Project 1 - Bilateral Filter, BM3D, and ADMM-PnP

Yoav Ellinson 206036949
Ofir Miran 206564189

Spring 2025

All the code for this project can be found here:
`https://github.com/yoavellinson/signals_and_images_course/tree/main/project_1`

1 Project questions

Q1. We ran hyper parameter tuning and got that the best results were with $\sigma_s = 1.7$ and $\sigma_r = 0.26$. Code in Appendix 1. In Table 1 we can see that our Bilateral filter's performance is not as good

Picture	**Input PSNR**	**Bilateral Output PSNR**	**BM3D Output PSNR**
1 Cameraman256	20.04	27.12	29.42
2 house	20.00	28.84	32.88
3 peppers256	20.01	27.48	30.11
4 Lena512	20.01	29.12	31.89
5 barbara	19.99	25.63	30.53
6 boat	19.99	27.50	29.78
7 hill	20.01	28.16	29.77
8 couple	20.02	27.14	29.64
Average	**20.01**	**27.62**	**30.50**

Table 1: A summary of PSNR results for the Bilateral filter and the BM3D at denoising

as the BM3D's, reaching average PSNR of 27.62 and 30.5, respectively. visual results in Fig 1.

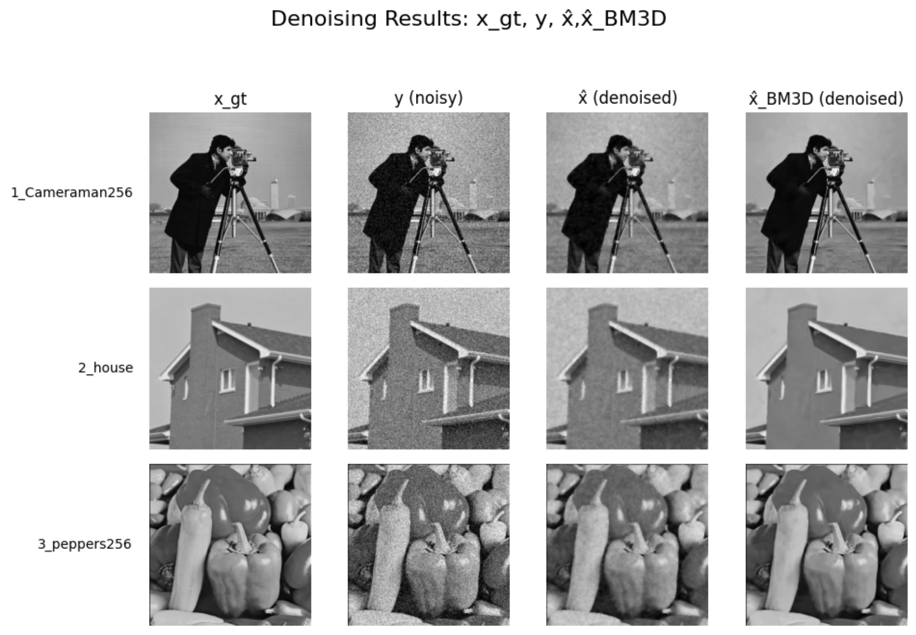

Figure 1: Visual results of the denoising

Q2. (a) In this part we will formulate the ADMM-PnP scheme that we will use to deblurr the images. The blurred and noised image y is formulated as:

$$\boldsymbol{y} = \boldsymbol{x} * h + \boldsymbol{n} \tag{1}$$

where x is the original image, h is the blurring kernel and n is the additive gaussian noise. The optimization problem to be solved:

$$\min_{x} \frac{1}{2}||\boldsymbol{y} - \boldsymbol{x} * k||^2 + \beta s(\boldsymbol{x}) \tag{2}$$

In order to solve the optimization problem via the ADMM-PnP framework, we introduce an auxiliary variable $\boldsymbol{v}$ such that $\boldsymbol{x} = \boldsymbol{v}$ ($\boldsymbol{z}$in the lectures - we went by the paper's notation) . This reformulates the problem as:

$$\min_{\boldsymbol{x},\boldsymbol{v}} \ \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{x} * h\|^2 + \beta s(\boldsymbol{v}) \quad \text{subject to } \boldsymbol{x} = \boldsymbol{v} \tag{3}$$

We then write the augmented Lagrangian for this constrained optimization problem:

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{v}, \boldsymbol{u}) = \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{x} * h\|^2 + \beta s(\boldsymbol{v}) + \frac{\rho}{2}\|\boldsymbol{x} - \boldsymbol{v} + \boldsymbol{u}\|^2 - \frac{\rho}{2}\|\boldsymbol{u}\|^2 \tag{4}$$

where $\boldsymbol{u}$ is the scaled dual variable and ρ is the penalty parameter.
The ADMM iterations consist of the following updates:

i. **$\boldsymbol{x}$-update:**

$$\boldsymbol{x}^{(k+1)} = \arg\min_{x} \ \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{x} * h\|^2 + \frac{\rho}{2}\|\boldsymbol{x} - \boldsymbol{v}^{(k)} + \boldsymbol{u}^{(k)}\|^2 \tag{5}$$

ii. **$\boldsymbol{v}$-update:**

$$\boldsymbol{v}^{(k+1)} = \arg\min_{v} \ \beta s(\boldsymbol{v}) + \frac{\rho}{2}\|\boldsymbol{x}^{(k+1)} - \boldsymbol{v} + \boldsymbol{u}^{(k)}\|^2 \tag{6}$$

This step is interpreted as a denoising operation, so we use a denoiser $\mathcal{D}_\sigma$ instead of solving this subproblem exactly:

$$\boldsymbol{v}^{(k+1)} = \mathcal{D}_\sigma\left(\boldsymbol{x}^{(k+1)} + \boldsymbol{u}^{(k)}\right) \tag{7}$$

iii. **u-update:**

$$\boldsymbol{u}^{(k+1)} = \boldsymbol{u}^{(k)} + \boldsymbol{x}^{(k+1)} - \boldsymbol{v}^{(k+1)} \tag{8}$$

Here, $\mathcal{D}_\sigma$ is a denoising operator that implicitly defines the prior $s(\boldsymbol{x})$, and $\sigma = \sqrt{\beta/\rho}$ controls its strength. This is the core idea of Plug-and-Play priors: replacing an explicit regularizer with a powerful off-the-shelf denoiser. In practice we just searched optimized for the hype-parametr - σ which defines the strength of the denoiser.
The data Fidelity update has a closed form solution:

$$\boldsymbol{x}^{(k+1)} = (\boldsymbol{A}^T\boldsymbol{A} + \rho\boldsymbol{I})^{-1}(\boldsymbol{A}^T\boldsymbol{y} + \rho\tilde{\boldsymbol{x}}^{(k)}) \tag{9}$$

Where $\tilde{\boldsymbol{x}} = \boldsymbol{v} - \boldsymbol{u}$
The lectures shows that:

$$\boldsymbol{A}^T\boldsymbol{y} = \mathcal{F}^{-1}\big(\overline{\mathcal{F}(h)}\mathcal{F}(\boldsymbol{y})\big)$$
$$(\boldsymbol{A}^T\boldsymbol{A} + \rho\boldsymbol{I}) = \mathcal{F}^{-1}\big(|\mathcal{F}(h)|^2\big) + \rho$$

Therefore (9) can be written as:

$$\boldsymbol{x}^{(k)} = \mathcal{F}^{-1}\big(\frac{\overline{\mathcal{F}(h)}\mathcal{F}(\boldsymbol{y}) + \rho\mathcal{F}(\tilde{\boldsymbol{x}}^{(k)})}{|\mathcal{F}(h)|^2 + \rho}\big) \tag{10}$$

Where $\mathcal{F}(\cdot)$ is the DFT operation and $\overline{(\cdot)}$ is the complex conjugate operation.

(b) We implemented the ADMM-PnP method in Python (Appendix 2) and got the following results. The results are summarized in Table 2 and includes both the bilateral filter and BM3D as the denoisers. Visual results can be found in Figs 2,3. The improvment in avarage output PSNR is clear when using both denosiers, but the BM3D seems to be much more sutible for that problem.

Picture	**Input PSNR**	**Bilateral Filter**	**BM3D**	**Modified BM3D**
1 Cameraman256	21.0	26.75	28.86	28.92
2 house	24.23	32.11	34.12	34.12
3 peppers256	21.26	26.96	31.13	31.18
4 Lena512	25.5	32.36	33.84	33.83
5 barbara	22.19	24.8	27.39	27.44
6 boat	23.68	29.52	30.98	31.03
7 hill	25.14	30.37	31.09	31.11
8 couple	23.64	29.21	30.70	30.77
Average	**23.33**	**29.21**	**31.01**	**31.05**

Table 2: A summary of PSNR [dB] results for the Bilateral filter and the BM3D at denoising, the columns of the modified BM3D is where we adjust ρ and σ while iterating.

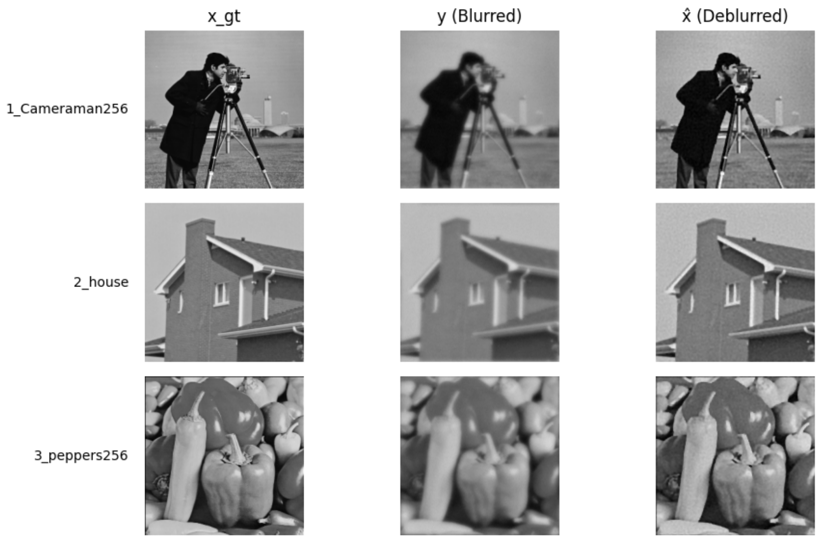

Figure 2: Visual results of the deblurring using the bilateral filter as the denoiser, hyper-parameters: $\sigma_s = 0.5, \sigma_r = 0.08, \rho = 0.2$

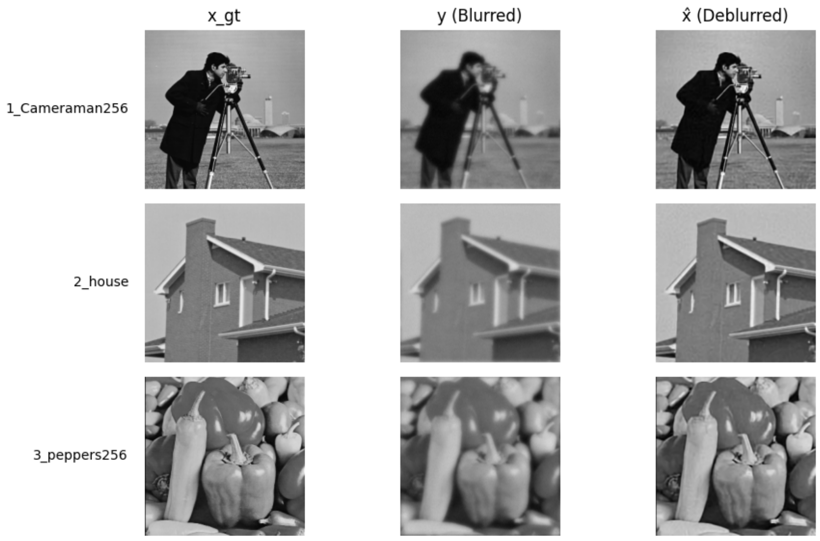

Figure 3: Visual results of the deblurring using BM3D as the denoiser, hyper-parameters: $\sigma = 0.09, \rho = 0.013$

(c) In this part we were asked to find a way to increase performance with hyper-parameter manipulation. There are two ways of doing that: 1) reducing the denoiser's noise level in each iteration. 2) increasing the penalty parameter ρ in each iteration. The motivation is that each iteration the image should be getting less noisy - therfore reducing σ will allow the denoiser to work in an appropriate strength level. In the other hand, increasing ρ will increase the effect of the ADMM penalty - while the image gets less and less blurred. Thus, allowing for better results as well. After performing grid search for those parameters we found that the optimal are: $\eta = 0.99, \gamma = 1.01$, where we update the parameters in each iteration (k) as formulated below:

$$\sigma^{(k)} = \eta \cdot \sigma^{(k-1)}$$

$$\rho^{(k)} = \gamma \cdot \rho^{(k-1)}$$

Using this method we managed to increase the output PSNR!

Q3. (a) In this part we will formulate the ADMM-PnP scheme that we will use to inpaint the images. Let $\boldsymbol{M}$ be a binary matrix mask. Thus,the image y (missing pixels) is formulated as:

$$\boldsymbol{y} = \boldsymbol{x} \cdot \boldsymbol{M} \tag{11}$$

We formulate the PnP-ADMM objective as:

$$\min_{\boldsymbol{x},\boldsymbol{v}} \frac{1}{2}\|\boldsymbol{M}\boldsymbol{x} - \boldsymbol{y}\|^2 + \beta s(\boldsymbol{v}) \quad \text{subject to } \boldsymbol{x} = \boldsymbol{v} \tag{12}$$

The augmented Lagrangian is:

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{v}, \boldsymbol{u}) = \frac{1}{2}\|\boldsymbol{M}\boldsymbol{x} - \boldsymbol{y}\|^2 + \beta s(\boldsymbol{v}) + \frac{\rho}{2}\|\boldsymbol{x} - \boldsymbol{v} + \boldsymbol{u}\|^2 - \frac{\rho}{2}\|\boldsymbol{u}\|^2 \tag{13}$$

The ADMM iterations are:

i. **$\boldsymbol{x}$-update:**

$$\boldsymbol{x}^{(k+1)} = arg\min_{x} \frac{1}{2}||\boldsymbol{M}\boldsymbol{x} - \boldsymbol{y}||^2 + \frac{\rho}{2}||\boldsymbol{x} - \boldsymbol{v}^{(k)} + \boldsymbol{u}^{(k)}||^2 \tag{14}$$

ii. **$\boldsymbol{v}$-update (denoising):**

$$\boldsymbol{v}^{(k+1)} = \mathcal{D}_\beta \left(\boldsymbol{x}^{(k+1)} + \boldsymbol{u}^{(k)}, \frac{\beta}{\rho} \right) \tag{15}$$

iii. **$\boldsymbol{u}$-update:**

$$\boldsymbol{u}^{(k+1)} = \boldsymbol{u}^{(k)} + \boldsymbol{x}^{(k+1)} - \boldsymbol{v}^{(k+1)} \tag{16}$$

Updating $\boldsymbol{v}, \boldsymbol{u}$ is self explanatory. But for $\boldsymbol{x}$ we separate the problem into two:

$$\begin{cases} \frac{1}{2}||\boldsymbol{M}\boldsymbol{x} - \boldsymbol{y}||^2 = \frac{1}{2}\sum_i (\boldsymbol{M}_i x_i - y_i)^2 \\ \frac{\rho}{2}||\boldsymbol{x} - \boldsymbol{v}^{(k-1)} + \boldsymbol{u}^{(k-1)}||^2 = \frac{\rho}{2}\sum_i (x_i - v_i^{(k-1)} + u_i^{(k-1)})^2 \end{cases} \tag{17}$$

Since both are translated to element wise sum, we can derive by x_i:

$$\frac{\partial}{\partial x_i} \begin{cases} \frac{1}{2}\sum_i (\boldsymbol{M}_i x_i - y_i)^2 \\ \frac{\rho}{2}\sum_i (x_i - v_i^{(k-1)} + u_i^{(k-1)})^2 \end{cases} = \begin{cases} \boldsymbol{M}_i x_i - y_i \\ \rho(x_i - v_i^{(k-1)} + u_i^{(k-1)}) \end{cases} \tag{18}$$

We sum both problems and solve the equation:

$$\frac{\partial}{\partial x_i}\frac{1}{2}||\boldsymbol{M}\boldsymbol{x} - \boldsymbol{y}||^2 + \frac{\rho}{2}||\boldsymbol{x} - \boldsymbol{v}^{(k)} + \boldsymbol{u}^{(k)}||^2 = 0 \tag{19}$$

The solution is:

$$x_i^{(k)} = \begin{cases} \frac{y_i + \rho(v_i^{(k-1)} - u_i^{(k-1)})}{1+\rho}, \boldsymbol{M}_i = 1 \\ v_i^{(k-1)} - u_i^{(k-1)}, \boldsymbol{M}_i = 0 \end{cases} \tag{20}$$

Q4. We optimized for the best hyper parameters and got the best resaults for $\rho = 0.2, \beta = 0.006$. Here,$\beta$ controls the denoisers strength. Similarly to the previous question, we added γ and η as factors for ρ and β, respectivly. The best values for the factoring coefficients are:$\gamma = 1.001, \eta = 0.999$.

The results are at Table 3,there are slight improvment when using factoring coeffieiants that are not equal to 1. Figure 4 has the visual samples of the inpainted images. Code in Appendix 3.

Picture	**Input PSNR**	**Reconstructed** -$\{\gamma, \eta{=}1\}$	**Reconstructed** -$\gamma = 1.001, \eta = 0.999$
1_Cameraman256	6.55	24.47	24.51
2_house	5.85	30.50	30.64
3_peppers256	6.55	26.31	26.37
4_Lena512	6.65	29.84	29.89
5_barbara	6.85	27.39	27.44
6_boat	6.31	26.60	26.67
7_hill	7.33	26.88	26.53
8_couple	6.90	26.65	26.70
Average	**6.62**	**27.33**	**27.35**

Table 3: A summary of PSNR [dB] results for inpainting using the BM3D as the denosier.

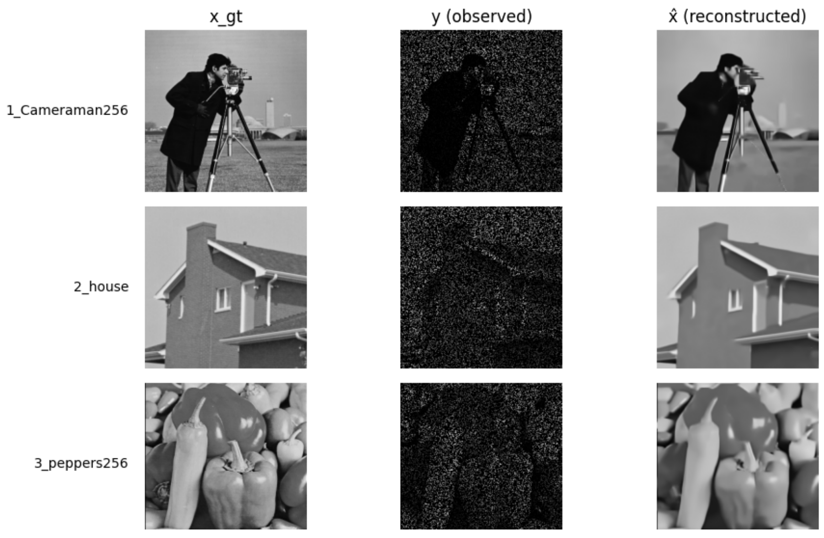

Figure 4: Visual results of the inpainting using BM3D as the denoiser, hyper-parameters: $\rho = 0.2, \beta = 0.006$

Appendix 1: Code for image denoising

```
import numpy as np
import matplotlib.pyplot as plt
from bm3d import bm3d

### ============================
###        Functions
### ============================

np.random.seed(0)

def psnr(x, x_gt):
    """
    Compute the Peak Signal-to-Noise Ratio (PSNR) between two images.

    Inputs:
        x      : Estimated image (numpy array, float32, values in [0, 1])
        x_gt   : Ground truth image (same size and type as x)

    Output:
        PSNR value in decibels (float)
    """
    mse = np.mean((x - x_gt) ** 2)
    return 10 * np.log10(1 / mse)

def add_noise(img, sigma_e):
    """
    Add Gaussian noise to an image with fixed random seed for reproducibility.

    Inputs:
        img      : Clean image (numpy array, float32, values in [0, 1])
        sigma_e : Standard deviation of Gaussian noise

    Output:
        Noisy image y = x_gt + e (numpy array, float32)
    """
    noise = np.random.normal(0, sigma_e, img.shape)
    return img + noise

def Compute_spatial_weights(half_w, sigma_s):
    """
    Precompute the spatial (distance-based) Gaussian weights for the bilateral filter.

    Inputs:
        half_w   : Half of the filter window size (int)
        sigma_s : Standard deviation of the spatial Gaussian kernel (float)

    Output:
        spatial_weights : 2D numpy array of shape (2*half_w+1, 2*half_w+1)
                          containing spatial Gaussian weights centered at (0,0)
    """
    x = np.arange(-half_w, half_w + 1)
    y = np.arange(-half_w, half_w + 1)
    X, Y = np.meshgrid(x, y)
    spatial_weights = np.exp(-(X ** 2 + Y ** 2) / (2 * sigma_s ** 2))
    return spatial_weights

def bilateral_filter(img, sigma_s, sigma_r):
    """
    Apply bilateral filtering to a grayscale image.

    Inputs:
        img       : Noisy grayscale image (numpy array, float32, values in [0, 1])
        sigma_s   : Standard deviation for spatial kernel
        sigma_r   : Standard deviation for range kernel

    Output:
        Denoised image (numpy array, float32, same shape as img)
    """
    H, W = img.shape
    half_w = int(3 * sigma_s)  # Size of the window is 6*sigma_s + 1
    padded_img = np.pad(img, half_w, mode='reflect')
    output = np.zeros_like(img)

    # Compute spatial weights (Gaussian based on distance)
    spatial_weights = Compute_spatial_weights(half_w, sigma_s)

    # Iterate over each pixel
    for i in range(H):
        for j in range(W):
            i1 = i + half_w
            j1 = j + half_w

            # Extract local patch
            patch = padded_img[i1 - half_w:i1 + half_w + 1, j1 - half_w:j1 + half_w + 1]

            # Compute range weights (Gaussian based on intensity difference)
            intensity_diff = patch - padded_img[i1, j1]
            range_weights = np.exp(-(intensity_diff ** 2) / (2 * sigma_r ** 2))

            # Combine weights
            weights = spatial_weights * range_weights
            weights /= np.sum(weights) # Normalizing

            # Apply weighted average
            output[i, j] = np.sum(patch * weights)

    return output

### ============================
###            Main
### ============================

def main():
    """
    Denoise a set of grayscale images using bilateral filtering,
    compute and print PSNR values, and display visual comparisons.
    """
    image_names = ['1_Cameraman256', '2_house', '3_peppers256', '4_Lena512',
                   '5_barbara', '6_boat', '7_hill', '8_couple']
    sigma_e = 0.1  # Noise standard deviation
    sigma_s = 1.7    # Spatial kernel std (tuning)
    sigma_r = 0.26  # Range kernel std (tuning)

    input_psnrs = []
    denoised_psnrs = []
    bm3d_psnrs = []
    # Store results for visualization later
    images_gt = []
    images_noisy = []
    images_denoised = []
    images_bm3d_denoised = []

    dir_path = './test_set'
    for name in image_names:
        try:
            img = plt.imread(f'{dir_path}/{name}.png')
        except FileNotFoundError:
            print(f"Error: File {dir_path}+/{name}.png not found.")
            return -1
        except Exception as e:
            print(f"Could not load {name}.png due to error: {e}")
            return -1

        # Convert to grayscale and normalize
        if img.ndim == 3:
            img = np.mean(img, axis=2)  # convert RGB to grayscale
        if img.dtype != np.float32 and img.max() > 1.0:
            img = img.astype(np.float32) / 255.0  # normalize to [0, 1]

        y = add_noise(img, sigma_e)
        x_hat = bilateral_filter(y, sigma_s, sigma_r)
        x_bm3d_hat = bm3d(y, sigma_psd=sigma_e)

        psnr_input = psnr(img, y)
        psnr_output = psnr(img, x_hat)
        psnr_bm3d_output= psnr(img, x_bm3d_hat)

        input_psnrs.append(psnr_input)
        denoised_psnrs.append(psnr_output)
        bm3d_psnrs.append(psnr_bm3d_output)

        print(f"{name}: Input PSNR = {psnr_input:.2f}, Bilateral Output PSNR = {
                                                    psnr_output:.2f}, Output BM3D PSNR =
                                                     {psnr_bm3d_output:.2f}")

        # Save for visualization
        images_gt.append(img)
        images_noisy.append(y)
        images_denoised.append(x_hat)
        images_bm3d_denoised.append(x_bm3d_hat)

    print("\nAverage Input PSNR: {:.2f}".format(np.mean(input_psnrs)))
    print("Average Bilateral Output PSNR: {:.2f}".format(np.mean(denoised_psnrs)))
    print("Average BM3D Output PSNR: {:.2f}".format(np.mean(bm3d_psnrs)))

    # ===============================
    #  Plot: x_gt, y (noisy), x  (denoised)
    # ===============================
    fig, axs = plt.subplots(len(image_names), 4, figsize=(10, 2 * len(image_names)))
    fig.suptitle('Denoising Results: x_gt, y, x  ,x _BM3D', fontsize=16)

    for row_idx, name in enumerate(image_names):
        img = images_gt[row_idx]  # Ground truth
        y = images_noisy[row_idx]  # Noisy image
        x_hat = images_denoised[row_idx]  # Denoised output
        x_bm3d_hat = images_bm3d_denoised[row_idx]
        for col_idx, image in enumerate([img, y, x_hat,x_bm3d_hat]):
            axs[row_idx, col_idx].imshow(image, cmap='gray', vmin=0, vmax=1)
            axs[row_idx, col_idx].axis('off')

            # Set column titles only on first row
            if row_idx == 0:
                if col_idx == 0:
                    axs[row_idx, col_idx].set_title("x_gt")
                elif col_idx == 1:
                    axs[row_idx, col_idx].set_title("y (noisy)")
                elif col_idx ==2 :
                    axs[row_idx, col_idx].set_title(" x  (denoised)")
                else:
                    axs[row_idx, col_idx].set_title(" x _BM3D (denoised)")


        # Add image name label on the left of each row
        axs[row_idx, 0].text(-0.1, 0.5, name, fontsize=10, va='center', ha='right',
                             transform=axs[row_idx, 0].transAxes, rotation=0)

    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.savefig(f'./plots/denoising_results.png')

    plt.show()



# Run the main script
if __name__ == "__main__":
    main()
```

Appendix 2: Code for image deblurring

```
    import numpy as np
from bm3d import bm3d
from runme_denoising import bilateral_filter,psnr,add_noise
import matplotlib.pyplot as plt
from tqdm import tqdm

def cconv2_invAAt_by_fft2_numpy(A,B,eta=0.01):
    # assumes that A (2D image) is bigger than B (2D kernel)
    m, n = A.shape
    mb, nb = B.shape
    # Pad kernel to image size
    bigB = np.zeros_like(A)
    bigB[:mb, :nb] = B
    # Roll to center the PSF
    bigB = np.roll(bigB, shift=(-mb // 2, -nb // 2), axis=(0, 1))
    # FFT of kernel and input
    fft2B = np.fft.fft2(bigB)
    fft2B_norm2 = np.abs(fft2B)**2
    inv_fft2B_norm = 1 / (fft2B_norm2 + eta)
    result = np.real(np.fft.ifft2(np.fft.fft2(A) * inv_fft2B_norm))
    return result



# y_k = AtA_add_eta_inv(At(y) + rho(x -u))

def cconv2_by_fft2_numpy(A, B,flag_conjB=False, eta=1e-2):
    """
    Circular 2D convolution or deconvolution using FFT (NumPy version).

    Args:
        A (np.ndarray): 2D input image (H x W)
        B (np.ndarray): 2D kernel (h x w)
        flag_invertB (bool): If True, performs deconvolution
        eta (float): Regularization parameter for deconvolution

    Returns:
        np.ndarray: Output after convolution or deconvolution
    """
    m, n = A.shape
    mb, nb = B.shape

    # Pad kernel to image size
    bigB = np.zeros_like(A)
    bigB[:mb, :nb] = B

    # Roll to center the PSF
    bigB = np.roll(bigB, shift=(-mb // 2, -nb // 2), axis=(0, 1))

    # FFT of kernel and input
    fft2B = np.fft.fft2(bigB)
    fft2A = np.fft.fft2(A)

    if flag_conjB:
        # Tikhonov regularization for inverse filtering
        fft2B = np.conj(fft2B)# / (np.abs(fft2B)**2 + eta)

    result = np.real(np.fft.ifft2(fft2A * fft2B))

    return result

class PnPADMMDeBlurr:
    def __init__(self,denoiser,max_iter,rho,sigmas,kernel,gamma=1,eta=1,tol=1e-6):
        '''
        denosiser (string): denosing function
        max_iter (int): maximum ADMM itterations
        rho(float): ADMM penalty parameter
        sigmas (list): [sigma_psd] for bm3d denoiser and [sigma_r,sigma_s] for BF
                                                                  denosier
        '''
        self.kernel = kernel
        self.denoiser = denoiser
        self.max_iter = max_iter
        self.rho = rho
        self.sigmas = sigmas
        self.gamma = gamma
        self.eta=eta
        self.tol = tol
    def get_txt(self):
        return f'rho_{self.rho}_sigma_{self.sigmas},eta_{self.eta}_gamma_{self.gamma}'
    def denoise_sample(self,y):
        if self.denoiser =='bm3d':
            return bm3d(y,sigma_psd=self.sigmas[0])
        else: #blf
            sigma_s = self.sigmas[0]
            sigma_r = self.sigmas[-1]
            return bilateral_filter(y,sigma_s,sigma_r)

    def AtA_add_eta_inv_numpy(self, vec,):
        I = vec.reshape(vec.shape[0], vec.shape[1])

        out = cconv2_invAAt_by_fft2_numpy(I, self.kernel, eta=self.rho)

        return out.reshape(vec.shape[0], -1)

    def At_numpy(self,vec):
        I = vec.reshape(vec.shape[0], vec.shape[1])
        out = cconv2_by_fft2_numpy(I,self.kernel, flag_conjB=True)
        return out.reshape(vec.shape[0], -1)

    def __call__(self, y,img):
        '''
        y: blurred image (2D numpy array)
        Returns: Deblurred image
        '''
        # Initialization
        N = y.shape[0]*y.shape[1]
        #as in the paper

        x_k = y.copy()
        v_k = y.copy()
        u_k = np.zeros_like(y)

        rho_tmp = self.rho
        sigma_tmp = self.sigmas[0]

        i = 0
        res = 10
        pbar = tqdm(total=self.max_iter,desc='Residuals',leave=False)
        psnr_old = 0
        while (res > self.tol) and i < self.max_iter:
            x_k_1,v_k_1,u_k_1 = self.pnp_admm_step(y,x_k,v_k,u_k)
            psnr_mid = psnr(x_k_1,img)
            # print(f'PSNR:{psnr_mid}')
            delta_psnr = psnr_mid - psnr_old
            res_x = (1/np.sqrt(N)) * np.sqrt(np.sum((x_k_1-x_k)**2,axis=(0,1)))
            res_z = (1/np.sqrt(N)) * np.sqrt(np.sum((v_k_1-v_k)**2,axis=(0,1)))
            res_u = (1/np.sqrt(N)) * np.sqrt(np.sum((u_k_1-u_k)**2,axis=(0,1)))

            res = res_u+res_x+res_z
            # if res < 0 and i>10:
            #     break
            if delta_psnr >0:
                self.rho *= self.gamma
            elif delta_psnr<0 and i > 5:
                break
            v_k=v_k_1
            x_k=x_k_1
            u_k=u_k_1
            psnr_old = psnr_mid
            # self.rho*=1.5
            self.sigmas[0] *=self.eta
            i+=1
            pbar.update(1)
            pbar.set_description(f'PSNR={psnr_mid:.8f}')

        self.rho = rho_tmp
        self.sigmas[0] = sigma_tmp
        return x_k

    def prox(self,y,x_tilde):
        a = self.At_numpy(y) + self.rho*(x_tilde)
        return self.AtA_add_eta_inv_numpy(a)

    def pnp_admm_step(self,y,x,v,u):
        x_tilde = v-u
        x = self.prox(y,x_tilde)
        v_tilde = x+u
        v= self.denoise_sample(v_tilde)
        u += x-v
        return x,v,u

def main(denoiser='bm3d'):
    """
    Denoise a set of grayscale images using bilateral filtering,
    compute and print PSNR values, and display visual comparisons.
    """
    image_names = ['1_Cameraman256', '2_house', '3_peppers256', '4_Lena512',
                   '5_barbara', '6_boat', '7_hill', '8_couple']
    #hyper parameters

    max_iter=25
    sigmas = [0.09] if denoiser =='bm3d' else [0.5,0.08]
    rho = 0.013 if denoiser == 'bm3d' else 0.2

    eta = 0.99 if denoiser=='bm3d' else 1
    gamma=1.01 if denoiser=='bm3d' else 1

    #blurring kernel

    i = np.arange(-7, 8)
    j = np.arange(-7, 8)
    kernel = np.zeros((len(i),len(j)))
    for ii in range(len(i)):
        for jj in range(len(j)):
            kernel[ii,jj] = 1/(1+i[ii]**2+j[jj]**2)
    kernel /= np.sum(kernel)

    input_psnrs = []
    denoised_psnrs = []
    images_gt = []
    images_noisy = []
    images_denoised = []

    dir_path = './test_set'
    deblurrer = PnPADMMDeBlurr(denoiser=denoiser,max_iter=max_iter,rho=rho,sigmas=sigmas
                                                ,kernel=kernel,eta=eta,gamma=gamma,tol=
                                                1e-5)

    for name in image_names:
        try:
            img = plt.imread(f'{dir_path}/{name}.png')
        except FileNotFoundError:
            print(f"Error: File {dir_path}+/{name}.png not found.")
            return -1
        except Exception as e:
            print(f"Could not load {name}.png due to error: {e}")
            return -1

        # Convert to grayscale and normalize
        if img.ndim == 3:
            img = np.mean(img, axis=2)  # convert RGB to grayscale
        if img.dtype != np.float32 and img.max() > 1.0:
            img = img.astype(np.float32) / 255.0  # normalize to [0, 1]

        y = cconv2_by_fft2_numpy(img, kernel)
        y = add_noise(y,sigma_e=0.01)
        x_hat = deblurrer(y,img)
        psnr_input = psnr(y, img)
        psnr_output = psnr(x_hat,img)

        input_psnrs.append(psnr_input)
        denoised_psnrs.append(psnr_output)

        print(f"{name}: Input PSNR = {psnr_input:.2f}, Output PSNR = {psnr_output:.2f}")

        # Save for visualization
        images_gt.append(img)
        images_noisy.append(y)
        images_denoised.append(x_hat)

    input_psnr_mean =np.mean(input_psnrs)
    output_psnr_mean =np.mean(denoised_psnrs)

    print("\nAverage Input PSNR: {:.2f}".format(input_psnr_mean))
    print("Average Deblurred PSNR: {:.2f}".format(output_psnr_mean))

    if denoiser =='bm3d':
        sigma_txt = f'sigma={deblurrer.sigmas[0]:.4f}'
    else:
        sigma_txt = f'sigma_s={deblurrer.sigmas[0]:.4f},sigma_r={deblurrer.sigmas[-1]:.
                                                    4f}'

    hyperparams = f'{sigma_txt},rho={rho},eta={eta},gamma={gamma}'
    # ===============================
    #  Plot: x_gt, y (noisy), x   (denoised)
    # ===============================
    fig, axs = plt.subplots(len(image_names), 3, figsize=(10, 2 * len(image_names)))
    fig.suptitle(f'Deblurring Results:PSNR-Input={input_psnr_mean:.2f},PSNR-Output={
                                              output_psnr_mean:.2f}\nHyperparams:{
                                              hyperparams}', fontsize=16)

    for row_idx, name in enumerate(image_names):
        img = images_gt[row_idx]  # Ground truth
        y = images_noisy[row_idx]  # Noisy image
        x_hat = images_denoised[row_idx]  # Denoised output
        for col_idx, image in enumerate([img, y, x_hat]):
            axs[row_idx, col_idx].imshow(image, cmap='gray', vmin=0, vmax=1)
            axs[row_idx, col_idx].axis('off')

            # Set column titles only on first row
            if row_idx == 0:
                if col_idx == 0:
                    axs[row_idx, col_idx].set_title("x_gt")
                elif col_idx == 1:
                    axs[row_idx, col_idx].set_title("y (Blurred)")
                elif col_idx ==2 :
                    axs[row_idx, col_idx].set_title(" x  (Deblurred)")

        # Add image name label on the left of each row
        axs[row_idx, 0].text(-0.1, 0.5, name, fontsize=10, va='center', ha='right',
                             transform=axs[row_idx, 0].transAxes, rotation=0)

    if deblurrer.denoiser!='bm3d':
        rho_fixed_txt=''
        sigma_fixed_txt=''
    plt.tight_layout(rect=[0, 0, 1, 0.96])
    filename = f'./plots/pnp_admm_results_max_{deblurrer.get_txt()}.png'
    plt.savefig(filename)
    print(f'Saved: {filename}')
    plt.show()

if __name__ == "__main__":
    # main(denoiser='BL') #for bilateral filter denoiser
    main(denoiser='bm3d')
```

Appendix 3: Code for image inpainting

```python
import numpy as np
import matplotlib.pyplot as plt
from bm3d import bm3d
from tqdm import tqdm
np.random.seed(0)

def psnr(x, x_gt):
    mse = np.mean((x - x_gt) ** 2)
    return 10 * np.log10(1.0 / mse)

def create_mask(shape, percent):
    """
    Randomly sample a binary mask with `percent` percent of pixels observed.
    """
    total_pixels = np.prod(shape)
    num_samples = int(total_pixels * percent)
    mask = np.zeros(total_pixels, dtype=bool)
    mask[np.random.choice(total_pixels, num_samples, replace=False)] = True
    return mask.reshape(shape)

def inpainting_admm_pnp(y, mask, denoiser, rho=0.8, beta=0.01, gamma=1,eta=1,num_iters=
                                                        25):
    """
    ADMM-PnP for image inpainting using a given denoiser.
    Inputs:
        y         : Observed image with only known pixels in mask.
        mask      : Boolean mask of known pixels.
        denoiser  : Function to apply as plug-and-play denoiser.
        rho       : ADMM penalty parameter.
        beta      : Denoiser strength parameter.
        num_iters: Number of ADMM iterations.
    Output:
        x_hat     : Reconstructed image.
    """
    x = np.copy(y)
    v = np.copy(x)
    u = np.zeros_like(x)

    for k in tqdm(range(num_iters)):

        x = np.where(mask,
                     (y + rho * (v - u)) / (1 + rho),
                     v - u)

        v_tilde = x+u

        v= denoiser(v_tilde,sigma_psd=np.sqrt(beta / rho))

        u += x-v
        rho *=gamma
        beta*=eta**2

    return x

def main():
    image_names = ['1_Cameraman256', '2_house', '3_peppers256', '4_Lena512',
                   '5_barbara', '6_boat', '7_hill', '8_couple']
    # image_names = ['1_Cameraman256', '2_house']
    observed_fraction = 0.2
    rho = 0.2
    beta = 0.006
    gamma=1.001
    eta=0.999
    num_iters = 25

    input_psnrs = []
    recon_psnrs = []
    images_gt = []
    images_observed = []
    images_reconstructed = []
    for name in image_names:
        try:
            img = plt.imread(f'test_set/{name}.png')
        except FileNotFoundError:
            print(f"Error: File {name}.png not found.")
            return -1

        if img.ndim == 3:
            img = np.mean(img, axis=2)
        if img.dtype != np.float32 and img.max() > 1.0:
            img = img.astype(np.float32) / 255.0

        mask = create_mask(img.shape, observed_fraction)
        y = img * mask  # observed pixels only

        x_hat = inpainting_admm_pnp(y=y, mask=mask,denoiser= bm3d, rho=rho, beta=beta,
                                                    num_iters=num_iters,gamma=gamma,eta=
                                                    eta)

        psnr_input = psnr(img, y)
        psnr_output = psnr(img, x_hat)

        input_psnrs.append(psnr_input)
        recon_psnrs.append(psnr_output)
        images_gt.append(img)
        images_observed.append(y)
        images_reconstructed.append(x_hat)

        print(f"{name}: Input PSNR = {psnr_input:.2f}, Reconstructed PSNR = {psnr_output
                                                    :.2f}")

    print("\nAverage Input PSNR: {:.2f}".format(np.mean(input_psnrs)))
    print("Average Reconstructed PSNR: {:.2f}".format(np.mean(recon_psnrs)))

    # Display images
    fig, axs = plt.subplots(len(image_names), 3, figsize=(10, 2 * len(image_names)))
    fig.suptitle('Inpainting Results: x_gt, y, x  ', fontsize=16)

    for row_idx, name in enumerate(image_names):
        for col_idx, image in enumerate([images_gt[row_idx], images_observed[row_idx],
                                                    images_reconstructed[row_idx]]):
            axs[row_idx, col_idx].imshow(image, cmap='gray', vmin=0, vmax=1)
            axs[row_idx, col_idx].axis('off')
            if row_idx == 0:
                axs[row_idx, col_idx].set_title(["x_gt", "y (observed)", " x   (
                                                            reconstructed)"][col_idx])

        axs[row_idx, 0].text(-0.1, 0.5, name, fontsize=10, va='center', ha='right',
                             transform=axs[row_idx, 0].transAxes)

    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.savefig('./plots/pnp_admm_inpainting.png')
    plt.show()

if __name__ == "__main__":
    main()
```