Assignment 5

Xuchen Zhu, Jingxue Yan

December 11, 2025

1 Exercise 1

1.1 Q1:

1.1.1 Derivation of $\mu|y,\tau$:

$$p(\mu|y,\tau) \propto p(y|\mu,\tau)p(\mu)$$

Expanding the Gaussian densities:

$$\begin{aligned} p(\mu|y,\tau) &\propto \left[\exp\left(-\frac{\tau}{2}\sum_{i=1}^{n}(y_i-\mu)^2\right)\right]\cdot\left[\exp\left(-\frac{\omega}{2}\mu^2\right)\right] \\ &\propto \exp\left(-\frac{1}{2}\left[\tau\sum(y_i^2-2y_i\mu+\mu^2)+\omega\mu^2\right]\right) \end{aligned}$$

We have:

$$p(\mu|y,\tau) \propto \exp\left(-\frac{1}{2}\left[(n\tau+\omega)\mu^2-2(\tau\sum y_i)\mu\right]\right)$$

This is the Gaussian distribution $\mathcal{N}(m,s^{-1})$ with precision $s=n\tau+\omega$ and mean $m=\frac{\tau\sum y_i}{n\tau+\omega}$. Thus:

$$\mu|y,\tau \sim \mathcal{N}\left(\frac{\tau\sum y_i}{n\tau+\omega},\frac{1}{n\tau+\omega}\right)$$

1.1.2 Derivation of $\tau|y,\mu$:

$$p(\tau|y,\mu) \propto p(y|\mu,\tau)p(\tau)$$

Using the PDF of Gamma density and Gaussian likelihood:

$$\begin{aligned} p(\tau|y,\mu) &\propto \left[\tau^{n/2}\exp\left(-\frac{\tau}{2}\sum_{i=1}^{n}(y_i-\mu)^2\right)\right]\cdot\left[\tau^{\alpha-1}\exp(-\beta\tau)\right] \\ &\propto \tau^{\frac{n}{2}+\alpha-1}\exp\left(-\tau\left[\beta+\frac{1}{2}\sum_{i=1}^{n}(y_i-\mu)^2\right]\right) \end{aligned}$$

This matches the Gamma distribution with shape parameter $A=\alpha+\frac{n}{2}$ and rate parameter $B=\beta+\frac{1}{2}\sum(y_i-\mu)^2$. Thus:

$$\tau|y,\mu \sim \text{Gamma}\left(\alpha+\frac{n}{2},\beta+\frac{1}{2}\sum(y_i-\mu)^2\right)$$

1.2 Q2:

1.2.1 Derivation of $\mu|y,\beta$:

$$p(\mu|y,\beta) \propto p(y|\mu)p(\mu|\beta)$$

The likelihood for n observations is $\prod \frac{e^{-\mu}\mu^{y_i}}{y_i!} \propto e^{-n\mu}\mu^{\sum y_i}$. The prior is $\propto \mu^{2-1}e^{-\beta\mu}$.

$$\begin{aligned} p(\mu|y,\beta) &\propto \left(e^{-n\mu}\mu^{\sum y_i}\right)\left(\mu^1 e^{-\beta\mu}\right) \\ &\propto \mu^{(2+\sum y_i)-1}e^{-(n+\beta)\mu} \end{aligned}$$

This is the Gamma distribution with shape $2+\sum y_i$ and rate $n+\beta$. Thus:

$$\mu|y,\beta \sim \text{Gamma}\left(2+\sum_i y_i, n+\beta\right)$$

1.2.2 Derivation of $\beta|y,\mu$:

Since β appears only in the prior for μ, we have:

$$p(\beta|y,\mu) \propto p(\mu|\beta)p(\beta)$$

Using the Gamma normalization constant for $p(\mu|\beta)$ where shape=2 and rate=β:

$$p(\mu|\beta) = \frac{\beta^2}{\Gamma(2)}\mu^{2-1}e^{-\beta\mu} \propto \beta^2 e^{-\beta\mu}$$

Combining with the prior $p(\beta) \propto e^{-\beta}$:

$$\begin{aligned} p(\beta|y,\mu) &\propto (\beta^2 e^{-\mu\beta})(e^{-\beta}) \\ &\propto \beta^{3-1}e^{-(1+\mu)\beta} \end{aligned}$$

This is a Gamma distribution with shape 3 and rate $1+\mu$. Thus:

$$\beta|y,\mu \sim \text{Gamma}(3, 1+\mu)$$

1.3 Q3:

Estimated Mu: 4.8799 (True: 5)

Estimated Variance: 3.8727 (True: 4)

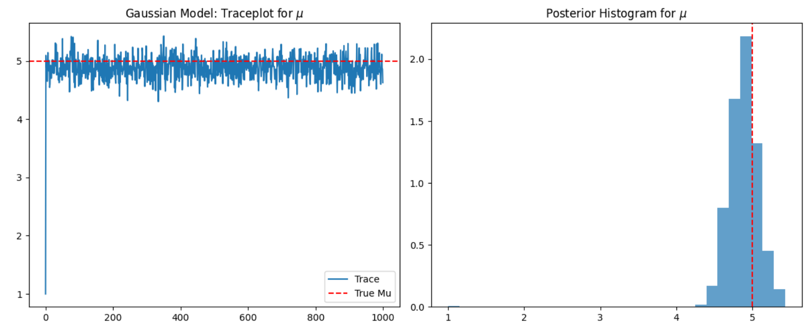

Figure 1: Q3: Traceplots and Histograms for μ

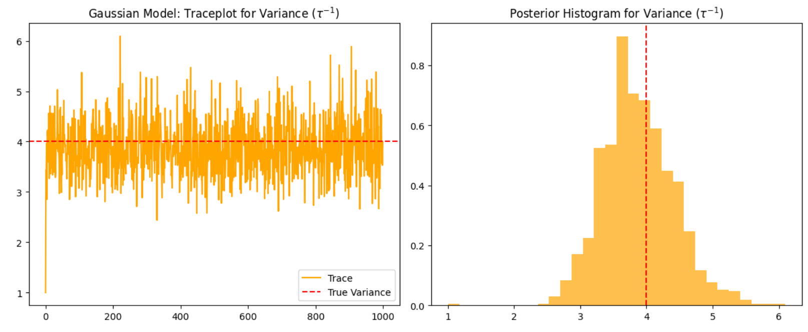

Figure 2: Q3: Traceplots and Histograms for Variance

1.4 Q4:

Poisson Estimated Mu: 4.8993 (True: 5)

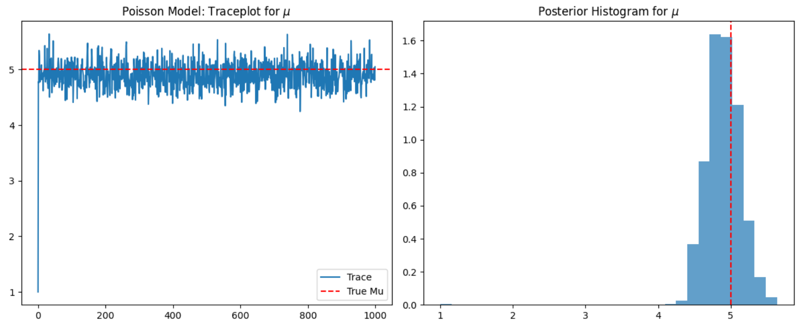

Figure 3: Q4: Traceplots and Histograms for μ

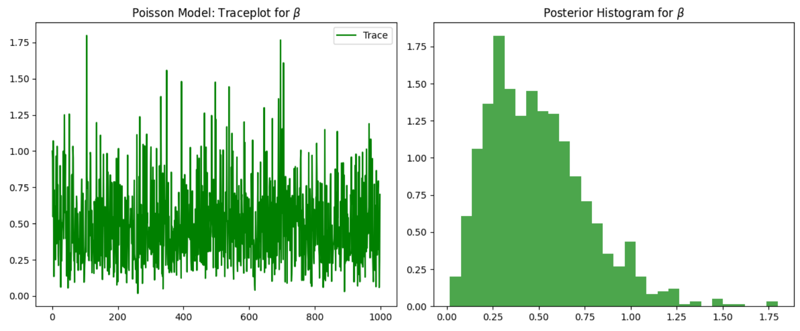

Figure 4: Q4: Traceplots and Histograms for β

2 Exercise 2

2.1 Q1:

As seen in the Fig, it's the iteration result using Gibbs sampling.

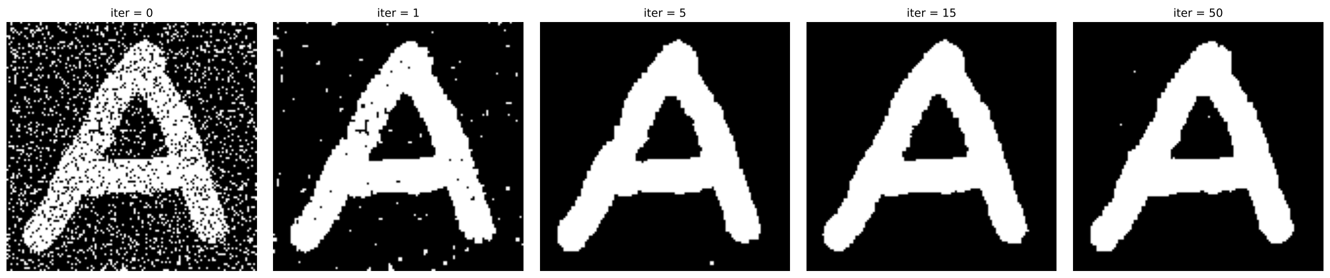

Figure 5: Gibbs sampling.

2.2 Q2:

As seen in the Fig, it's the iteration result using MFA.

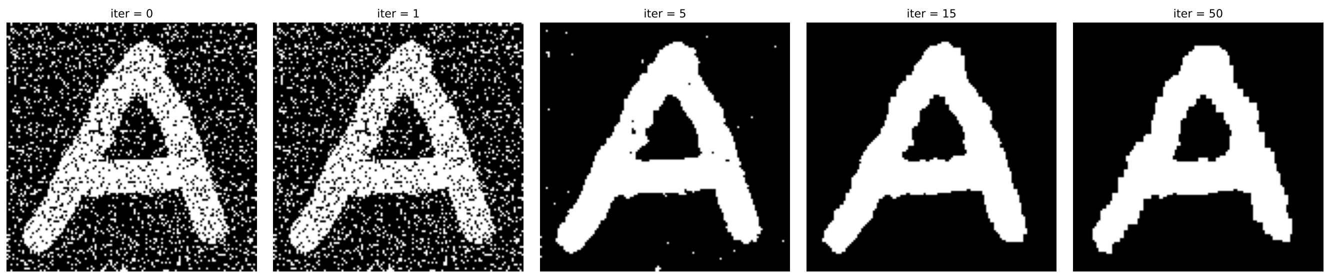

Figure 6: Gibbs sampling.

2.3 Q3:

As seen in the Fig, it's the iteration result using MFA.

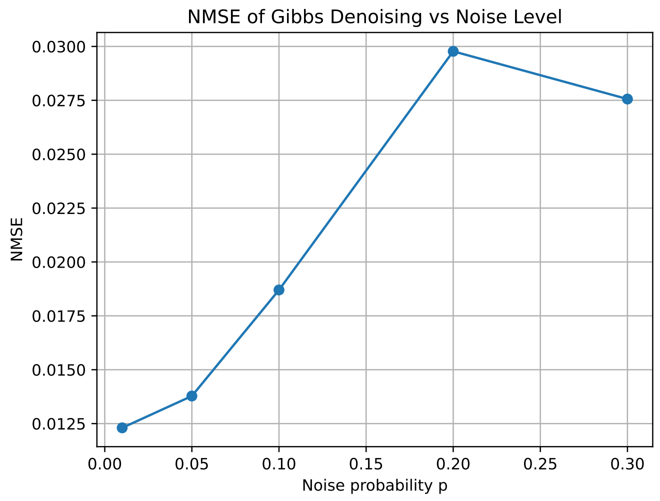

Figure 7: Gibbs: NMSE vs noise level.

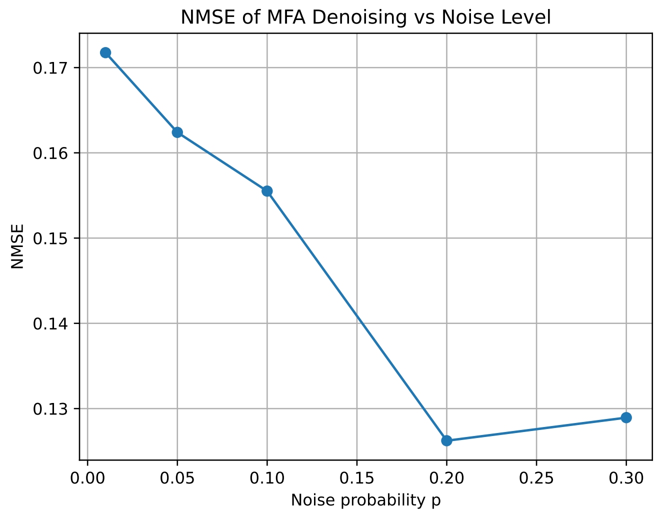

Figure 8: MFA: NMSE vs noise level.

As noise increases, the NMSE of Gibbs increases, because When observation is unreliable → denoiser's error must rise.

As noise increases, the NMSE of MFA decreases, because MFA stops listening to the data and collapses to a very smooth solution.

2.4 Q4:

In terms of convergence, MFA is significantly faster than Gibbs sampling. MFA performs a deterministic fixed-point iteration and typically stabilizes within 20–50 iterations, while Gibbs sampling requires a substantially larger number of iterations to approach the stationary posterior distribution.

The two methods also differ in terms of final NMSE performance. For a fixed number of iterations, MFA consistently achieves lower NMSE values than Gibbs sampling across most noise levels.

Finally, the visual quality of the outputs reflects these structural differences. MFA produces clean, stable, and very smooth images, often yielding the most visually appealing results within a limited iteration budget. Gibbs sampling, on the other hand, generates images that remain grainy or speckled.

2.5 Q5:

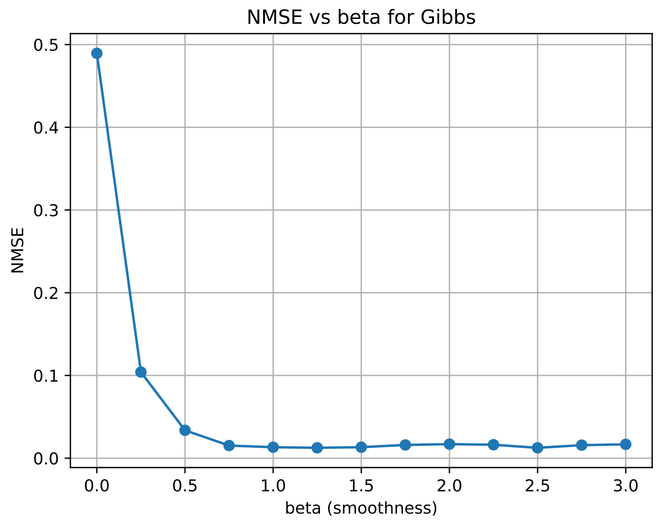

Figure 9: Gibbs sampling vs β.

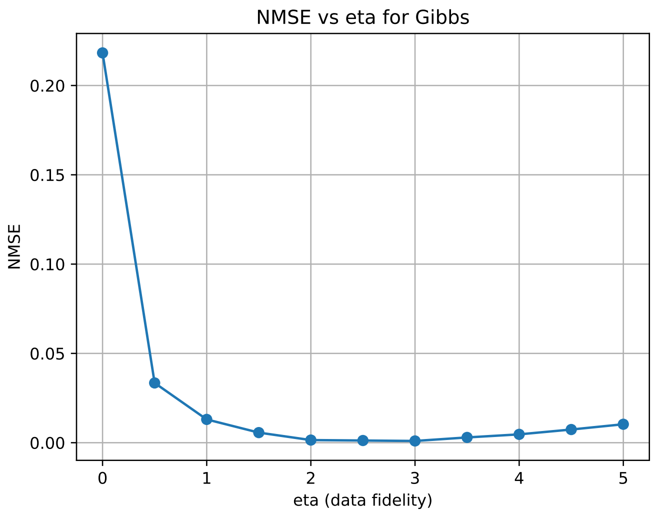

Figure 10: Gibbs sampling vs η.

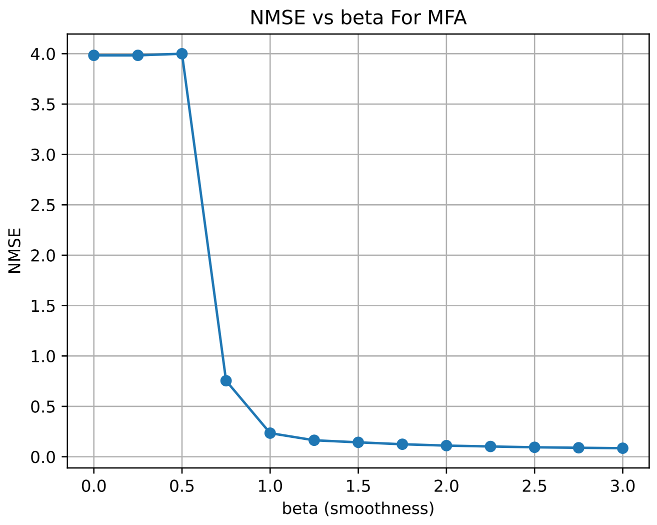

Figure 11: MFA sampling vs β.

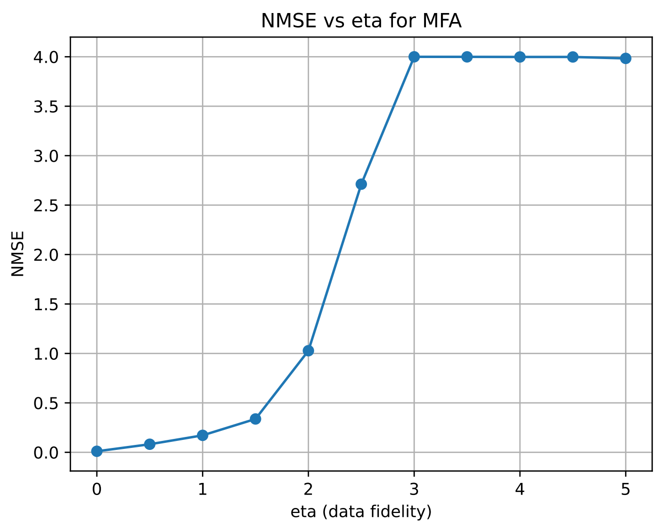

Figure 12: MFA sampling vs η.

The NMSE trends for β and η illustrate a clear balance between smoothness and data fidelity. As β increases, the denoising initially improves because neighboring pixels are encouraged to agree, but excessive β produces oversmoothing and higher NMSE. Similarly, increasing η first improves performance by enforcing adherence to the observed image, but very large η causes the method to follow the noise too closely. These results show that effective denoising requires choosing β and η within a moderate range, where the model neither oversmooths the image nor overfits the noisy observations.

A Source Code for Assignment 5

A.1 Exercise 1

Listing 1: Gaussian Model

```
def gibbs_sampler(initial_values, y, w, alfa, beta, num_posterior_samples=50):
    n = len(y)
    sum_y = np.sum(y)

    # Storage
    mu_samples = np.zeros((num_posterior_samples, 1))
    tau_samples = np.zeros((num_posterior_samples, 1))

    # Initialization
    mu_curr = initial_values[0]
    tau_curr = initial_values[1]

    mu_samples[0] = mu_curr
    tau_samples[0] = tau_curr

    for i in range(1, num_posterior_samples):
        # 1. Sample mu | y, tau
        # Precision s = n*tau + w
        # Mean m = (tau * sum_y) / s
        prec_mu = (n * tau_curr) + w
        mean_mu = (tau_curr * sum_y) / prec_mu
        std_mu = np.sqrt(1.0 / prec_mu)

        mu_curr = np.random.normal(loc=mean_mu, scale=std_mu)
        # 2. Sample tau | y, mu
        # Shape A = alpha + n/2
        # Rate B = beta + 0.5 * sum((y - mu)^2)
        shape_tau = alfa + (n / 2.0)
        rate_tau = beta + 0.5 * np.sum((y - mu_curr)**2)
        scale_tau = 1.0 / rate_tau  # Numpy uses scale = 1/rate

        tau_curr = np.random.gamma(shape=shape_tau, scale=scale_tau)

        # Store
        mu_samples[i] = mu_curr
        tau_samples[i] = tau_curr

    return mu_samples, tau_samples
```

Listing 2: Poisson Model

```
def gibbs_sampler_pois(initial_values, y, num_posterior_samples):
    n = len(y)
    sum_y = np.sum(y)

    # Storage
    mu_samples = np.zeros((num_posterior_samples, 1))
    beta_samples = np.zeros((num_posterior_samples, 1))

    # Initialization
    mu_curr = initial_values[0]
    beta_curr = initial_values[1]

    mu_samples[0] = mu_curr
    beta_samples[0] = beta_curr

    for i in range(1, num_posterior_samples):
        # 1. Sample mu | y, beta
        # Gamma(2 + sum_y, n + beta)
        shape_mu = 2 + sum_y
        rate_mu = n + beta_curr
        scale_mu = 1.0 / rate_mu

        mu_curr = np.random.gamma(shape=shape_mu, scale=scale_mu)

        # 2. Sample beta | y, mu
        # Gamma(3, 1 + mu)
        shape_beta = 3
        rate_beta = 1 + mu_curr
        scale_beta = 1.0 / rate_beta

        beta_curr = np.random.gamma(shape=shape_beta, scale=scale_beta)

        # Store
        mu_samples[i] = mu_curr
        beta_samples[i] = beta_curr

    return mu_samples, beta_samples
```

A.2 Exercise 2

Listing 3: Exercise 2

```
import pandas as pd
img = pd.read_csv('letterA.csv').to_numpy()

import matplotlib.pyplot as plt
import matplotlib.image as mpimg

plt.imshow(img, cmap='gray')  # or cmap='gray' if it's a grayscale image
plt.axis('off')
plt.show()
import numpy as np

def salt_pepper_noise(img, prob):
    noisy = img.copy()
    rnd = np.random.rand(*img.shape)

    noisy[rnd < prob/2] = -1           # pepper
    noisy[rnd > 1 - prob/2] = 1     # salt

    return noisy


noise_1 = salt_pepper_noise(img, 0.01)
noise_2 = salt_pepper_noise(img, 0.05)
noise_3 = salt_pepper_noise(img, 0.1)
noise_4 = salt_pepper_noise(img, 0.2)
noise_5 = salt_pepper_noise(img, 0.3)
noises = [noise_1, noise_2, noise_3, noise_4, noise_5]
noise_levels = [0.01, 0.05, 0.10, 0.20, 0.30]

import matplotlib.pyplot as plt

fig, axs = plt.subplots(1, len(noises), figsize=(4 * len(noises), 4))

for idx, ax in enumerate(axs):
    ax.imshow(noises[idx], cmap='gray', vmin=-1, vmax=1)
    ax.set_title(f"p = {noise_levels[idx]}")
    ax.axis('off')

plt.tight_layout()

plt.show()
def Gibbs_denoise(imag: np.ndarray, h: float, beta: float, ita: float, max_iter: int = 0)
    H, W = imag.shape
    x = imag.copy()

    for _ in range(max_iter):
        for i in range(H):
            for j in range(W):

                # Compute neighbor sum correctly
                neigh = 0
                if i > 0:      neigh += x[i-1, j]
                if i < H-1:    neigh += x[i+1, j]
                if j > 0:      neigh += x[i, j-1]
                if j < W-1:    neigh += x[i, j+1]
                y_ij = imag[i, j]

                # Correct energy formula
                E_pos = (+1) * (h - beta * neigh - ita * y_ij)
                E_neg = (-1) * (h - beta * neigh - ita * y_ij)
                p_pos = np.exp(-E_pos)
                p_neg = np.exp(-E_neg)

                P = p_pos / (p_pos + p_neg)



                u = np.random.uniform(0, 1)
                if u < P:
                    x[i, j] = 1
                else:
                    x[i, j] = -1

    return x


beta = 1.2
ita = 1.0
h = 0

noises = [noise_1, noise_2, noise_3,noise_4,noise_5]
denoises = []
iters = [0,1,5,15,50]

for max_iter in iters:
    denoises.append(Gibbs_denoise(noise_5, h, beta, ita, max_iter))
import matplotlib.pyplot as plt

fig, axs = plt.subplots(1, len(denoises), figsize=(4 * len(denoises), 4))

for idx, ax in enumerate(axs):
    ax.imshow(denoises[idx], cmap='gray')
    ax.set_title(f"iter = {iters[idx]}")
    ax.axis('off')

plt.tight_layout()
#plt.savefig("q3.pdf", format="pdf", bbox_inches="tight")
plt.show()

fig, axs = plt.subplots(1, len(denoises), figsize=(4 * len(denoises), 4))

for idx, ax in enumerate(axs):
    ax.imshow(denoises[idx], cmap='gray')
    ax.set_title(f"iter = {iters[idx]}")
    ax.axis('off')

plt.tight_layout()
plt.savefig("HW5_E2_Q1.pdf", format="pdf", bbox_inches="tight")
plt.show()
```

```
import numpy as np

def MFA_denoise(imag: np.ndarray, h: float, beta: float, ita: float,
                max_iter: int = 20, lamb: float = 0.2) -> np.ndarray:

    H, W = imag.shape
    x = imag.astype(float).copy()   # mean field expectations

    for _ in range(max_iter):
        x_new = x.copy()

        for i in range(H):
            for j in range(W):

                # neighbor sum using old iteration's x
                neigh = 0.0
                if i > 0:      neigh += x[i-1, j]
                if i < H-1:    neigh += x[i+1, j]
                if j > 0:      neigh += x[i, j-1]
                if j < W-1:    neigh += x[i, j+1]

                # local field
                a_i = beta * neigh - h - ita * imag[i, j]

                # mean-field update
                mu_update = np.tanh(a_i)

                # damping update
                x_new[i, j] = (1 - lamb) * x[i, j] +lamb * mu_update

        x = x_new

    return np.sign(x)   # final discrete image




beta = 1.2
ita = 1.0
h = 0

noises = [noise_1, noise_2, noise_3,noise_4,noise_5]
denoises_MFA = []
iters = [0,1,5,15,50]

for max_iter in iters:
    denoises_MFA.append(MFA_denoise(noise_5, h, beta, ita, max_iter))
import matplotlib.pyplot as plt

fig, axs = plt.subplots(1, len(denoises_MFA), figsize=(4 * len(denoises_MFA), 4))

for idx, ax in enumerate(axs):
    ax.imshow(denoises_MFA[idx], cmap='gray')
    ax.set_title(f"iter = {iters[idx]}")
    ax.axis('off')
```

```
plt.tight_layout()
#plt.savefig("q3.pdf", format="pdf", bbox_inches="tight")
plt.show()

fig, axs = plt.subplots(1, len(denoises_MFA), figsize=(4 * len(denoises_MFA), 4))

for idx, ax in enumerate(axs):
    ax.imshow(denoises_MFA[idx], cmap='gray')
    ax.set_title(f"iter = {iters[idx]}")
    ax.axis('off')

plt.tight_layout()
plt.savefig("HW5_E2_Q2.pdf", format="pdf", bbox_inches="tight")
plt.show()

Gibbs_Q3 = []
for noise in noises:
    Gibbs_Q3.append(Gibbs_denoise(noise, h, beta, ita, max_iter))
ps = [0.01, 0.05, 0.10, 0.20, 0.30]

nmse_list = [
    np.sum((a - img)**2) / np.sum(img**2)
    for a in Gibbs_Q3
]
plt.plot(ps, nmse_list, marker='o')
plt.xlabel("Noise probability p")
plt.ylabel("NMSE")
plt.title("NMSE of Gibbs Denoising vs Noise Level")
plt.grid(True)
plt.savefig("HW5_Gibbs_Q3.pdf", format="pdf", bbox_inches="tight")
plt.show()

MFA_Q3 = []
for noise in noises:
    MFA_Q3.append(MFA_denoise(noise, h, beta, ita, max_iter))
ps = [0.01, 0.05, 0.10, 0.20, 0.30]

nmse_list = [
    np.sum((a - img)**2) / np.sum(img**2)
    for a in MFA_Q3
]
plt.plot(ps, nmse_list, marker='o')
plt.xlabel("Noise probability p")
plt.ylabel("NMSE")
plt.title("NMSE of MFA Denoising vs Noise Level")
plt.grid(True)
plt.savefig("HW5_MFA_Q3.pdf", format="pdf", bbox_inches="tight")
plt.show()

def nmse(x, xhat):
    return np.sum((x - xhat)**2) / np.sum(x**2)

h_values = np.linspace(-2, 2, 11)
nmse_h = []

for h_test in h_values:
    den = Gibbs_denoise(noise_5, h_test, beta, ita, max_iter=50)
    nmse_h.append(nmse(img, den))

plt.plot(h_values, nmse_h, marker='o')
plt.xlabel("h")
plt.ylabel("NMSE")
plt.title("NMSE vs h for Gibbs")
plt.grid(True)
plt.savefig("Gibbs_q5_1.pdf", format="pdf", bbox_inches="tight")
plt.show()

plt.plot(h_values, nmse_h, marker='o')
plt.xlabel("h")
plt.ylabel("NMSE")
plt.title("NMSE vs h For MFA")
plt.grid(True)
plt.savefig("MFA_q5_1.pdf", format="pdf", bbox_inches="tight")
plt.show()

beta_values = np.linspace(0, 3, 13)
nmse_beta = []

for b in beta_values:
    den = Gibbs_denoise(noise_1, h, b, ita, max_iter=50)
    nmse_beta.append(nmse(img, den))

plt.plot(beta_values, nmse_beta, marker='o')
plt.xlabel("beta (smoothness)")
plt.ylabel("NMSE")
plt.title("NMSE vs beta for Gibbs")
plt.grid(True)
plt.savefig("Gibbs_denoise_q5_2.pdf", format="pdf", bbox_inches="tight")
plt.show()

plt.plot(beta_values, nmse_beta, marker='o')
plt.xlabel("beta (smoothness)")
plt.ylabel("NMSE")
plt.title("NMSE vs beta For MFA")
plt.grid(True)
plt.savefig("MFA_q5_2.pdf", format="pdf", bbox_inches="tight")
plt.show()

eta_values = np.linspace(0, 5, 11)
nmse_eta = []

for e in eta_values:
    den = Gibbs_denoise(noise_1, h, beta, e, max_iter=50)
    nmse_eta.append(nmse(img, den))

plt.plot(eta_values, nmse_eta, marker='o')
plt.xlabel("eta (data fidelity)")
plt.ylabel("NMSE")
plt.title("NMSE vs eta for Gibbs")
plt.grid(True)
plt.savefig("Gibbs_denoise_q5_3.pdf", format="pdf", bbox_inches="tight")
plt.show()
plt.plot(eta_values, nmse_eta, marker='o')
plt.xlabel("eta (data fidelity)")
plt.ylabel("NMSE")
plt.title("NMSE vs eta for MFA")
plt.grid(True)
plt.savefig("MFA_q5_3.pdf", format="pdf", bbox_inches="tight")
plt.show()
```