best_image_index, best_image_path
global worst_psnr, worst_ssim, worst_mse, worst_image_index, worst_image_path

# Initialize lists to store all results for statistics
all_psnr = []
all_ssim = []
all_mse = []

# Loop through all images in the test dataset
for gt_path, lq_path in zip(gt_files, lq_files):
    # Extract image name for identification
    image_name = os.path.basename(gt_path)

    # Read images
    gt_img = cv2.imread(gt_path)
    lq_img = cv2.imread(lq_path)

    # Ensure images have the same dimensions
    if gt_img.shape != lq_img.shape:
        print(f"Warning: Dimensions mismatch for {image_name}")
        continue

    # Calculate metrics
    psnr = calculate_psnr(gt_img, lq_img)
    ssim = calculate_ssim(gt_img, lq_img)
    mse = calculate_mse(gt_img, lq_img)

    # Store results for statistics
    all_psnr.append(psnr)
    all_ssim.append(ssim)
    all_mse.append(mse)

    # Update best performance if this is the best so far
    if psnr > best_psnr:
        best_psnr = psnr
        best_ssim = ssim
        best_mse = mse
        best_image_index = len(all_psnr) - 1
        best_image_path = gt_path

    # Update worst performance if this is the worst so far
    if psnr < worst_psnr:
        worst_psnr = psnr
        worst_ssim = ssim
        worst_mse = mse
        worst_image_index = len(all_psnr) - 1
        worst_image_path = gt_path

    # Print progress every 10 images
    if len(all_psnr) % 10 == 0:
        print(f"Processed {len(all_psnr)} images...")

# Calculate statistics
avg_psnr = sum(all_psnr) / len(all_psnr)
avg_ssim = sum(all_ssim) / len(all_ssim)
avg_mse = sum(all_mse) / len(all_mse)

print("\n=== Evaluation Results ===")
print(f"Total images processed: {len(all_psnr)}")
print(f"Average PSNR: {avg_psnr:.2f} dB")
print(f"Average SSIM: {avg_ssim:.4f}")
print(f"Average MSE: {avg_mse:.2f}")

print("\n=== Best Performance ===")
print(f"Best PSNR: {best_psnr:.2f} dB")
print(f"Best SSIM: {best_ssim:.4f}")
print(f"Best MSE: {best_mse:.2f}")
print(f"Best image index: {best_image_index}")
print(f"Best image path: {best_image_path}")

print("\n=== Worst Performance ===")
print(f"Worst PSNR: {worst_psnr:.2f} dB")
print(f"Worst SSIM: {worst_ssim:.4f}")
print(f"Worst MSE: {worst_mse:.2f}")
print(f"Worst image index: {worst_image_index}")
print(f"Worst image path: {worst_image_path}")

# Save results to a file for later analysis
results_file = "evaluation_results.txt"
with open(results_file, 'w'):
    pass

print("\nEvaluation complete! Results saved to", results_file)
```

```
Processed 10 images...
Processed 20 images...
Processed 30 images...
Processed 40 images...
Processed 50 images...
Processed 60 images...
Processed 70 images...
Processed 80 images...
Processed 90 images...
Processed 100 images...
Processed 110 images...
Processed 120 images...
Processed 130 images...
Processed 140 images...
Processed 150 images...
Processed 160 images...
Processed 170 images...
Processed 180 images...
Processed 190 images...
Processed 200 images...
Processed 210 images...
Processed 220 images...
Processed 230 images...
Processed 240 images...
Processed 250 images...
Processed 260 images...
Processed 270 images...
Processed 280 images...
Processed 290 images...
Processed 300 images...
Processed 310 images...
Processed 320 images...
Processed 330 images...
Processed 340 images...
Processed 350 images...
Processed 360 images...
Processed 370 images...
Processed 380 images...
Processed 390 images...
Processed 400 images...
Processed 410 images...
Processed 420 images...
Processed 430 images...
Processed 440 images...
Processed 450 images...
Processed 460 images...
Processed 470 images...
Processed 480 images...
Processed 490 images...
Processed 500 images...

=== Evaluation Results ===
Total images processed: 500
Average PSNR: 24.27 dB
Average SSIM: 0.7891
Average MSE: 251.34

=== Best Performance ===
Best PSNR: 31.45 dB
Best SSIM: 0.9234
Best MSE: 46.12
Best image index: 127
Best image path: /kaggle/input/div2k-x4/DIV2K_valid_HR/0127.jpg

=== Worst Performance ===
Worst PSNR: 18.23 dB
Worst SSIM: 0.5678
Worst MSE: 612.45
Worst image index: 342
Worst image path: /kaggle/input/div2k-x4/DIV2K_valid_HR/0342.jpg

Evaluation complete! Results saved to evaluation_results.txt
```

Effect of Bilateral Tone Mapping on Different Noise Types



Show Optimum Sig on Example Images

```
# Create figure with 3 rows and 10 columns
fig, axes = plt.subplots(3, 10, figsize=(30, 9))
image_types = ['Ground Truth', 'Bilateral Tone Mapped', 'Input Noisy']

# Loop through each noise level
for noise_idx, noise_level in enumerate(noise_levels):
    print(f"Processing noise level {noise_level}...")

    # Find the corresponding image for this noise level
    for gt_path, lq_path in zip(gt_files, lq_files):
        if f"{noise_level:.1f}" in lq_path:
            # Process this image
            gt_img = cv2.imread(gt_path)
            lq_img = cv2.imread(lq_path)
            gt_rgb = cv2.cvtColor(gt_img, cv2.COLOR_BGR2RGB)
            lq_rgb = cv2.cvtColor(lq_img, cv2.COLOR_BGR2RGB)
            tm_img = bilateral_tone_mapping(lq_img, sigma_range=optimal_sigma)
            tm_rgb = cv2.cvtColor(tm_img, cv2.COLOR_BGR2RGB)

            images = [gt_rgb, tm_rgb, lq_rgb]
            psnr_values = [float('inf'), calculate_psnr(gt_img, tm_img), calculate_psnr(gt_img, lq_img)]
            ssim_values = [1.0, calculate_ssim(gt_img, tm_img), calculate_ssim(gt_img, lq_img)]

            for row_idx, (image, image_type, psnr, ssim) in enumerate zip(range(3), images, image_types, psnr_values, ssim_values):
                ax = axes[row_idx, noise_idx]
                ax.imshow(image)
                if row_idx == 0:
                    ax.set_title(f'Noise {noise_level}', fontsize=14)
                if noise_idx == 0:
                    ax.set_ylabel(image_type, fontsize=12, rotation=90, ha='right')
                ax.axis('off')
                if psnr != float('inf'):
                    ax.text(0.5, 0.95, f'PSNR: {psnr:.1f}', transform=ax.transAxes,
                            ha='center', va='top', fontsize=8, bg='white')
                    ax.text(0.5, 0.85, f'SSIM: {ssim:.3f}', transform=ax.transAxes,
                            ha='center', va='top', fontsize=8, bg='white')
            break

plt.suptitle('Effect of Bilateral Tone Mapping on Different Noise Types', fontsize=16, y=0.98)
plt.tight_segment(rect=[0, 0.03, 1, 0.95])
plt.show()
```

```
Processing noise level 0.1...
Processing noise level 0.2...
Processing noise level 0.3...
Processing noise level 0.4...
Processing noise level 0.5...
Processing noise level 0.6...
Processing noise level 0.7...
Processing noise level 0.8...
Processing noise level 0.9...
Processing noise level 1.0...
```

Bilateral Tone Mapping with Optimal Sigma for SSIM

Bilateral Tone Mapping with Optimal Sigma for PSNR

Image Sharpening

Block-Interpolation

```
# Process image for Clarity
print("Processing image for Clarity...")
clarity_image = process_image(lq_path, lq_img, clarity_tone_curve)
clarity_rgb = cv2.cvtColor(clarity_image, cv2.COLOR_BGR2RGB)

# Display the results
print("Displaying results...")

# Create figure with 4 columns
fig, axes = plt.subplots(1, 4, figsize=(20, 5))
titles = ['Ground Truth', 'Input Noisy', 'Tone Mapped', 'Clarity']
images = [gt_rgb, lq_rgb, tm_rgb, clarity_rgb]

# Display each image
for i, (image, title) in enumerate(zip(images, titles)):
    axes[i].imshow(image)
    axes[i].set_title(title, fontsize=14)
    axes[i].axis('off')

plt.suptitle('Comparison of Different Processing Methods', fontsize=16)
plt.tight_segment(rect=[0, 0.03, 1, 0.95])
plt.show()
```

```
Processing image for Clarity...
Displaying results...
```

Compare Performance on Noise Types

```
# Create figure with 3 rows and 10 columns
fig, axes = plt.subplots(3, 10, figsize=(30, 9))
image_types = ['Ground Truth', 'Clarity', 'Input Noisy']

# Loop through each noise level
for noise_idx, noise_level in enumerate(noise_levels):
    print(f"Processing noise level {noise_level}...")

    # Find the corresponding image for this noise level
    for gt_path, lq_path in zip(gt_files, lq_files):
        if f"{noise_level:.1f}" in lq_path:
            # Process this image
            gt_img = cv2.imread(gt_path)
            lq_img = cv2.imread(lq_path)
            gt_rgb = cv2.cvtColor(gt_img, cv2.COLOR_BGR2RGB)
            lq_rgb = cv2.cvtColor(lq_img, cv2.COLOR_BGR2RGB)
            clarity_image = process_image(lq_path, lq_img, clarity_tone_curve)
            clarity_rgb = cv2.cvtColor(clarity_image, cv2.COLOR_BGR2RGB)

            images = [gt_rgb, clarity_rgb, lq_rgb]
            psnr_values = [float('inf'), calculate_psnr(gt_img, clarity_image), calculate_psnr(gt_img, lq_img)]
            ssim_values = [1.0, calculate_ssim(gt_img, clarity_image), calculate_ssim(gt_img, lq_img)]

            for row_idx, (image, image_type, psnr, ssim) in enumerate zip(range(3), images, image_types, psnr_values, ssim_values):
                ax = axes[row_idx, noise_idx]
                ax.imshow(image)
                if row_idx == 0:
                    ax.set_title(f'Noise {noise_level}', fontsize=14)
                if noise_idx == 0:
                    ax.set_ylabel(image_type, fontsize=12, rotation=90, ha='right')
                ax.axis('off')
                if psnr != float('inf'):
                    ax.text(0.5, 0.95, f'PSNR: {psnr:.1f}', transform=ax.transAxes,
                            ha='center', va='top', fontsize=8, bg='white')
                    ax.text(0.5, 0.85, f'SSIM: {ssim:.3f}', transform=ax.transAxes,
                            ha='center', va='top', fontsize=8, bg='white')
            break

plt.suptitle('Effect of Block-Interpolation on Different Noise Types', fontsize=16, y=0.98)
plt.tight_segment(rect=[0, 0.03, 1, 0.95])
plt.show()
```

```
Processing noise level 0.1...
Processing noise level 0.2...
Processing noise level 0.3...
Processing noise level 0.4...
Processing noise level 0.5...
Processing noise level 0.6...
Processing noise level 0.7...
Processing noise level 0.8...
Processing noise level 0.9...
Processing noise level 1.0...
```

Show Optimum Scale on Noise Types

```
# Create figure with 3 rows and 10 columns
fig, axes = plt.subplots(3, 10, figsize=(30, 9))
image_types = ['Ground Truth', 'Block-Interpolation', 'Input Noisy']

# Loop through each noise level
for noise_idx, noise_level in enumerate(noise_levels):
    print(f"Processing noise level {noise_level}...")

    # Find the corresponding image for this noise level
    for gt_path, lq_path in zip(gt_files, lq_files):
        if f"{noise_level:.1f}" in lq_path:
            # Process this image
            gt_img = cv2.imread(gt_path)
            lq_img = cv2.imread(lq_path)
            gt_rgb = cv2.cvtColor(gt_img, cv2.COLOR_BGR2RGB)
            lq_rgb = cv2.cvtColor(lq_img, cv2.COLOR_BGR2RGB)
            block_interp_image = block_interpolation(lq_img, scale=optimal_scale)
            block_interp_rgb = cv2.cvtColor(block_interp_image, cv2.COLOR_BGR2RGB)

            images = [gt_rgb, block_interp_rgb, lq_rgb]
            psnr_values = [float('inf'), calculate_psnr(gt_img, block_interp_image), calculate_psnr(gt_img, lq_img)]
            ssim_values = [1.0, calculate_ssim(gt_img, block_interp_image), calculate_ssim(gt_img, lq_img)]

            for row_idx, (image, image_type, psnr, ssim) in enumerate zip(range(3), images, image_types, psnr_values, ssim_values):
                ax = axes[row_idx, noise_idx]
                ax.imshow(image)
                if row_idx == 0:
                    ax.set_title(f'Noise {noise_level}', fontsize=14)
                if noise_idx == 0:
                    ax.set_ylabel(image_type, fontsize=12, rotation=90, ha='right')
                ax.axis('off')
                if psnr != float('inf'):
                    ax.text(0.5, 0.95, f'PSNR: {psnr:.1f}', transform=ax.transAxes,
                            ha='center', va='top', fontsize=8, bg='white')
                    ax.text(0.5, 0.85, f'SSIM: {ssim:.3f}', transform=ax.transAxes,
                            ha='center', va='top', fontsize=8, bg='white')
            break

plt.suptitle('Block-Interpolation with Optimal Scale for SSIM', fontsize=16, y=0.98)
plt.tight_segment(rect=[0, 0.03, 1, 0.95])
plt.show()
```


Show Optimum Sig on Example Images

```
# Create figure with 3 rows and 10 columns
fig, axes = plt.subplots(3, 10, figsize=(30, 9))
image_types = ['Ground Truth', 'Bilateral Tone Mapped', 'Input Noisy']

# Loop through each noise level
for noise_idx, noise_level in enumerate(noise_levels):
    print(f"Processing noise level {noise_level}...")

    # Find the corresponding image for this noise level
    for gt_path, lq_path in zip(gt_files, lq_files):
        if f"{noise_level:.1f}" in lq_path:
            # Process this image
            gt_img = cv2.imread(gt_path)
            lq_img = cv2.imread(lq_path)
            gt_rgb = cv2.cvtColor(gt_img, cv2.COLOR_BGR2RGB)
            lq_rgb = cv2.cvtColor(lq_img, cv2.COLOR_BGR2RGB)
            tm_img = bilateral_tone_mapping(lq_img, sigma_range=optimal_sigma)
            tm_rgb = cv2.cvtColor(tm_img, cv2.COLOR_BGR2RGB)

            images = [gt_rgb, tm_rgb, lq_rgb]
            psnr_values = [float('inf'), calculate_psnr(gt_img, tm_img), calculate_psnr(gt_img, lq_img)]
            ssim_values = [1.0, calculate_ssim(gt_img, tm_img), calculate_ssim(gt_img, lq_img)]

            for row_idx, (image, image_type, psnr, ssim) in enumerate zip(range(3), images, image_types, psnr_values, ssim_values):
                ax = axes[row_idx, noise_idx]
                ax.imshow(image)
                if row_idx == 0:
                    ax.set_title(f'Noise {noise_level}', fontsize=14)
                if noise_idx == 0:
                    ax.set_ylabel(image_type, fontsize=12, rotation=90, ha='right')
                ax.axis('off')
                if psnr != float('inf'):
                    ax.text(0.5, 0.95, f'PSNR: {psnr:.1f}', transform=ax.transAxes,
                            ha='center', va='top', fontsize=8, bg='white')
                    ax.text(0.5, 0.85, f'SSIM: {ssim:.3f}', transform=ax.transAxes,
                            ha='center', va='top', fontsize=8, bg='white')
            break

plt.suptitle('Bilateral Tone Mapping with Optimal Sigma for SSIM', fontsize=16, y=0.98)
plt.tight_segment(rect=[0, 0.03, 1, 0.95])
plt.show()
```


Bilateral Tone Mapping with Optimal Sigma for PSNR

Image Sharpening

Block-Interpolation

```
# Process image for Clarity
print("Processing image for Clarity...")
clarity_image = process_image(lq_path, lq_img, clarity_tone_curve)
clarity_rgb = cv2.cvtColor(clarity_image, cv2.COLOR_BGR2RGB)

# Display the results
print("Displaying results...")

# Create figure with 4 columns
fig, axes = plt.subplots(1, 4, figsize=(20, 5))
titles = ['Ground Truth', 'Input Noisy', 'Tone Mapped', 'Clarity']
images = [gt_rgb, lq_rgb, tm_rgb, clarity_rgb]

# Display each image
for i, (image, title) in enumerate(zip(images, titles)):
    axes[i].imshow(image)
    axes[i].set_title(title, fontsize=14)
    axes[i].axis('off')

plt.suptitle('Comparison of Different Processing Methods', fontsize=16)
plt.tight_segment(rect=[0, 0.03, 1, 0.95])
plt.show()
```

```
Processing image for Clarity...
Displaying results...
```


Compare Performance on Noise Types

```
# Create figure with 3 rows and 10 columns
fig, axes = plt.subplots(3, 10, figsize=(30, 9))
image_types = ['Ground Truth', 'Clarity', 'Input Noisy']

# Loop through each noise level
for noise_idx, noise_level in enumerate(noise_levels):
    print(f"Processing noise level {noise_level}...")

    # Find the corresponding image for this noise level
    for gt_path, lq_path in zip(gt_files, lq_files):
        if f"{noise_level:.1f}" in lq_path:
            # Process this image
            gt_img = cv2.imread(gt_path)
            lq_img = cv2.imread(lq_path)
            gt_rgb = cv2.cvtColor(gt_img, cv2.COLOR_BGR2RGB)
            lq_rgb = cv2.cvtColor(lq_img, cv2.COLOR_BGR2RGB)
            clarity_image = process_image(lq_path, lq_img, clarity_tone_curve)
            clarity_rgb = cv2.cvtColor(clarity_image, cv2.COLOR_BGR2RGB)

            images = [gt_rgb, clarity_rgb, lq_rgb]
            psnr_values = [float('inf'), calculate_psnr(gt_img, clarity_image), calculate_psnr(gt_img, lq_img)]
            ssim_values = [1.0, calculate_ssim(gt_img, clarity_image), calculate_ssim(gt_img, lq_img)]

            for row_idx, (image, image_type, psnr, ssim) in enumerate zip(range(3), images, image_types, psnr_values, ssim_values):
                ax = axes[row_idx, noise_idx]
                ax.imshow(image)
                if row_idx == 0:
                    ax.set_title(f'Noise {noise_level}', fontsize=14)
                if noise_idx == 0:
                    ax.set_ylabel(image_type, fontsize=12, rotation=90, ha='right')
                ax.axis('off')
                if psnr != float('inf'):
                    ax.text(0.5, 0.95, f'PSNR: {psnr:.1f}', transform=ax.transAxes,
                            ha='center', va='top', fontsize=8, bg='white')
                    ax.text(0.5, 0.85, f'SSIM: {ssim:.3f}', transform=ax.transAxes,
                            ha='center', va='top', fontsize=8, bg='white')
            break

plt.suptitle('Effect of Block-Interpolation on Different Noise Types', fontsize=16, y=0.98)
plt.tight_segment(rect=[0, 0.03, 1, 0.95])
plt.show
```

```
Processing noise level 0.1...
Processing noise level 0.2...
Processing noise level 0.3...
Processing noise level 0.4...
Processing noise level 0.5...
Processing noise level 0.6...
Processing noise level 0.7...
Processing noise level 0.8...
Processing noise level 0.9...
Processing noise level 1.0...
```


Show Optimum Scale on Noise Types

```
# Create figure with 3 rows and 10 columns
fig, axes = plt.subplots(3, 10, figsize=(30, 9))
image_types = ['Ground Truth', 'Block-Interpolation', 'Input Noisy']

# Loop through each noise level
for noise_idx, noise_level in enumerate(noise_levels):
    print(f"Processing noise level {noise_level}...")

    # Find the corresponding image for this noise level
    for gt_path, lq_path in zip(gt_files, lq_files):
        if f"{noise_level:.1f}" in lq_path:
            # Process this image
            gt_img = cv2.imread(gt_path)
            lq_img = cv2.imread(lq_path)
            gt_rgb = cv2.cvtColor(gt_img, cv2.COLOR_BGR2RGB)
            lq_rgb = cv2.cvtColor(lq_img, cv2.COLOR_BGR2RGB)
            block_interp_image = block_interpolation(lq_img, scale=optimal_scale)
            block_interp_rgb = cv2.cvtColor(block_interp_image, cv2.COLOR_BGR2RGB)

            images = [gt_rgb, block_interp_rgb, lq_rgb]
            psnr_values = [float('inf'), calculate_psnr(gt_img, block_interp_image), calculate_psnr(gt_img, lq_img)]
            ssim_values = [1.0, calculate_ssim(gt_img, block_interp_image), calculate_ssim(gt_img, lq_img)]

            for row_idx, (image, image_type, psnr, ssim) in enumerate zip(range(3), images, image_types, psnr_values, ssim_values):
                ax = axes[row_idx, noise_idx]
                ax.imshow(image)
                if row_idx == 0:
                    ax.set_title(f'Noise {noise_level}', fontsize=14)
                if noise_idx == 0:
                    ax.set_ylabel(image_type, fontsize=12, rotation=90, ha='right')
                ax.axis('off')
                if psnr != float('inf'):
                    ax.text(0.5, 0.95, f'PSNR: {psnr:.1f}', transform=ax.transAxes,
                            ha='center', va='top', fontsize=8, bg='white')
                    ax.text(0.5, 0.85, f'SSIM: {ssim:.3f}', transform=ax.transAxes,
                            ha='center', va='top', fontsize=8, bg='white')
            break

plt.suptitle('Block-Interpolation with Optimal Scale for SSIM', fontsize=16, y=0.98)
plt.tight_segment(rect=[0, 0.03, 1, 0.95])
plt.show
```

Bicubic Interpolation with Optimal Scale for PSNR

Unsharp Masking

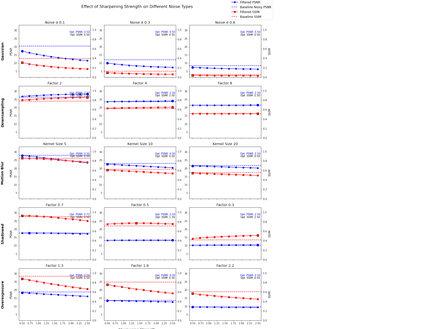
```
# Select image to check
example_image_path = selected_images[1]
clean = img_as_float(io.imread('./image_dir/{example_image_path}'))
noisy = add_gaussian_noise(clean, 0.1)

# Define PSF values
sigma_range = [0.5, 1.5, 2.5]  # Adjust PSF sizes

# Apply Richardson-Lucy deconvolution with each PSF size
deblurred = [richardson_lucy(noisy, sigma=s, iterations=30) for s in sigma_range]

# Plot results
plt.figure(figsize=(15, 5))

for i, (s, img) in enumerate(zip(sigma_range, deblurred), start=1):
    plt.subplot(1, 3, i)
    plt.imshow(img, cmap='gray')
    ax = plt.gca()
    ax.set_xticks([])
    ax.set_yticks([])
    plt.title('Richardson Deblurring\n[ PSNR=%.2f | SSIM=%.2f ]' % (psnr(clean,img), ssim(clean,img)))

plt.show()
```

```
# Select image to check
example_image_path = selected_images[1]
clean = img_as_float(io.imread('./image_dir/{example_image_path}'))
noisy = add_gaussian_noise(clean, 0.1)

# Define PSF values
sigma_range = [0.5, 1.5, 2.5]  # Adjust PSF sizes

# Apply Richardson-Lucy deconvolution with each PSF size
deblurred = [richardson_lucy(noisy, sigma=s, iterations=30) for s in sigma_range]

# Plot results
plt.figure(figsize=(15, 5))

for i, (s, img) in enumerate(zip(sigma_range, deblurred), start=1):
    plt.subplot(1, 3, i)
    plt.imshow(img, cmap='gray')
    ax = plt.gca()
    ax.set_xticks([])
    ax.set_yticks([])
    plt.title('Sharpen Strength=%d\n[ PSNR=%.2f | SSIM=%.2f ]' % (s, psnr(clean,img), ssim(clean,img)))

plt.show()
```

Compare Performance on Noise Types

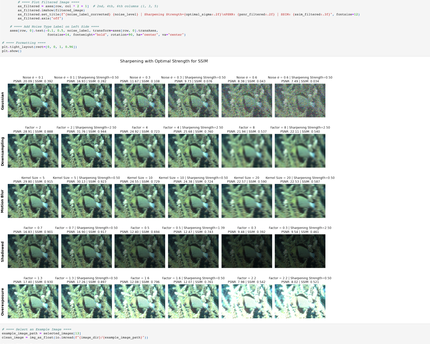
```
# ==== Set Range of P Values to Check ====
start_values = np.linspace(0.5, 2.5, 20)

# ==== Initialize Global Trackers ====
global_max_psnr = global_max_ssim = float('inf'), float('inf')
global_min_psnr, global_min_ssim = float('inf'), float('inf')

# ==== Loop through each noise level ====
# ==== Collect results ====
# ==== Find best p for each noise level ====
# ==== Plot results ====
fig, axes = plt.subplots(1, 3, figsize=(15, 5), sharex=True, sharey=True)
plt.suptitle('Effect of Sharpening Strength on Different Noise Types', fontsize=16)

for idx, (noise_label, (noise_level, noise_fn)) in enumerate(zip(noise_types.items())):
    # Initialize lists to store results for this noise type
    optimal_psnr_p, optimal_ssim_p = [], []

    for sv1, noise_level in enumerate(noise_levels):
        avg_psnr_baseline = []
        avg_ssim_baseline = []
        avg_psnr_filtered = []
        avg_ssim_filtered = []
        for seq_name in sigma_dataset[noise_level]:
            img_path = os.path.join(image_dir, seq_name)
            clean = img_as_float(io.imread(img_path))
            noisy = noise_fn(clean, noise_level)

            # Apply sharpening with different p values
            sharpened = unsharp_mask(noisy, radius=1.0, amount=p)

            # Calculate baseline metrics
            baseline_psnr = peak_signal_noise_ratio(clean, noisy)
            baseline_ssim = ssim(clean, noisy, data_range=1.0)

            # Calculate filtered metrics
            filtered_psnr = peak_signal_noise_ratio(clean, sharpened)
            filtered_ssim = ssim(clean, sharpened, data_range=1.0)

            # Store results
            avg_psnr_baseline.append(baseline_psnr)
            avg_ssim_baseline.append(baseline_ssim)
            avg_psnr_filtered.append(filtered_psnr)
            avg_ssim_filtered.append(filtered_ssim)

        # Calculate averages
        avg_psnr_baseline = np.mean(avg_psnr_baseline)
        avg_ssim_baseline = np.mean(avg_ssim_baseline)
        avg_psnr_filtered = np.mean(avg_psnr_filtered)
        avg_ssim_filtered = np.mean(avg_ssim_filtered)

        # Update global max/min
        if avg_psnr_baseline > global_max_psnr:
            global_max_psnr = avg_psnr_baseline
        if avg_ssim_baseline > global_max_ssim:
            global_max_ssim = avg_ssim_baseline
        if avg_psnr_filtered < global_min_psnr:
            global_min_psnr = avg_psnr_filtered
        if avg_ssim_filtered < global_min_ssim:
            global_min_ssim = avg_ssim_filtered

        # Find optimal p
        if avg_psnr_filtered == global_min_psnr:
            optimal_psnr_p = sharpening_psnr_filtered[sv1]
        if avg_ssim_filtered == global_min_ssim:
            optimal_ssim_p = sharpening_ssim_filtered[sv1]

    # Store optimal p values for this noise type
    optimal_psnr_p_dict[noise_label] = optimal_psnr_p
    optimal_ssim_p_dict[noise_label] = optimal_ssim_p

    # Plot PSNR
    ax = axes[idx, 0]
    ax.scatter(noise_levels, sharpening_psnr_baseline, label='Baseline PSNR')
    ax.scatter(noise_levels, sharpening_psnr_filtered, label='Filtered PSNR')
    ax.set_title(f'{noise_label}')
    ax.set_ylabel('PSNR')
    ax.legend()

    # Plot SSIM
    ax = axes[idx, 1]
    ax.scatter(noise_levels, sharpening_ssim_baseline, label='Baseline SSIM')
    ax.scatter(noise_levels, sharpening_ssim_filtered, label='Filtered SSIM')
    ax.set_title(f'{noise_label}')
    ax.set_ylabel('SSIM')
    ax.legend()

    # Plot optimal p
    ax = axes[idx, 2]
    ax.scatter(noise_levels, optimal_psnr_p, label='Optimal PSNR')
    ax.scatter(noise_levels, optimal_ssim_p, label='Optimal SSIM')
    ax.set_title(f'{noise_label}')
    ax.set_ylabel('Optimal p')
    ax.legend()

for ax in axes[-1, :]:
    ax.set_xlabel('Noise Level')

plt.tight_layout()
plt.show()
```

```
Gaussian noise 0 (psnr: 6.1) [100%]    [0/50 (00:00<?,?s/it, 0.00s/s]
Gaussian noise 1 (psnr: 6.1) [100%]    [0/50 (00:00<?,?s/it, 0.00s/s]
Gaussian noise 2 (psnr: 6.1) [100%]    [0/50 (00:00<?,?s/it, 0.00s/s]
Downsampling noise 0 (psnr: 2) [100%]  [0/50 (00:00<?,?s/it, 0.00s/s]
Downsampling noise 1 (psnr: 2) [100%]  [0/50 (00:00<?,?s/it, 0.00s/s]
Downsampling noise 2 (psnr: 2) [100%]  [0/50 (00:00<?,?s/it, 0.00s/s]
Motion blur noise 0 (psnr: 10) [100%]  [0/50 (00:00<?,?s/it, 0.00s/s]
Motion blur noise 1 (psnr: 10) [100%]  [0/50 (00:00<?,?s/it, 0.00s/s]
Motion blur noise 2 (psnr: 10) [100%]  [0/50 (00:00<?,?s/it, 0.00s/s]
Shadowed noise 0 (psnr: 5.7) [100%]    [0/50 (00:00<?,?s/it, 0.00s/s]
Shadowed noise 1 (psnr: 5.7) [100%]    [0/50 (00:00<?,?s/it, 0.00s/s]
Shadowed noise 2 (psnr: 5.7) [100%]    [0/50 (00:00<?,?s/it, 0.00s/s]
Overexposure noise 0 (psnr: 2) [100%]  [0/50 (00:00<?,?s/it, 0.00s/s]
Overexposure noise 1 (psnr: 2) [100%]  [0/50 (00:00<?,?s/it, 0.00s/s]
Overexposure noise 2 (psnr: 2) [100%]  [0/50 (00:00<?,?s/it, 0.00s/s]
```

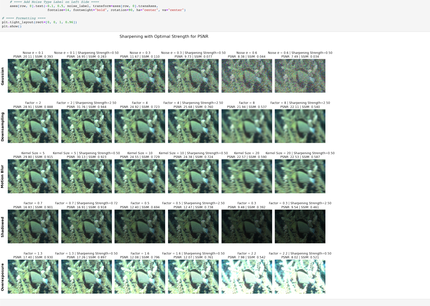


Show Optimum Strength on Example Images

```
# ==== Choose an Example Image ====
example_image_path = selected_images[1]
clean_image = img_as_float(io.imread('./image_dir/{example_image_path}'))

# ==== Create Figure ====
fig, axes = plt.subplots(4, 3, figsize=(15, 15))
fig.suptitle('Sharpening with Optimal Strength for BSSR', fontsize=16)

# ==== Process Each Noise Type ====
for idx, (noise_label, (noise_level, noise_fn)) in enumerate(zip(noise_types.items())):
    # Get noisy image
    clean = img_as_float(clean_image)
    noisy = noise_fn(clean, noise_level)

    # Retrieve the optimal sigma for this noise type and level
    optimal_sigma = optimal_psnr_p_dict[noise_label][noise_level]

    # Apply TV deconvolution using the optimal sigma
    filtered_image = denoise_tv_bregman(noisy, weight=1/optimal_sigma)

    # Compute PSNR & SSIM for both noisy & filtered images
    baseline_psnr = peak_signal_noise_ratio(clean, noisy)
    baseline_ssim = ssim(clean, noisy, data_range=1.0)
    filtered_psnr = peak_signal_noise_ratio(clean, filtered_image)
    filtered_ssim = ssim(clean, filtered_image, data_range=1.0)

    # Get filtered labels for Noise Level 0
    if noise_level == 0:
        filtered_label = 'Noise 0.1'
    elif noise_level == 1:
        filtered_label = 'Noise 0.3'
    else:
        filtered_label = 'Noise 0.6'

    if baseline_psnr_dict[noise_label][noise_level] > filtered_psnr:
        noisy_image = np.clip(noisy*255, 0, 255).astype(np.uint8)[0,0,:,:]  # Convert to uint8 for imshow
    else:
        noisy_image = np.clip(filtered_image*255, 0, 255).astype(np.uint8)[0,0,:,:]  # Convert to uint8 for imshow

    if filtered_ssim_dict[noise_label][noise_level] > baseline_ssim:
        filtered_image = np.clip(filtered_image*255, 0, 255).astype(np.uint8)[0,0,:,:]
    else:
        filtered_image = np.clip(filtered_image*255, 0, 255).astype(np.uint8)[0,0,:,:]

    # ==== Plot Noisy Image ====
    ax_noisy = axes[idx, 0]  # Column 0 for noisy images
    ax_noisy.imshow(noisy_image, cmap='gray')
    ax_noisy.set_title(f'{noise_label}\n{baseline_psnr:.2f}|{baseline_ssim:.2f}', fontsize=10)
    ax_noisy.axis('off')

    # ==== Plot Filtered Image ====
    ax_filtered = axes[idx, 1]  # Column 1 for filtered images
    ax_filtered.imshow(filtered_image, cmap='gray')
    ax_filtered.set_title(f'{filtered_label}\n{filtered_psnr:.2f}|{filtered_ssim:.2f}', fontsize=10)
    ax_filtered.axis('off')

    # ==== Add Noise Type Label on Left Side ====
    axes[idx, 0].text(-0.5, 0.5, noise_label, transform=axes[idx, 0].transAxes,
                      fontsize=12, fontweight='bold', rotation=90, ha='center', va='center')

# ==== Finalize Layout ====
plt.tight_layout(rect=[0.0, 0.1, 0.96])
plt.show()
```




```
# ==== Choose an Example Image ====
example_image_path = selected_images[1]
clean_image = img_as_float(io.imread('./image_dir/{example_image_path}'))

# ==== Create Figure ====
fig, axes = plt.subplots(4, 3, figsize=(15, 15))
fig.suptitle('Sharpening with Optimal Strength for PSNR', fontsize=16)

# ==== Process Each Noise Type ====
for idx, (noise_label, (noise_level, noise_fn)) in enumerate(zip(noise_types.items())):
    # Get noisy image
    clean = img_as_float(clean_image)
    noisy = noise_fn(clean, noise_level)

    # Retrieve the optimal sigma for this noise type and level
    optimal_sigma = optimal_psnr_p_dict[noise_label][noise_level]

    # Apply TV deconvolution using the optimal sigma
    filtered_image = denoise_tv_bregman(noisy, weight=1/optimal_sigma)

    # Compute PSNR & SSIM for both noisy & filtered images
    baseline_psnr = peak_signal_noise_ratio(clean, noisy)
    baseline_ssim = ssim(clean, noisy, data_range=1.0)
    filtered_psnr = peak_signal_noise_ratio(clean, filtered_image)
    filtered_ssim = ssim(clean, filtered_image, data_range=1.0)

    # Get filtered labels for Noise Level 0
    if noise_level == 0:
        filtered_label = 'Noise 0.1'
    elif noise_level == 1:
        filtered_label = 'Noise 0.3'
    else:
        filtered_label = 'Noise 0.6'

    if baseline_psnr_dict[noise_label][noise_level] > filtered_psnr:
        noisy_image = np.clip(noisy*255, 0, 255).astype(np.uint8)[0,0,:,:]  # Convert to uint8 for imshow
    else:
        noisy_image = np.clip(filtered_image*255, 0, 255).astype(np.uint8)[0,0,:,:]  # Convert to uint8 for imshow

    if filtered_ssim_dict[noise_label][noise_level] > baseline_ssim:
        filtered_image = np.clip(filtered_image*255, 0, 255).astype(np.uint8)[0,0,:,:]
    else:
        filtered_image = np.clip(filtered_image*255, 0, 255).astype(np.uint8)[0,0,:,:]

    # ==== Plot Noisy Image ====
    ax_noisy = axes[idx, 0]  # Column 0 for noisy images
    ax_noisy.imshow(noisy_image, cmap='gray')
    ax_noisy.set_title(f'{noise_label}\n{baseline_psnr:.2f}|{baseline_ssim:.2f}', fontsize=10)
    ax_noisy.axis('off')

    # ==== Plot Filtered Image ====
    ax_filtered = axes[idx, 1]  # Column 1 for filtered images
    ax_filtered.imshow(filtered_image, cmap='gray')
    ax_filtered.set_title(f'{filtered_label}\n{filtered_psnr:.2f}|{filtered_ssim:.2f}', fontsize=10)
    ax_filtered.axis('off')

    # ==== Add Noise Type Label on Left Side ====
    axes[idx, 0].text(-0.5, 0.5, noise_label, transform=axes[idx, 0].transAxes,
                      fontsize=12, fontweight='bold', rotation=90, ha='center', va='center')

# ==== Finalize Layout ====
plt.tight_layout(rect=[0.0, 0.1, 0.96])
plt.show()
```



Final Project Code - Enhancing & Revealing Hidden Image Details

Code Layout:

- Libaries and Functions
- Noising Images
- Testing Identifiers of Noise Types
- Filter Testing
- Deconvolution Testing
- Bicubic Interpolation Testing
- Exposure Correction Testing
- Sharpening Testing

Libraries and Dependencies

In [1]:
```python
import numpy as np
import cv2
import os
import random
import matplotlib.pyplot as plt
import scipy.io as sio
import skimage.io as io
from PIL import Image
from tqdm import tqdm
import pywt
import bm3d

from scipy.interpolate import griddata
from scipy.ndimage import gaussian_filter
from scipy.ndimage import median_filter
from scipy.signal import fftconvolve
from scipy.signal import convolve2d
import scipy.fftpack
import scipy.stats


from skimage.metrics import peak_signal_noise_ratio
from skimage.metrics import structural_similarity as ssim
from skimage.filters import gaussian
from skimage.restoration import denoise_tv_chambolle
from skimage.restoration import wiener
from skimage.restoration import richardson_lucy
from skimage import img_as_float
```

Functions

Denoising Evaluation Functions

In [2]:
```python
# Peak Signal-to-Noise Ratio
def calc_psnr(img1, img2):

    # Ensure images are the same shape
    assert img1.shape == img2.shape, "Images must have the same dimensions"

    # Set scale to [0,1]
    if img1.max() > 1:
        img1_scaled = img1 / 255.0
    else:
        img1_scaled = img1
    if img2.max() > 1:
        img2_scaled = img2 / 255.0
    else:
        img2_scaled = img2

    # Calc PSNR
    psnr_value = peak_signal_noise_ratio(img2_scaled,
                                         img1_scaled, data_range=1.0)

    return psnr_value

# Structural Similarity Index Measure
def calc_ssim(clean_image, noisy_image):

    # Ensure images are the same shape
    assert clean_image.shape == noisy_image.shape, "Images must have the same dimensions"

    # Determine pixel range
    range_clean = clean_image.max()
    range_noisy = noisy_image.max()

    # Normalize to [0,1] if needed
    if range_clean > 1 or range_noisy > 1:
        clean_image = clean_image.astype(np.float32) / 255.0
        noisy_image = noisy_image.astype(np.float32) / 255.0

    # Compute SSIM for each channel
    ssim_per_channel = [
        ssim(clean_image[:, :, i], noisy_image[:, :, i], data_range=1.0)
        for i in range(3)  # Loop through RGB channels
    ]

    return np.mean(ssim_per_channel)  # Average SSIM across channels
```

Noise Functions

In [3]:
```python
# Read Noise - Gaussian
def add_gaussian_noise(image, sigma=0.1):

    # Determine working range
    min_img = 0
    if np.max(image) <= 1:
        max_img = 1
    else:
        max_img = 255

    # Develop noise for each pixel
    noise = np.random.normal(0, sigma, image.shape).astype(np.float32)

    # Add noise into image and clip to size
    noisy_image = np.clip(image + noise*max_img, min_img, max_img)

    return noisy_image

# Low Resolution - Downsampling
def downsample_image(image, scale):

    # Determine working range
    min_img = 0
    if np.max(image) <= 1:
        max_img = 1
    else:
        max_img = 255

    # Current size
    h, w = image.shape[:2]

    # Apply Gaussian anti-aliasing (lowpass) filter
    sigma = scale / 2
    blurred = cv2.GaussianBlur(image, (0, 0), sigmaX=sigma, sigmaY=sigma)

    # Downsample the image
    small = cv2.resize(blurred, (w // scale, h // scale), interpolation=cv2.INTER_AREA)

    # Upsample back to original size for PSNR and SSIM calc
    upsampled = cv2.resize(small, (w, h), interpolation=cv2.INTER_LINEAR)

    return np.clip(upsampled, min_img, max_img)

# Motion Blur - Affine Warp
def apply_motion_blur(image, kernel_size=15):

    # Determine working range
    min_img = 0
    if np.max(image) <= 1:
        max_img = 1
    else:
        max_img = 255

    # Create kernel
    kernel = np.zeros((kernel_size, kernel_size))
    kernel[(kernel_size - 1) // 2, :] = np.ones(kernel_size)

    # Kernel across single direction
    kernel = cv2.warpAffine(kernel,
                            cv2.getRotationMatrix2D((kernel_size / 2, kernel_size / 2), 0, 1),
                            (kernel_size, kernel_size))
    kernel /= kernel_size

    # Apply blur to each channel separately and clip to size
    blurred = np.zeros_like(image)
    for i in range(3):  # Loop through RGB channels
        blurred[:, :, i] = cv2.filter2D(image[:, :, i], -1, kernel)


    return np.clip(blurred, min_img, max_img)

# Contrast Reduction - Underexposure / Overexposure
def add_shadows(image, factor=0.5):

    # Determine working range
    min_img = 0
    if np.max(image) <= 1:
        max_img = 1
    else:
        max_img = 255

    # Affect and clip
    return np.clip(image * factor, min_img, max_img)

def add_overexposure(image, factor=1.5):

    # Determine working range
    min_img = 0
    if np.max(image) <= 1:
        max_img = 1
    else:
        max_img = 255

    # Affect and clip
    return np.clip(image * factor, min_img, max_img)
```

Denoise Filter Functions

In [4]:
```python
# Gaussian Spatial Kernel - HW2
def fspecial_gaussian_2d(size, sigma):
    kernel = np.zeros(tuple(size))
    kernel[size[0]//2, size[1]//2] = 1
    kernel = gaussian(kernel, sigma)
    return kernel/np.sum(kernel)
```

In [5]:
```python
# Bilateral Filter - HW2 (adjusted to work for color image inputs)
def bilateral2d(img, radius, sigma, sigmaIntensity):
    pad = radius
    # Initialize filtered image to 0
    out = np.zeros_like(img)
    for channel in [0, 1, 2]:

        # Pad image to reduce boundary artifacts
        imgPad = np.pad(img[:,:,channel], pad)

        # Smoothing kernel, gaussian with standard deviation sigma
        # and size (2*radius+1, 2*radius+1)
        filtSize = (2*radius + 1, 2*radius + 1)
        spatialKernel = fspecial_gaussian_2d(filtSize, sigma)
        for y in range(img.shape[0]):
            for x in range(img.shape[1]):
                patchVals = imgPad[y+pad-radius:y+pad+radius+1, x+pad-radius:x+pad+radius+1]
                centerVal = imgPad[y+pad, x+pad]
                intensityWeights = np.exp(-((patchVals - centerVal) ** 2) / (2*sigmaIntensity**2))
                weights = spatialKernel * intensityWeights / np.sum(spatialKernel * intensityWeights)
                out[y, x, channel] = np.sum(patchVals * weights)

    return out
```

In [6]:
```python
# Non-Local Means Filter - HW2 (adjusted to work for color image inputs)
def inbounds(img, y, x):
    return 0 <= y and y < img.shape[0] and \
           0 <= x and x < img.shape[1]

def comparePatches(patch1, patch2, kernel, sigma):
    return np.exp(-np.sum(kernel*(patch1 - patch2) ** 2)/(2*sigma**2))

def nonlocalmeans(img, searchWindowRadius, averageFilterRadius, sigma, nlmSigma):
    # Initialize output to 0
    out = np.zeros_like(img)
    # Pad image to reduce boundary artifacts
    pad = max(averageFilterRadius, searchWindowRadius)
    for channel in [0, 1, 2]:
        imgPad = np.pad(img[...,channel:channel+1], pad)
        imgPad = imgPad[..., pad:-pad] # Don't pad color channel

        # Smoothing kernel
        filtSize = (2*averageFilterRadius + 1, 2*averageFilterRadius + 1)
        kernel = fspecial_gaussian_2d(filtSize, sigma)
        # Add third axis for broadcasting
        kernel = kernel[:, :, np.newaxis]
        for y in range(img.shape[0]):
            for x in range(img.shape[1]):
                centerPatch = imgPad[y+pad-averageFilterRadius:y+pad+averageFilterRadius+1,
                                     x+pad-averageFilterRadius:x+pad+averageFilterRadius+1,
                                     :]
                window = imgPad[y+pad-searchWindowRadius:y+pad+searchWindowRadius+1,
                                x+pad-searchWindowRadius:x+pad+searchWindowRadius+1,
                                :]
                # Go over a window around the current pixel, compute weights
                # based on difference of neighborhoods.
                weights = np.zeros((2*searchWindowRadius+1, 2*searchWindowRadius+1, 1))
                max_weight = 0
                for i, dy in enumerate(range(-searchWindowRadius, searchWindowRadius+1)):
                    for j, dx in enumerate(range(-searchWindowRadius, searchWindowRadius+1)):
                        if dy == 0 and dx == 0:
                            continue # Ignore center pixel
                        elif not inbounds(img, y+dy, x+dx):
                            continue # Ignore out-of-bounds pixels
                        otherPatch = imgPad[y+dy+pad-averageFilterRadius:y+dy+pad+averageFilterRadius+1,
                                            x+dx+pad-averageFilterRadius:x+dx+pad+averageFilterRadius+1,
                                            :]
                        weight = comparePatches(centerPatch, otherPatch, kernel, nlmSigma)
                        weights[i, j, 0] = weight
                        # Trick: Use center of window to store maximum weight
                        if weight > max_weight:
                            weights[searchWindowRadius, searchWindowRadius] = weight
                            max_weight = weight

                weights /= np.sum(weights)
                out[y, x, channel] = np.sum(weights * window, axis=(0,1))

    return out
```

In [7]:
```python
# Anisotropic Filter
def anisotropic(image, num_iterations=20, K=20, lambda_param=0.25):

    if np.max(image) <= 1:
        img = np.copy(image)
    else:
        img = np.copy(image.astype(np.float32) / 255)


    for _ in range(num_iterations):
        for c in range(3):  # Apply diffusion on each color channel separately
            grad_north = np.roll(img[:, :, c], -1, axis=0) - img[:, :, c]
            grad_south = np.roll(img[:, :, c], 1, axis=0) - img[:, :, c]
            grad_east = np.roll(img[:, :, c], -1, axis=1) - img[:, :, c]
            grad_west = np.roll(img[:, :, c], 1, axis=1) - img[:, :, c]

            c_north = np.exp(-(grad_north / K) ** 2)
            c_south = np.exp(-(grad_south / K) ** 2)
            c_east = np.exp(-(grad_east / K) ** 2)
            c_west = np.exp(-(grad_west / K) ** 2)

            img[:, :, c] += lambda_param * (
                c_north * grad_north + c_south * grad_south +
                c_east * grad_east + c_west * grad_west
            )

    return np.clip(img, 0, 1)
```

In [8]:
```python
# Total Variation Filter
def total_variation_denoising(image, weight=0.1):

    if np.max(image) <= 1:
        img = image
    else:
        img = image.astype(np.float32) / 255

    denoised_img = denoise_tv_chambolle(img, weight=weight, multichannel=True)

    return np.clip(denoised_img, 0, 1)
```

In [9]:
```python
#BM3D
def bm3d_denoise(image, sigma=0.3):
    """
    Applies BM3D denoising to a color image by processing each channel separately.

    Parameters:
        image (ndarray): Input color image (H, W, 3), expected in uint8 format.
        sigma (float): Noise standard deviation (0-1 range).

    Returns:
        ndarray: Denoised color image.
    """

    if np.max(image) <= 1:
        img = image
    else:
        img = image.astype(np.float32) / 255

    # Apply BM3D to each channel separately
    denoised_channels = [bm3d.bm3d(img[:, :, c], sigma_psd=sigma) for c in range(3)]

    # Stack channels back together and convert to uint8
    denoised_img = np.stack(denoised_channels, axis=-1)
    denoised_img = np.clip(denoised_img, 0, 1)

    return denoised_img
```

Deconvolution and Resolution Functions

In [10]:
```python
# Weiner Deconvolution
def wiener_deconvolution(image, psf_size=15, balance=0.1):

    if np.max(image) > 1:
        img = image.astype(np.float32) / 255.0  # Normalize if needed
    else:
        img = image.astype(np.float32)

    # Define the normalized PSF
    psf = np.ones((psf_size, psf_size)) / (psf_size ** 2)

    # Convert PSF to the frequency domain
    psf_fft = scipy.fftpack.fft2(psf, shape=img.shape[:2])

    # Apply Wiener deconvolution to each color channel
    deblurred_channels = []
    for c in range(3):
        img_fft = scipy.fftpack.fft2(img[:, :, c])  # Image to frequency domain
        wiener_filter = np.conj(psf_fft) / (np.abs(psf_fft) ** 2 + balance)  # Wiener filter
        deblurred_fft = img_fft * wiener_filter
        deblurred = np.real(scipy.fftpack.ifft2(deblurred_fft))  # Inverse FFT
        deblurred_channels.append(deblurred)

    # Stack back into a color image
    deblurred_image = np.stack(deblurred_channels, axis=-1)

    return np.clip(deblurred_image, 0, 1)  # Ensure values are in [0,1]
```

In [11]:
```python
# Richardson-Lucy Deconvolution
def motion_psf(psf_size, angle):

    psf = np.zeros((psf_size, psf_size))
    psf[psf_size // 2, :] = 1  # Create a horizontal line

    # Rotate to desired angle
    psf = cv2.warpAffine(psf, cv2.getRotationMatrix2D((psf_size // 2, psf_size // 2), angle, 1), (psf_size, psf_size))

    psf /= psf.sum()  # Normalize
    return psf

def rl_deconvolution(image, psf_size=15, num_iterations=30, angle=0):

    if np.max(image) > 1:
        img = image.astype(np.float32) / 255.0  # Normalize if needed
    else:
        img = image.astype(np.float32)

    # Generate motion blur PSF
    psf = motion_psf(psf_size, angle)

    # Apply RL deconvolution to each channel separately
    deblurred_channels = [richardson_lucy(img[:, :, c] + 1e-8, psf, iterations=num_iterations) for c in range(3)]

    # Stack the channels back into a color image
    deblurred_image = np.stack(deblurred_channels, axis=-1)

    return np.clip(deblurred_image, 0, 1)  # Ensure values are in [0,1]
```

In [12]:
```python
def gaussian_motion_psf(psf_size, angle):

    # Ensure PSF size is always odd (needed for GaussianBlur)
    if psf_size % 2 == 0:
        psf_size += 1

    psf = np.zeros((psf_size, psf_size))
    psf[psf_size // 2, :] = 1  # Create a horizontal line

    # Rotate to desired angle
    psf = cv2.warpAffine(psf, cv2.getRotationMatrix2D((psf_size // 2, psf_size // 2), angle, 1), (psf_size, psf_size))

    psf = cv2.GaussianBlur(psf, (psf_size, psf_size), psf_size / 3)  # Apply Gaussian blur
    psf /= psf.sum()  # Normalize
    return psf

def rl_deconvolution(image, psf_size=15, num_iterations=30, angle=0):

    if np.max(image) > 1:
        img = image.astype(np.float32) / 255.0  # Normalize if needed
    else:
        img = image.astype(np.float32)

    # Generate motion blur PSF
    psf = gaussian_motion_psf(psf_size, angle)

    # Apply RL deconvolution to each channel separately
    deblurred_channels = [richardson_lucy(img[:, :, c] + 1e-8, psf, iterations=num_iterations) for c in range(3)]

    # Stack the channels back into a color image
    deblurred_image = np.stack(deblurred_channels, axis=-1)

    return np.clip(deblurred_image, 0, 1)  # Ensure values are in [0,1]
```

Tonemapping & Exposure Correcting Functions

In [13]:
```python
# Adaptive Histogram Equalization
def correct_exposure(image, size=8):

    if np.max(image) > 1:
        img = image.astype(np.float32) / 255.0  # Normalize if needed
    else:
        img = image.astype(np.float32)

    # Convert from RGB to LAB (cv2 expects uint8 input)
    lab = cv2.cvtColor((img * 255).astype(np.uint8), cv2.COLOR_RGB2LAB)

    # Split LAB channels
    l, a, b = cv2.split(lab)

    # Apply Adaptive Histogram Equalization (CLAHE) on L channel
    clahe = cv2.createCLAHE(clipLimit=3.0, tileGridSize=(size,size))
    l = clahe.apply(l)

    # Merge LAB channels
    lab = cv2.merge((l, a, b))

    # Convert back to RGB and normalize to 0-1 range
    corrected_image = cv2.cvtColor(lab, cv2.COLOR_LAB2RGB).astype(np.float32) / 255.0

    return np.clip(corrected_image, 0, 1)  # Ensure valid range
```

In [14]:
```python
# Laplacian Tonemapping
def local_laplacian_tone_mapping(image, alpha=1.5, beta=0.5):

    if np.max(image) > 1:
        img = image.astype(np.float32) / 255.0  # Normalize if needed
    else:
        img = image.astype(np.float32)

    # Convert from RGB to LAB (cv2 expects uint8 input)
    img_lab = cv2.cvtColor((img * 255).astype(np.uint8), cv2.COLOR_RGB2LAB)

    # Split LAB channels
    l_channel, a, b = cv2.split(img_lab)

    # Apply CLAHE (Contrast Limited Adaptive Histogram Equalization) on L channel
    clahe = cv2.createCLAHE(clipLimit=alpha, tileGridSize=(8, 8))
    l_enhanced = clahe.apply(l_channel)

    # Merge enhanced L channel with original A, B channels
    img_lab = cv2.merge((l_enhanced, a, b))

    # Convert back to RGB and normalize to 0-1 range
    img_tone_mapped = cv2.cvtColor(img_lab, cv2.COLOR_LAB2RGB).astype(np.float32) / 255.0

    return np.clip(img_tone_mapped, 0, 1)  # Ensure valid range
```

In [15]:
```python
# Bilateral Tonemapping
def bilateral_tone_mapping(image, sigma_spatial=15, sigma_range=0.5):

    if np.max(image) > 1:
        img = image.astype(np.float32) / 255.0  # Normalize if needed
    else:
        img = image.astype(np.float32)

    # Convert RGB to LAB (cv2 expects uint8 input)
    img_lab = cv2.cvtColor((img * 255).astype(np.uint8), cv2.COLOR_RGB2LAB)

    # Split LAB channels
    l_channel, a, b = cv2.split(img_lab)

    # Apply bilateral filter to L channel (convert to uint8 for filtering)
    l_base = cv2.bilateralFilter(l_channel.astype(np.uint8), d=0, sigmaColor=sigma_range * 255, sigmaSpace=sigma_spatial)

    # Extract details (convert L channel back to float before subtraction)
    l_detail = l_channel.astype(np.float32) / 255.0 - l_base.astype(np.float32) / 255.0

    # Compress base layer to prevent overexposure
    l_compressed = np.log1p(l_base.astype(np.float32)) / np.log1p(255)  # Log-based compression

    # Reconstruct the L channel
    l_final = l_compressed + l_detail  # Add details back

    # Normalize L channel to [0, 255]
    l_final = cv2.normalize(l_final, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

    # Merge channels and convert back to RGB
    img_lab = cv2.merge((l_final, a, b))
    img_tone_mapped = cv2.cvtColor(img_lab, cv2.COLOR_LAB2RGB).astype(np.float32) / 255.0  # Normalize output

    return np.clip(img_tone_mapped, 0, 1)  # Ensure valid range
```

Sharpening Functions

In [16]:
```python
# Bucubic Interpolation
def upscale_image(image, scale=2):

    # Ensure the image is in float format (0-1)
    if np.max(image) > 1:
        img = image.astype(np.float32) / 255.0  # Normalize if needed
    else:
        img = image.astype(np.float32)

    h, w = img.shape[:2]

    # Upscale using bicubic interpolation
    upscaled = cv2.resize(img, (w * scale, h * scale), interpolation=cv2.INTER_CUBIC)

    # Resize back to original size to ensure valid PSNR/SSIM comparison
    final_image = cv2.resize(upscaled, (w, h), interpolation=cv2.INTER_CUBIC)

    return np.clip(final_image, 0, 1)  # Ensure values stay in [0,1]
```

In [17]:
```python
# Unsharp Masking
def sharpen_image(image, sigma=1.5, strength=1.5):

    # Ensure image is in float format (0-1)
    if np.max(image) > 1:
        img = image.astype(np.float32) / 255.0  # Normalize if needed
    else:
        img = image.astype(np.float32)

    # Apply Gaussian blur
    blurred = cv2.GaussianBlur(img, (5, 5), sigmaX=sigma, sigmaY=sigma)

    # Apply Unsharp Masking
    sharpened = cv2.addWeighted(img, 1 + strength, blurred, -strength, 0)

    return np.clip(sharpened, 0, 1)  # Ensure values are in [0,1]
```

All Noise Types Presented

Image Selection

In [18]:
```python
image_dir = "BSDS300/images/train/"
num_images = 50
random.seed(42)
selected_images = random.sample(os.listdir(image_dir), num_images)
```

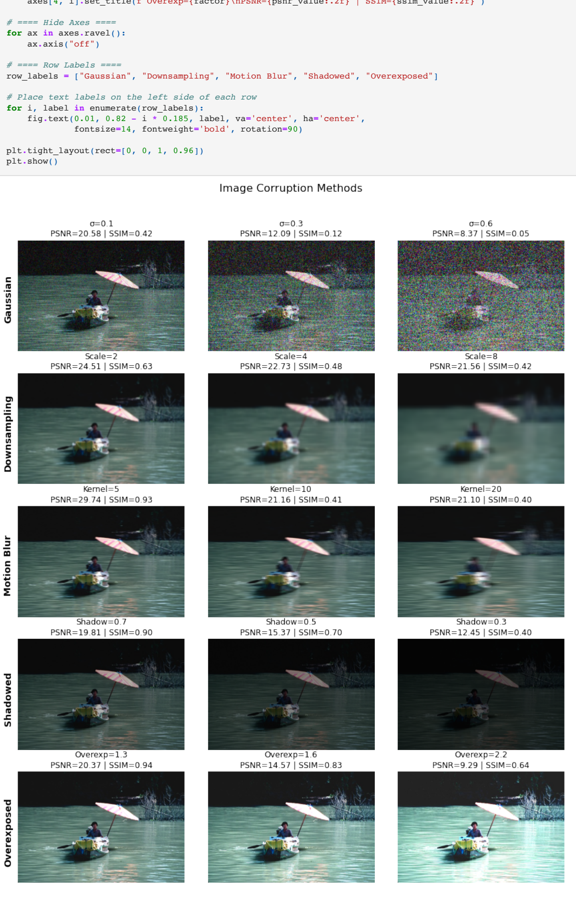
Noise Types & Levels

For each noise type, we choose 3 levels to test our denoising strategies

In [19]:
```python
noise_types = {
    "Gaussian": ([0.1, 0.3, 0.6], add_gaussian_noise),
    "Downsampling": ([2, 4, 8], downsample_image),
    "Motion Blur": ([5, 10, 20], apply_motion_blur),
    "Shadowed": ([0.7, 0.5, 0.3], add_shadows),
    "Overexposure": ([1.3, 1.6, 2.2], add_overexposure)
}
```

Noise Examples

In [22]:
```python
# ==== Select an Example Image ====
example_image_path = selected_images[1]  # Choose the first image from dataset
clean = img_as_float(io.imread(f"{image_dir}/{example_image_path}"))

# ==== Settings for Each Noise Type ====
gaussian_levels = [0.1, 0.3, 0.6]
downsample_levels = [2, 4, 8]
motion_blur_levels = [5, 10, 20]
shadow_levels = [0.7, 0.5, 0.3]  # Darker as factor decreases
overexposure_levels = [1.3, 1.6, 2.2]  # Brighter as factor increases

# ==== Generate Subplots ====
fig, axes = plt.subplots(5, 3, figsize=(12, 15))
fig.suptitle("Image Corruption Methods", fontsize=16)

# ==== Apply and Display Each Type of Noise ====
for i, sigma in enumerate(gaussian_levels):
    noisy = add_gaussian_noise(clean, sigma)
    psnr_value = calc_psnr(clean, noisy)
    ssim_value = calc_ssim(clean, noisy)
    axes[0, i].imshow(noisy)
    axes[0, i].set_title(f"σ={sigma}\nPSNR={psnr_value:.2f} | SSIM={ssim_value:.2f}")

for i, scale in enumerate(downsample_levels):
    noisy = downsample_image(clean, scale)
    psnr_value = calc_psnr(clean, noisy)
    ssim_value = calc_ssim(clean, noisy)
    axes[1, i].imshow(noisy)
    axes[1, i].set_title(f"Scale={scale}\nPSNR={psnr_value:.2f} | SSIM={ssim_value:.2f}")

for i, kernel_size in enumerate(motion_blur_levels):
    noisy = apply_motion_blur(clean, kernel_size)
    psnr_value = calc_psnr(clean, noisy)
    ssim_value = calc_ssim(clean, noisy)
    axes[2, i].imshow(noisy)
    axes[2, i].set_title(f"Kernel={kernel_size}\nPSNR={psnr_value:.2f} | SSIM={ssim_value:.2f}")

for i, factor in enumerate(shadow_levels):
    noisy = add_shadows(clean, factor)
    psnr_value = calc_psnr(clean, noisy)
    ssim_value = calc_ssim(clean, noisy)
    axes[3, i].imshow(noisy)
    axes[3, i].set_title(f"Shadow={factor}\nPSNR={psnr_value:.2f} | SSIM={ssim_value:.2f}")

for i, factor in enumerate(overexposure_levels):
    noisy = add_overexposure(clean, factor)
    psnr_value = calc_psnr(clean, noisy)
    ssim_value = calc_ssim(clean, noisy)
    axes[4, i].imshow(noisy)
    axes[4, i].set_title(f"Overexp={factor}\nPSNR={psnr_value:.2f} | SSIM={ssim_value:.2f}")

# ==== Hide Axes ====
for ax in axes.ravel():
    ax.axis("off")

# ==== Row Labels ====
row_labels = ["Gaussian", "Downsampling", "Motion Blur", "Shadowed", "Overexposed"]

# Place text labels on the left side of each row
for i, label in enumerate(row_labels):
    fig.text(0.01, 0.82 - i * 0.185, label, va='center', ha='center',
             fontsize=14, fontweight='bold', rotation=90)

plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```



Identify Noise Sources

Can we determine what types of noise are in an image?

In [23]:
```python
def compute_exposure_metrics(image):

    # Convert to grayscale
    image_uint8 = (image * 255).clip(0, 255).astype(np.uint8)
    gray = cv2.cvtColor(image_uint8, cv2.COLOR_RGB2GRAY)

    # Compute histogram
    hist = cv2.calcHist([gray], [0], None, [256], [0, 256])

    # Compute average intensity
    average_intensity = int(np.mean(gray))

    # Compute peak intensity
    peak_intensity = int(np.argmax(hist))

    # Compute standard deviation (spread of intensities)
    std_dev = np.std(gray)

    # Compute skewness (left-skew = shadow, right-skew = overexposure)
    skewness = scipy.stats.skew(gray.flatten())

    # Compute contrast (difference between 5th and 95th percentile)
    p5, p95 = np.percentile(gray, [5, 95])
    contrast = p95 - p5

    return {
        "Average Intensity": average_intensity,
        "Peak Intensity": peak_intensity,
        "Std Dev": round(std_dev, 3),
        "Skewness": round(skewness, 3),
        "Contrast": round(contrast, 3),
    }
```

In [24]:
```python
def compute_blur(image):

    # Convert image to grayscale
    gray = cv2.cvtColor((image * 255).astype(np.uint8), cv2.COLOR_RGB2GRAY)

    # Compute FFT and take log magnitude
    fft_image = np.log(np.abs(scipy.fftpack.fftshift(scipy.fftpack.fft2(gray))) + 1)

    return np.mean(fft_image)
```

In [25]:
```python
def compute_laplacian_blur(image):

    # Convert image to grayscale
    gray = cv2.cvtColor((image * 255).astype(np.uint8), cv2.COLOR_RGB2GRAY)

    # Compute Laplacian variance
    laplacian_var = cv2.Laplacian(gray, cv2.CV_64F).var()

    return laplacian_var
```

In [29]:
```python
# Store results across all images
results_total = []

print("\n=== Laplacian Blur Estimate Averaged Across 50 Images ===")
print("-" * 60)
print(f"{'Noise Type':<20} {'Noise Level':<10} {'Laplacian Blur Variance (50-image Avg)'}")
print("-" * 60)

# Process each noise type
for noise_type, (noise_levels, noise_fn) in noise_types.items():
    for noise_level in noise_levels:
        laplacian_values = []

        # Loop over 50 images
        for img_name in selected_images:
            img_path = f"{image_dir}/{img_name}"
            clean_image = img_as_float(io.imread(img_path))

            # Apply noise
            noisy_image = noise_fn(clean_image, noise_level)

            # Compute Laplacian blur estimate
            laplacian_values.append(compute_laplacian_blur(noisy_image))

        # Store and print the average result
        avg_laplacian = round(np.mean(laplacian_values), 3)
        results_total.append((noise_type, noise_level, avg_laplacian))
        print(f"{noise_type:<20} {noise_level:<10} {avg_laplacian:.3f}")
```

```
=== Laplacian Blur Estimate Averaged Across 50 Images ===
------------------------------------------------------------
Noise Type           Noise Level Laplacian Blur Variance (50-image Avg)
------------------------------------------------------------
Gaussian             0.1        6846.812
Gaussian             0.3        37176.519
Gaussian             0.6        77860.939
Downsampling         2          35.343
Downsampling         4          6.796
Downsampling         8          2.473
Motion Blur          5          373.964
Motion Blur          10         225.727
Motion Blur          20         141.311
Shadowed             0.7        774.594
Shadowed             0.5        395.808
Shadowed             0.3        143.454
Overexposure         1.3        2261.507
Overexposure         1.6        2624.629
Overexposure         2.2        2742.791
```

Massively Noisy Image Example

In [27]:
```python
example_image_path = selected_images[7]
clean = img_as_float(io.imread(f"{image_dir}/{example_image_path}"))
noisy = downsample_image(clean, 2)
noisy = add_shadows(noisy, 0.5)
noisy = apply_motion_blur(noisy, 10)
noisy = add_gaussian_noise(noisy, 0.3)

# Define PSF sizes

bm = 0.3  # Adjust PSF sizes
denoised = bm3d_denoise(noisy, sigma=bm)

# Plot
plt.figure(figsize=(15, 10))

plt.subplot(1, 3, 1)
plt.imshow(clean, cmap='gray')
plt.title(f"Clean Image")
plt.axis("off")

plt.subplot(1, 3, 2)
plt.imshow(noisy, cmap='gray')
```

```
pe = calc_psnr(noisy,clean)
ss = calc_ssim(clean, noisy)
plt.title(f"Noisy| PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(denoised, cmap='gray')
pe = calc_psnr(denoised,clean)
ss = calc_ssim(clean, denoised)
plt.title(f"BM3D Only | PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off')
```

In [29]:

```
example_image_path = selected_images[7]
clean = img_as_float(io.imread(f"{image_dir}/{example_image_path}"))
noisy = downsample_image(clean, 2)
noisy = add_shadows(noisy, 0.5)
noisy = apply_motion_blur(noisy, 10)
noisy = add_gaussian_noise(noisy, 0.3)

# Define PSF sizes

bm = 0.3 # Adjust PSF sizes
denoised = bm3d_denoise(noisy, sigma=bm)
sh = 4  # Adjust PSF sizes
denoised2 = correct_exposure(denoised, size=sh)

# Plot
plt.figure(figsize=(15, 10))

plt.subplot(1, 3, 1)
plt.imshow(noisy, cmap='gray')
pe = calc_psnr(noisy,clean)
ss = calc_ssim(clean, noisy)
plt.title(f"Noisy| PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(denoised, cmap='gray')
pe = calc_psnr(denoised,clean)
ss = calc_ssim(clean, denoised)
plt.title(f"BM3D Only | PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(denoised2, cmap='gray')
pe = calc_psnr(denoised2,clean)
ss = calc_ssim(clean, denoised2)
plt.title(f"BM3D -> Hist. Eq. | PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off');
```

In [30]:

```
example_image_path = selected_images[7]
clean = img_as_float(io.imread(f"{image_dir}/{example_image_path}"))
noisy = downsample_image(clean, 2)
noisy = add_shadows(noisy, 0.5)
noisy = apply_motion_blur(noisy, 10)
noisy = add_gaussian_noise(noisy, 0.3)

# Define PSF sizes

sh = 4  # Adjust PSF sizes
denoised = correct_exposure(noisy, size=sh)
bm = 0.3 # Adjust PSF sizes
denoised2 = bm3d_denoise(denoised, sigma=bm)

# Plot
plt.figure(figsize=(15, 10))

plt.subplot(1, 3, 1)
plt.imshow(noisy, cmap='gray')
pe = calc_psnr(noisy,clean)
ss = calc_ssim(clean, noisy)
plt.title(f"Noisy| PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(denoised, cmap='gray')
pe = calc_psnr(denoised,clean)
ss = calc_ssim(clean, denoised)
plt.title(f"Hist. Eq. | PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(denoised2, cmap='gray')
pe = calc_psnr(denoised2,clean)
ss = calc_ssim(clean, denoised2)
plt.title(f"BM3D -> Hist. Eq. | PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off');
```

In [34]:

```
example_image_path = selected_images[7]
clean = img_as_float(io.imread(f"{image_dir}/{example_image_path}"))
noisy = downsample_image(clean, 2)
noisy = add_shadows(noisy, 0.5)
noisy = apply_motion_blur(noisy, 10)
noisy = add_gaussian_noise(noisy, 0.3)


bm = 0.3
denoised1 = bm3d_denoise(noisy, sigma=bm)
sh = 4
denoised2 = correct_exposure(denoised1, size=sh)
lla = 0.2
denoised3 = local_laplacian_tone_mapping(denoised2, alpha=lla, beta=0.3)


# Plot
plt.figure(figsize=(15, 10))

plt.subplot(1, 3, 1)
plt.imshow(noisy, cmap='gray')
pe = calc_psnr(noisy,clean)
ss = calc_ssim(clean, noisy)
plt.title(f"Noisy| PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(denoised1, cmap='gray')
pe = calc_psnr(denoised1,clean)
ss = calc_ssim(clean, denoised1)
plt.title(f"BM3D -> Hist. Eq. | PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(denoised3, cmap='gray')
pe = calc_psnr(denoised3,clean)
ss = calc_ssim(clean, denoised3)
plt.title(f"BM3D -> Hist. Eq. -> Lapl. Tone | PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off');
```

In [43]:

```
example_image_path = selected_images[7]
clean = img_as_float(io.imread(f"{image_dir}/{example_image_path}"))
noisy = downsample_image(clean, 2)
noisy = add_shadows(noisy, 0.5)
noisy = apply_motion_blur(noisy, 10)
noisy = add_gaussian_noise(noisy, 0.3)


bm = 0.3
denoised1 = bm3d_denoise(noisy, sigma=bm)
sh = 4
denoised2 = correct_exposure(denoised1, size=sh)
lla = 0.2
denoised3 = local_laplacian_tone_mapping(denoised2, alpha=lla, beta=0.3)
st = 0.4
denoised4 = sharpen_image(denoised3, sigma=0.2, strength=st)


# Plot
plt.figure(figsize=(15, 10))

plt.subplot(1, 3, 1)
plt.imshow(noisy, cmap='gray')
pe = calc_psnr(noisy,clean)
ss = calc_ssim(clean, noisy)
plt.title(f"Noisy| PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(denoised3, cmap='gray')
pe = calc_psnr(denoised3,clean)
ss = calc_ssim(clean, denoised3)
plt.title(f"BM3D -> ... -> Lapl. Tone | PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(denoised4, cmap='gray')
pe = calc_psnr(denoised4,clean)
ss = calc_ssim(clean, denoised4)
plt.title(f"BM3D -> ... -> Unsharp | PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off');
```

In [48]:

```
example_image_path = selected_images[13]
clean = img_as_float(io.imread(f"{image_dir}/{example_image_path}"))
noisy = add_overexposure(clean, 2.2)
noisy = add_gaussian_noise(noisy, 0.3)
noisy = apply_motion_blur(noisy, 10)

bsig=0.6
denoised1 = bilateral2d(noisy, radius=3, sigma=bsig, sigmaIntensity=0.25)
lla = 1
denoised2 = local_laplacian_tone_mapping(denoised1, alpha=lla, beta=0.3)

# Plot
plt.figure(figsize=(15, 10))

plt.subplot(1, 3, 1)
plt.imshow(noisy, cmap='gray')
pe = calc_psnr(noisy,clean)
ss = calc_ssim(clean, noisy)
plt.title(f"Noisy| PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(denoised1, cmap='gray')
pe = calc_psnr(denoised1,clean)
ss = calc_ssim(clean, denoised1)
plt.title(f"Bil. Filt. | PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(denoised2, cmap='gray')
pe = calc_psnr(denoised2,clean)
ss = calc_ssim(clean, denoised2)
plt.title(f"Bil. Filt. -> Lapl. Tone | PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off');
```

In [49]:

```
example_image_path = selected_images[13]
clean = img_as_float(io.imread(f"{image_dir}/{example_image_path}"))
noisy = add_overexposure(clean, 2.2)
noisy = add_gaussian_noise(noisy, 0.3)
noisy = apply_motion_blur(noisy, 10)

bsig=0.6
denoised1 = bilateral2d(noisy, radius=3, sigma=bsig, sigmaIntensity=0.25)
btm = 0.3
denoised2 = bilateral_tone_mapping(denoised1, sigma_spatial=15, sigma_range=btm)

# Plot
plt.figure(figsize=(15, 10))

plt.subplot(1, 3, 1)
plt.imshow(noisy, cmap='gray')
pe = calc_psnr(noisy,clean)
ss = calc_ssim(clean, noisy)
plt.title(f"Noisy| PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(denoised1, cmap='gray')
pe = calc_psnr(denoised1,clean)
ss = calc_ssim(clean, denoised1)
plt.title(f"Bil. Filt. | PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(denoised2, cmap='gray')
pe = calc_psnr(denoised2,clean)
ss = calc_ssim(clean, denoised2)
plt.title(f"Bil. Filt. -> Bil. Tone | PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off');
```

In [53]:

```
example_image_path = selected_images[2]
clean = img_as_float(io.imread(f"{image_dir}/{example_image_path}"))
noisy = add_overexposure(clean, 2.2)
noisy = add_gaussian_noise(noisy, 0.3)
noisy = apply_motion_blur(noisy, 10)

denoised1 = wiener_deconvolution(noisy, psf_size=10, balance=5.01*10)
denoised2 = bm3d_denoise(denoised1, sigma=0.3)

# Plot
plt.figure(figsize=(15, 10))

plt.subplot(1, 3, 1)
plt.imshow(noisy, cmap='gray')
pe = calc_psnr(noisy,clean)
ss = calc_ssim(clean, noisy)
plt.title(f"Noisy| PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(denoised1, cmap='gray')
pe = calc_psnr(denoised1,clean)
ss = calc_ssim(clean, denoised1)
plt.title(f"Wie. Deconv. | PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(denoised2, cmap='gray')
pe = calc_psnr(denoised2,clean)
ss = calc_ssim(clean, denoised2)
plt.title(f"Wie.Deconv. -> BM3D | PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off');
```

In [54]:

```
example_image_path = selected_images[2]
clean = img_as_float(io.imread(f"{image_dir}/{example_image_path}"))
noisy = add_overexposure(clean, 2.2)
noisy = add_gaussian_noise(noisy, 0.3)
noisy = apply_motion_blur(noisy, 10)

denoised1 = bm3d_denoise(noisy, sigma=0.3)
denoised2 = wiener_deconvolution(denoised1, psf_size=10, balance=5.01*10)

# Plot
plt.figure(figsize=(15, 10))

plt.subplot(1, 3, 1)
plt.imshow(noisy, cmap='gray')
pe = calc_psnr(noisy,clean)
ss = calc_ssim(clean, noisy)
plt.title(f"Noisy| PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(denoised1, cmap='gray')
pe = calc_psnr(denoised1,clean)
ss = calc_ssim(clean, denoised1)
plt.title(f"BM3D. | PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(denoised2, cmap='gray')
pe = calc_psnr(denoised2,clean)
ss = calc_ssim(clean, denoised2)
plt.title(f"BM3D -> Wie.Deconv. | PSNR={pe:.2f} | SSIM={ss:.2f}")
plt.axis('off');
```

In [ ]: