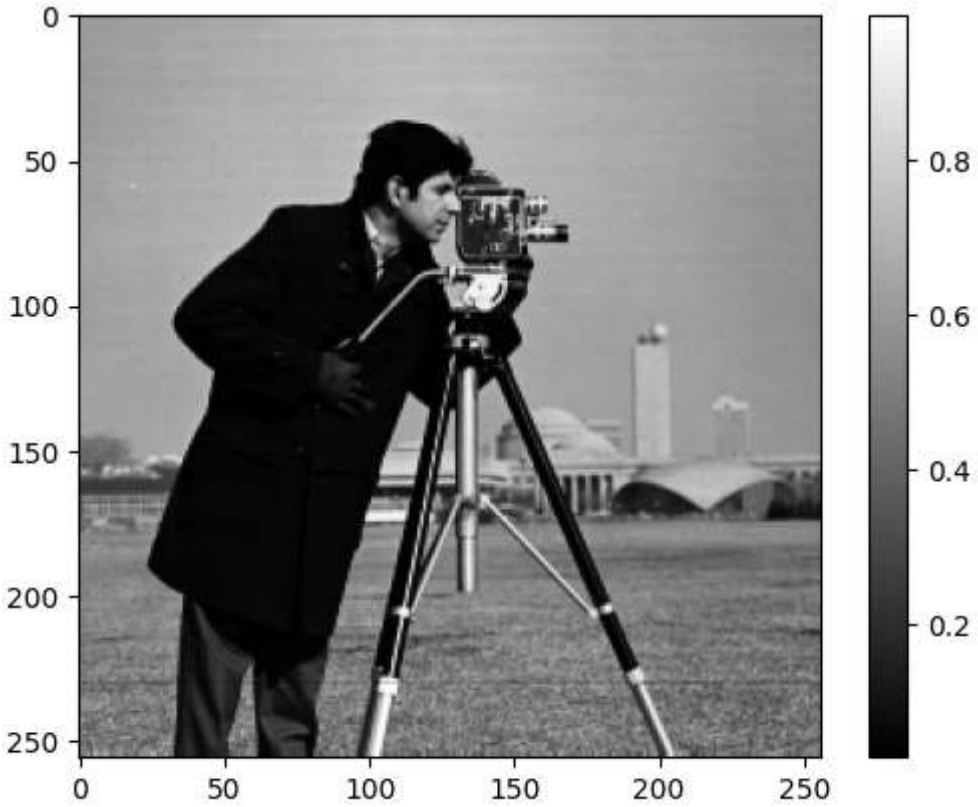
GOMP0114 Inverse Problems in Imaging. Coursework 2

Student ID: 18145399

Week 1

1. Convolution and deconvolution

(a) Read a gray colormap image from the given URL and convert it to a float, normalise and display it.



(b) Write a function that takes in an image f and outputs the blurred image Af with convolution mapping.

$$g = Af_{true} + n$$

In [ ]:
```
import scipy.ndimage.filters as filters

def apply_convolution(f, sigma, theta):
    # Apply Gaussian filter
    g = filters.gaussian_filter(f, sigma)

    # Add noise to blurred image
    w, h = g.shape
    noise = np.random.randn(w, h)
    g = g + theta * noise

    return g
```

(c) Deconvolve using normal equations, i.e. find f_α as the solution to

$$(A^TA + \alpha I)f_\alpha = A^Tg$$

In [ ]:
```
import scipy.sparse as sparse
import scipy.sparse.linalg as splinalg

def ATA_operator(f, sigma, alpha):
    # Apply A^T A + alpha*I operator to f
    Af = filters.gaussian_filter(f, sigma)
    ATAf = filters.gaussian_filter(Af, sigma)
    return ATAf + alpha * f

def deconvolve_normal_equations(g, sigma, alpha):
    # Set up linear operator for ATA
    M, N = g.shape
    A = sparse.linalg.LinearOperator((M*N, M*N), matvec=lambda x: np.ravel(ATA_operator(x.reshape(g.shape), sigma, alpha)))

    # Compute ATg
    ATg = np.ravel(g)
```

```
    # Solve linear system using GMRES
    f_alpha, info = splinalg.gmres(A, ATg)

    return f_alpha.reshape((M, N))

def normal_info(g, sigma, alpha):
    # Set up linear operator for ATA
    M, N = g.shape
    A = sparse.linalg.LinearOperator((M*N, M*N), matvec=lambda x: np.ravel(ATA_operator(x.reshape(g.shape), sigma, alpha)))

    # Compute ATg
    ATg = np.ravel(g)

    # Solve linear system using GMRES
    f_alpha, info = splinalg.gmres(A, ATg)

    return info
```

(d) Deconvolve by solving the augmented equations.

$$\begin{pmatrix} A \\ \sqrt{\alpha} I \end{pmatrix} f = \begin{pmatrix} g \\ 0 \end{pmatrix}$$

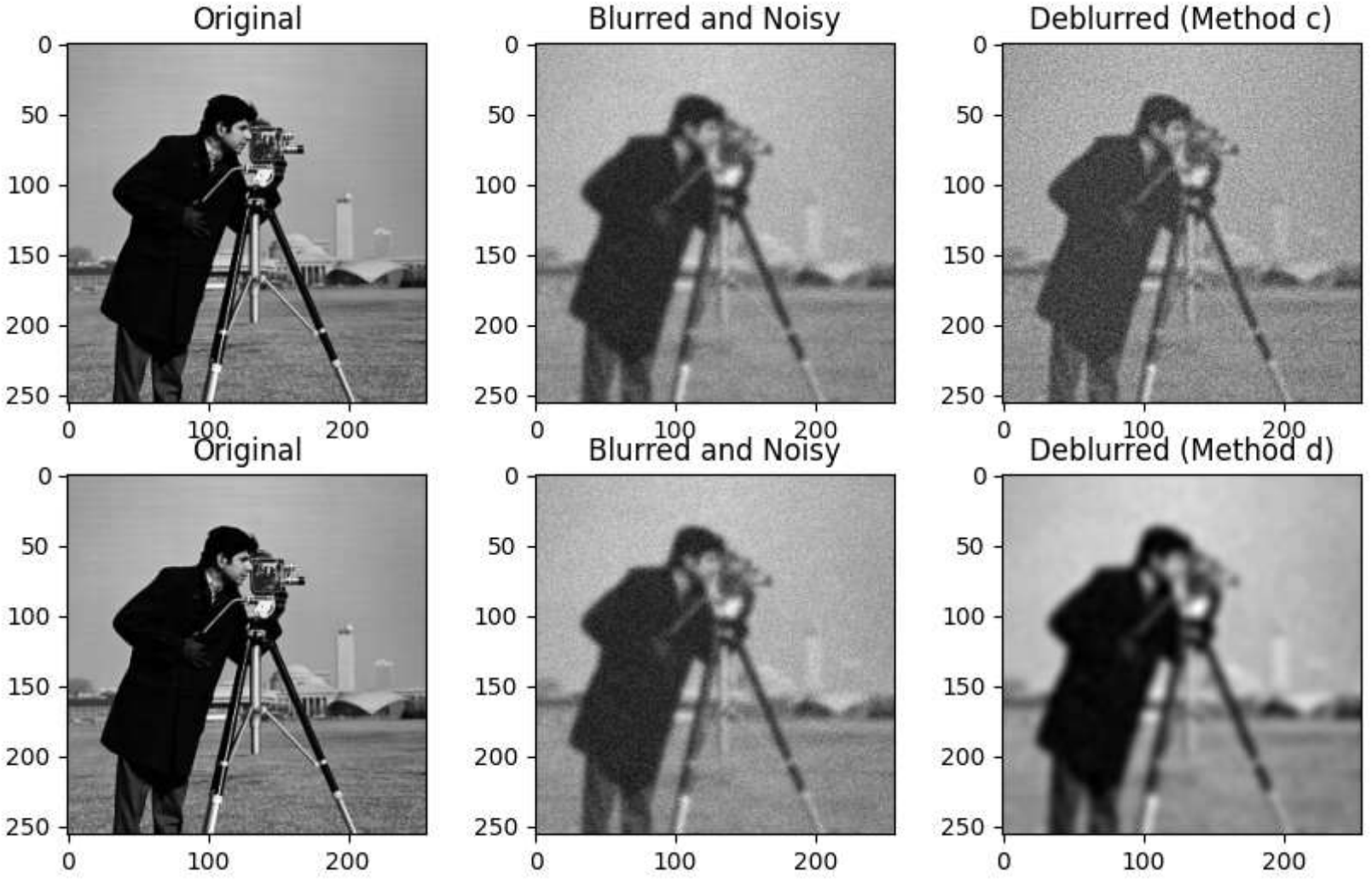
In [ ]:
```
def M_f(f):
    # Implementation of the augmented matrix multiplication
    y = filters.gaussian_filter(f, sigma)
    z = filters.gaussian_filter(y, sigma)
    M_f = np.vstack([np.ravel(z), np.sqrt(alpha)*np.ravel(f)])
    return M_f

def MT_b(b):
    # Implementation of the transposed augmented matrix multiplication
    global g
    M, N = g.shape
    g_vec = b[:M*N]
    f_vec = b[M*N:]
    g = np.reshape(g_vec, (M, N))
    y = filters.gaussian_filter(g, sigma)
```

```
    z = filters.gaussian_filter(y, sigma)
    MT_b = np.ravel(z) + np.sqrt(alpha)*np.ravel(f_vec)
    return MT_b

def solve_augmented_equations(g, sigma, alpha):
    # Define linear operator for lsqr
    M, N = g.shape
    size = M*N
    A = sparse.linalg.LinearOperator((2*size, size), matvec=M_f, rmatvec=MT_b)

    # Concatenate g with a zero vector
    b = np.vstack([np.reshape(g,(size,1)), np.zeros((size, 1))])

    # Solve linear system using lsqr
    f_lsqr= splinalg.lsqr(A, b)[0]

    return f_lsqr[:g.size].reshape(g.shape)

def augmented_info(g, sigma, alpha):
    # Define linear operator for lsqr
    M, N = g.shape
    size = M*N
    A = sparse.linalg.LinearOperator((2*size, size), matvec=M_f, rmatvec=MT_b)

    # Concatenate g with a zero vector
    b = np.vstack([np.reshape(g,(size,1)), np.zeros((size, 1))])

    # Solve linear system using lsqr
    info = splinalg.lsqr(A, b)[1]

    return info
```

Compare the performance to the one used in c.), in terms of number of iterations required to achieve convergence:

Method c: Converged in 0 iterations (0.03956246376037598 seconds)

Method d: Converged in 2 iterations (0.7001798152923584 seconds)



Comments:

The choice of the value of alpha

When the value of the regularization parameter alpha was set to 0.01, the deblurred image produced by Method c was unclear. However, when alpha was set to 1, the deblurred image became much clearer.

It is possible that alpha=0.01 was too small, and the regularization term had little effect in suppressing the noise in the observed image. As a result, the noise dominated the solution, leading to an unclear deblurred image. On the other hand, alpha=1 may have been a better choice, as it struck a good balance between noise suppression and image fidelity, resulting in a clearer deblurred image.

The choice of Deconvolution method

Method c is faster and requires fewer iterations to converge than Method d.

Method c uses the normal equations, which can be solved using a Krylov solver such as PCG or GMRES. The normal equations can be derived from the optimality conditions for the Tikhonov regularization problem, and their solution provides the optimal solution to the deblurring problem. Since the normal equations involve a symmetric positive definite matrix, a Krylov solver can efficiently solve the linear system without requiring an explicit matrix representation of the convolution operator.

On the other hand, Method d uses an augmented equation approach, which involves solving a linear least squares problem using a solver such as lsqr. This method may require more iterations to converge because the least squares problem may not have a closed-form solution and the iterative solver may need to compute many iterations to approximate the solution. In addition, the implementation of the augmented equation method requires more memory to store the augmented system matrix.

Week 2:

2. Choose a regularisation parameter α

i) Discrepency Principle

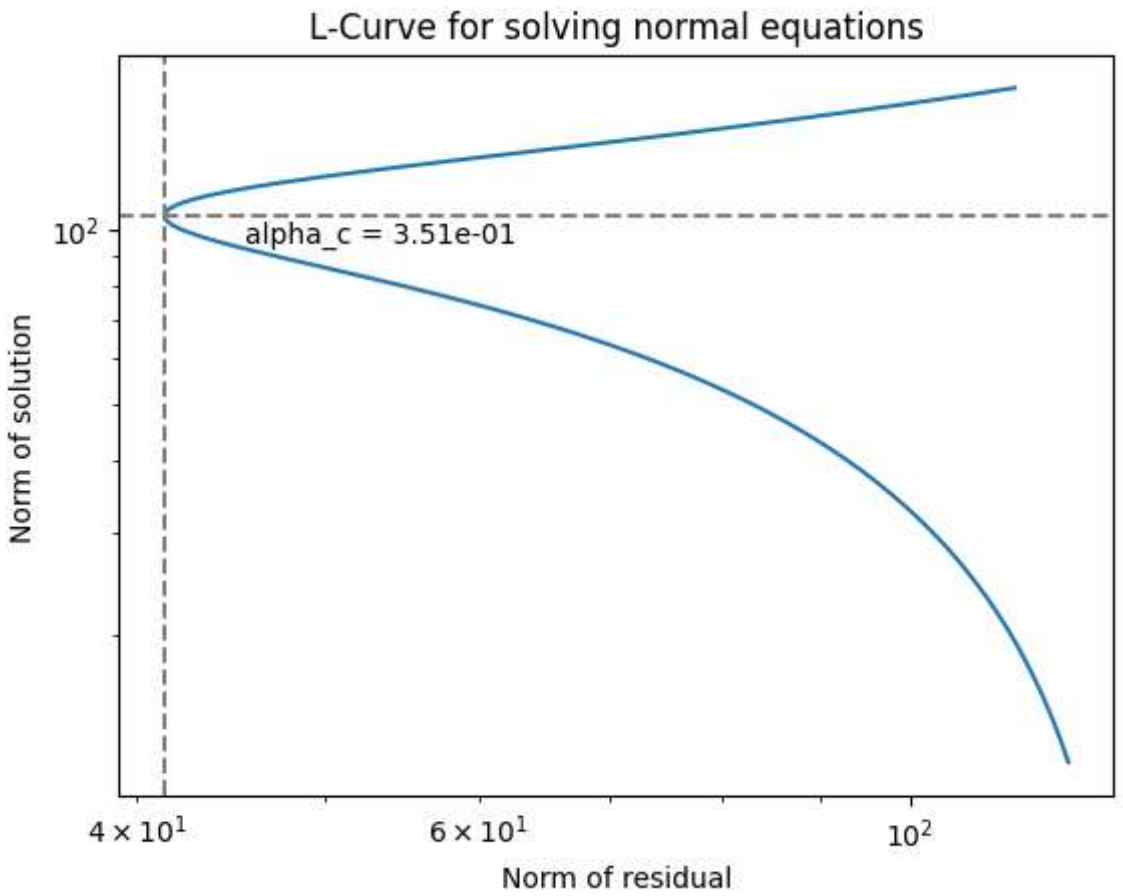
The Discrepancy Principle method was used to find the optimal values of α for the solutions obtained using both the normal equation and augmented equation methods. The optimal value of α was found to be `1e-5` for the normal equation method and `1e-2` for the augmented equation method. The initial guess for the value of α was set to `1e-5` for the normal equation method and `1e-2` for the augmented equation method, respectively.

In [ ]:
```
from scipy.optimize import root,brentq

# method c
```

```python
def discrepancy_function_nor(alpha, g, sigma):
    # Compute the residual
    f_alpha = deconvolve_normal_equations(g, sigma, alpha)
    r_alpha = apply_convolution(f_alpha, sigma, 0) - g

    r_norm = np.linalg.norm(r_alpha)

    # Compute the discrepancy principle
    n = g.size
    sigma_sq = (r_norm / np.sqrt(n))**2
    dp = 1/n * r_norm**2 - sigma_sq

    return dp

def find_optimal_alpha_dp_nor(g, sigma):
    # Set up alpha range
    alpha_min = 1e-5
    alpha_max = 1

    # Find the value of alpha that gives the zero of the DP(alpha) function
    try:
        # alpha_dp = brentq(discrepancy_function_nor, alpha_min, alpha_max, args=(g, sigma))
        alpha_dp = root(discrepancy_function_nor, alpha_min, args=(g, sigma)).x[0]

        print("Optimal alpha value using Discrepancy Principle for deconvolution using normal equations: ", alpha_dp)
    except ValueError:
        print("Failed to find optimal alpha using Discrepancy Principle.")
    return alpha_dp

alpha_dp_c = find_optimal_alpha_dp_nor(g,sigma)
```

ii) L-Curve

The L-curve method was used to determine the optimal value of α for the normal equation method, and it was found to be `3.51e-1`. However, this method cannot be used for the augmented equation method.



In [ ]:
```python
# Define range of alpha values to test
alphas = np.logspace(-1, 1, num=100)

# method c
def l_curve(g, sigma, alphas, deconvolve_fun):
    norm_f = []
    norm_residual = []


    for alpha in alphas:
        if deconvolve_fun == 'normal equations':
            # Compute f for current alpha
```

```
            f_alpha = deconvolve_normal_equations(g, sigma, alpha)
        elif deconvolve_fun == 'augmented equations':
            print('L-curve is not applicable to augmented equations method')
        # Compute residual and f norms
        r_alpha = g - f_alpha #gaussian_filter(f_alpha, sigma)
        norm_f.append(np.linalg.norm(f_alpha))
        norm_residual.append(np.linalg.norm(r_alpha))


    return norm_f, norm_residual

# Compute the L-curve
norm_f_nor, norm_residual_nor = l_curve(g, sigma, alphas, 'normal equations')


def plot_l_curve(norm_f, norm_residual, deconvolve_fun):
    assert deconvolve_fun == 'normal equations', 'L-curve is not applicable to augmented equations method'
    # Plot the L-curve
    import matplotlib.pyplot as plt
    plt.loglog(norm_residual, norm_f)
    plt.xlabel('Norm of residual')
    plt.ylabel('Norm of solution')
    plt.title('L-Curve for solving {}'.format(deconvolve_fun))

    # Find the optimal alpha value based on the L-curve
    curvature = []
    for i in range(1, len(alphas)-1):
        dx1 = norm_residual[i] - norm_residual[i-1]
        dy1 = norm_f[i] - norm_f[i-1]
        dx2 = norm_residual[i+1] - norm_residual[i]
        dy2 = norm_f[i+1] - norm_f[i]
        curvature.append(abs(dx1*dy2 - dx2*dy1) / ((dx1**2 + dy1**2)**1.5 * (dx2**2 + dy2**2)**1.5))

    optimal_index = np.argmax(curvature) + 1
    alpha_l = alphas[optimal_index]

    # Plot the optimal alpha value
    plt.axvline(norm_residual[optimal_index], linestyle='--', color='gray')
    plt.axhline(norm_f[optimal_index], linestyle='--', color='gray')
    plt.text(norm_residual[optimal_index]*1.1, norm_f[optimal_index]*0.9, 'alpha_c = {:.2e}'.format(alpha_l))
```

```
    plt.show()
    return alpha_l

alpha_l_c = plot_l_curve(norm_f_nor, norm_residual_nor, 'normal equations')
```

Comments

Difference in value obtained

It's possible that the optimal alpha found by the L-curve method is different from the optimal alpha found by the Discrepancy Principle method because they use different criteria to determine the optimal alpha.

The L-curve method tries to find the point on the L-curve that balances the trade-off between the residual norm and the regularization norm. This point represents the optimal alpha value that provides a good balance between overfitting and underfitting.

On the other hand, the Discrepancy Principle method tries to find the alpha value that provides a solution that is consistent with the noise level in the data. The optimal alpha value is the one that satisfies the principle that the residual norm is approximately equal to the noise level.

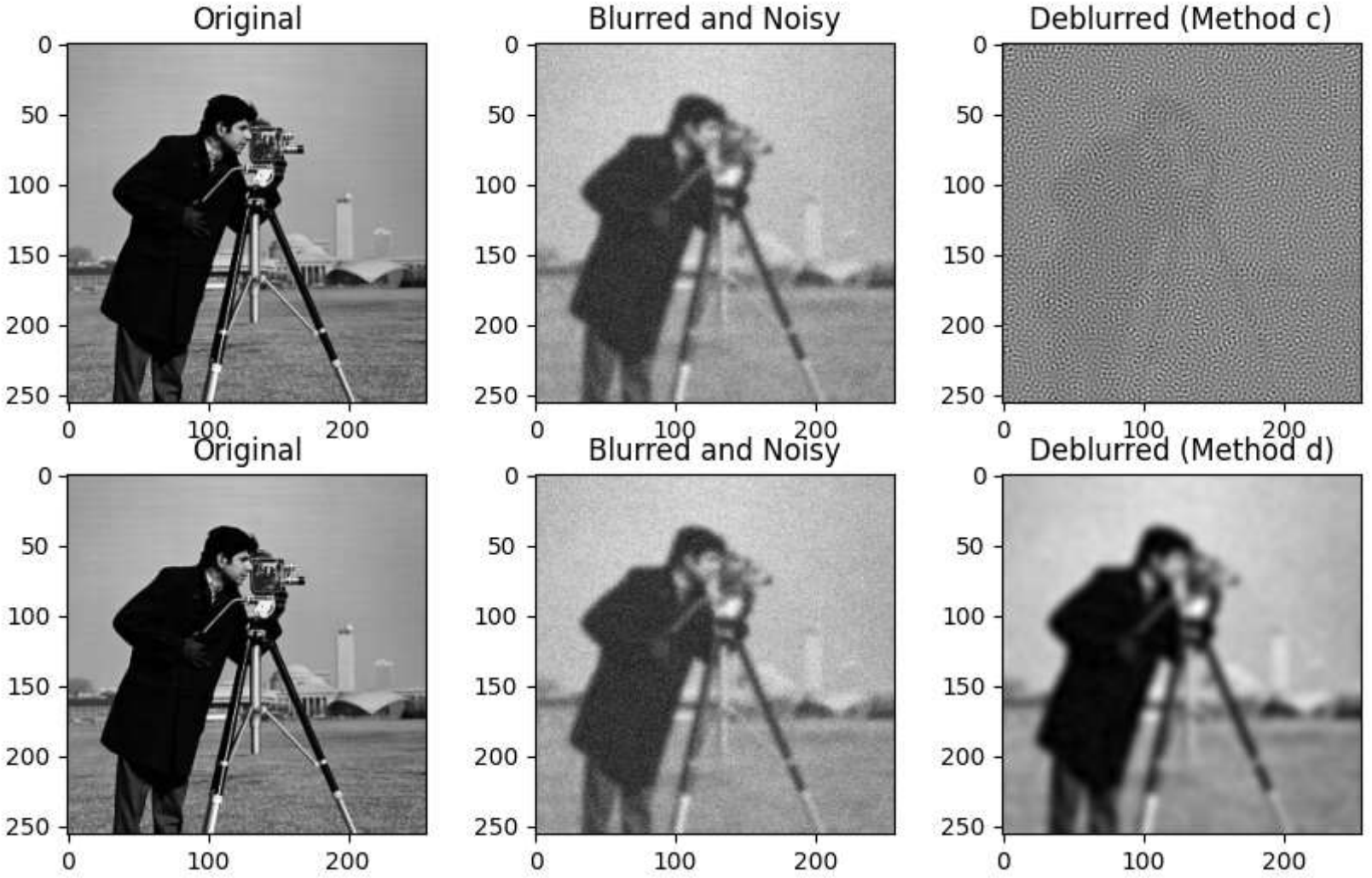
It's possible that the optimal alpha values found by these two methods are different because they are optimizing for different criteria. The L-curve method is optimizing for a trade-off between the residual and regularization norms, while the Discrepancy Principle method is optimizing for a solution that is consistent with the noise level.

Results using these optimal alpha values

1) Results using DP optimal alpha value for normal equation

Method c: Converged in 0 iterations (114.29729723930359 seconds)

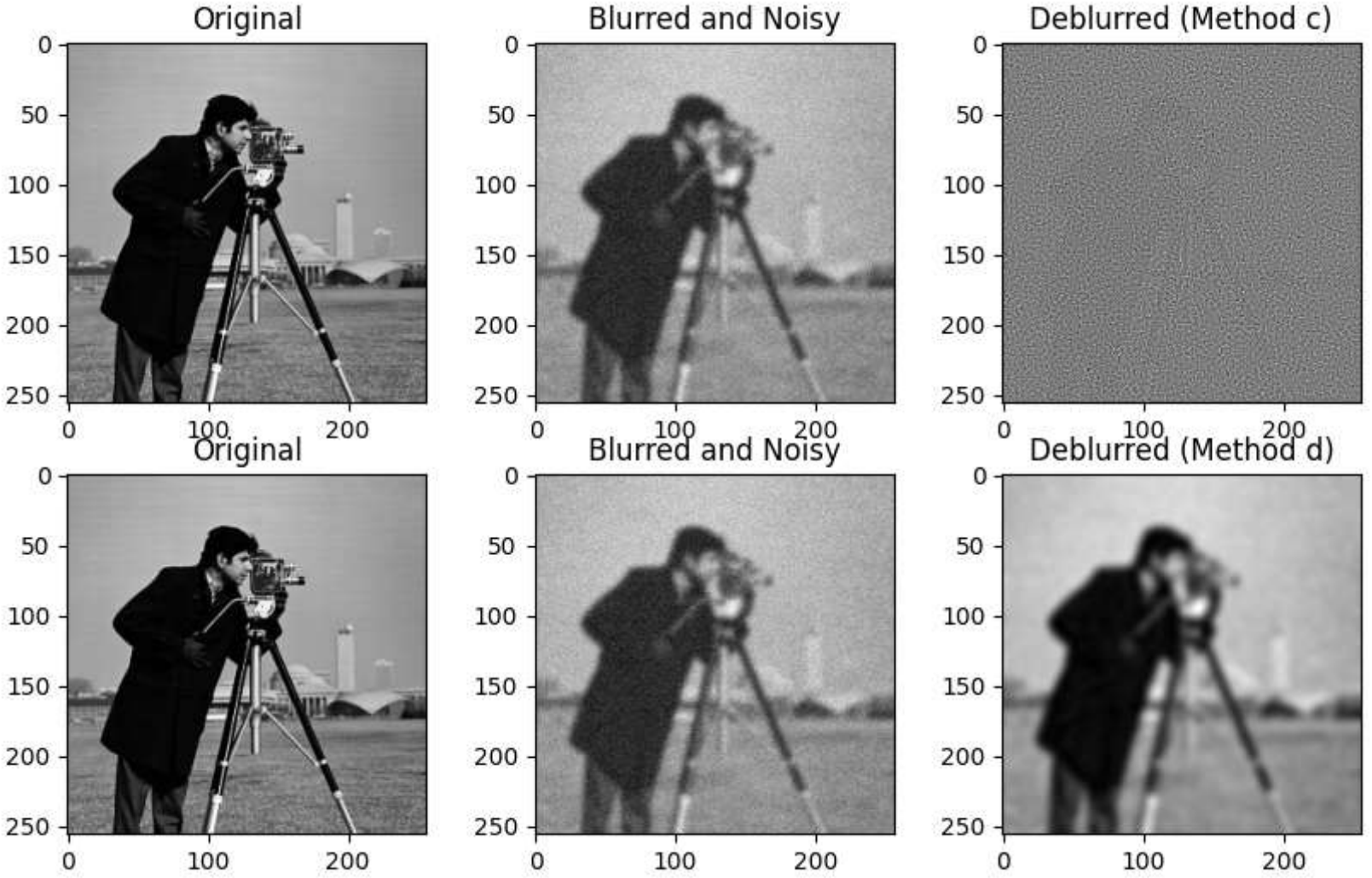
Method d: Converged in 2 iterations (0.046361446380615234 seconds)



2) Results using DP optimal alpha value for augmented equation

Method c: Converged in 0 iterations (0.27139997482299805 seconds)

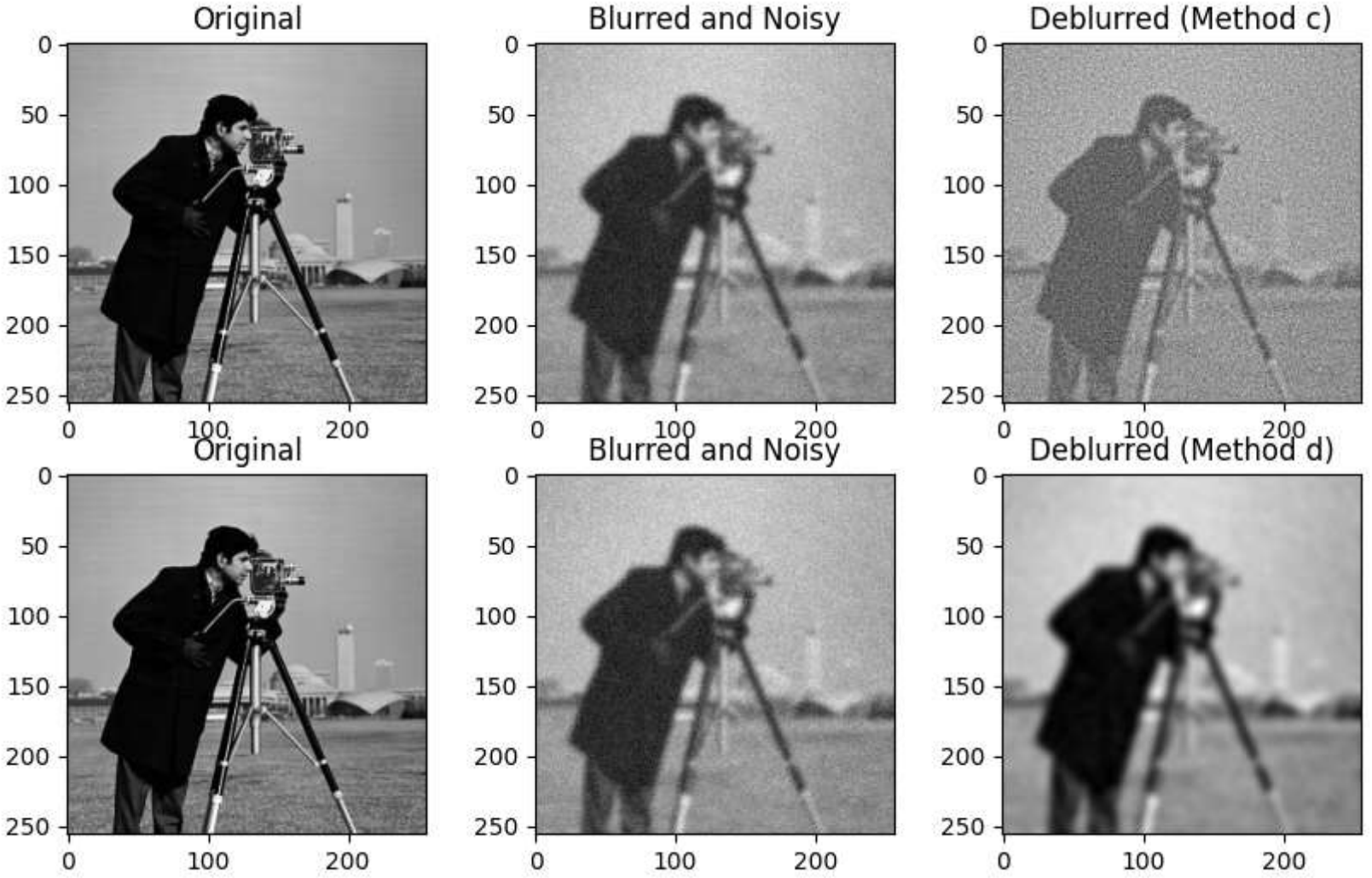
Method d: Converged in 2 iterations (0.04271340370178223 seconds)



3) Results using L-curve optimal alpha value for normal equation

Method c: Converged in 0 iterations (0.05376148223876953 seconds)

Method d: Converged in 2 iterations (0.04639315605163574 seconds)



Based on the fact that the optimal values of alpha found using the Discrepancy Principle method are the same as initial guess alphas, as well as the fact that the result of the deblurred plots are unclear when using the optimal alphas found by the DP method, it shows that Discrepancy Principle method is not a good choice to find the optimal value of α for the data being used.

The Discrepancy Principle method can be useful for finding an approximate range of α values that may be optimal for the given data, but it may not always be able to pinpoint the exact optimal value. In my case, the optimal value of α may lie outside of the range searched by the DP

method, or may require a more fine-tuned search to find.

Therefore, it may be necessary to explore other methods, such as the L-curve method or other regularization methods, to find the optimal value of α for the given data. Additionally, it may be helpful to visually inspect the deblurred plots to determine if the result is acceptable or if further optimization is needed.

3. Using a regularisation term based on the spatial derivative

a) Gradient operator D

Construct the gradient operator and implement it as a function like the forward convolution.

$$D = \begin{pmatrix} \nabla_x \\ \nabla_y \end{pmatrix}$$

In [ ]:
```python
def gradient_operator(im, is_transpose=False):
    if not is_transpose:
        # Forward difference operator
        grad_x = np.diff(im, axis=1)
        grad_y = np.diff(im, axis=0)
        # Pad the end of each dimension with zeros to maintain shape
        grad_x = np.pad(grad_x, ((0, 0), (0, 1)), mode='constant')
        grad_y = np.pad(grad_y, ((0, 1), (0, 0)), mode='constant')
        # Stack x and y gradients to make a (M*N, 2) matrix
        grad = np.column_stack([grad_x.ravel(), grad_y.ravel()])
        return grad
    else:
        # Transpose of forward difference operator
        shape = im.shape
        # Reshape input to a (M*N, 2) array
        # Compute the transpose operator by taking backward differences
        DT_im = np.zeros((*shape,2), dtype=np.float64)
        DT_im[:, :-1,  0] -= im[:, :-1] - im[:, 1:]
        DT_im[:, 1:,  0] += im[:, :-1] - im[:, 1:]
        DT_im[:-1, :, 1] -= im[:-1, :] - im[1:, :]
        DT_im[1:, :, 1] += im[:-1, :] - im[1:, :]
```

```
        DT_im = DT_im.squeeze() # remove extra dimension from DT_im
        return DT_im.reshape(*shape,2)
```

check the correctness of constructing transpose operator D^T by simply examining the identity

$$< Du, v >=< u, D^T v >$$

for random u and v.

In [ ]:
```
# Create random arrays u and v
shape = im.shape
u = np.random.rand(*shape)
v = np.random.rand(*shape)
u_flat = u.reshape(-1, 1)  # Shape is (65536, 1)
v_flat = v.reshape(-1, 1)  # Shape is (65536, 1)

# Compute < Du, v > and < u, DTv >
Du = gradient_operator(u)
DTv = gradient_operator(v,is_transpose = True)

DTv_flat = DTv.reshape(-1, 2)  # Shape is (65536, 2)
u_flat_broadcast = np.broadcast_to(u_flat, (65536, 2))  # Shape is (65536, 2)

Du_flat = Du.reshape(-1, 2)  # Shape is (65536, 2)
v_flat_broadcast = np.broadcast_to(v_flat, (65536, 2))  # Shape is (65536, 2)


left = np.dot(Du_flat, v_flat_broadcast.T).sum()
right = np.dot(u_flat_broadcast, DTv_flat.T).sum()

# Check if the two sides are equal
tol = 1e-8
if np.abs(left - right) < tol:
    print("Identity verified!")
else:
    print("Error: left and right sides are not equal.")
```

b) Normal equation solver

Solve the gradient regularised problem with normal equation solvers.

In Question 1, we solve the regularised least square problem

$$f_\alpha = \text{argmin}_f ||Af - g||_2^2 + \alpha ||f||_2^2$$

where $||f||_2$ is chosen to be the regulariser. In this Question, we are going to use a new regulariser $||Df||_2$ to penalise the gradient instead. We want to solve the regularized least squares problem:

$$f_\alpha = \arg\min_f ||Af - g||_2^2 + \alpha ||Df||_2^2,$$

where A is the data matrix, g is the observation vector, D is the gradient operator, and α is the regularization parameter. To derive the normal equations, we first expand the second term as follows:

$$\alpha ||Df||_2^2 = \alpha f^T D^T D f = f^T \alpha D^T D f.$$

Then, we form the Lagrangian:

$$L(f, \lambda) = ||Af - g||_2^2 + f^T \alpha D^T D f + \lambda^T (f - x),$$

where λ is the Lagrange multiplier and x is the solution to the constrained problem $f \in x | Dx = 0$. Taking the derivative of L with respect to f and setting it to zero, we get:

$$2A^T(Af - g) + 2\alpha D^T D f + \lambda = 0.$$

Multiplying both sides by D and using the fact that $D^T D = -\nabla^2$, we get:

$$-2\alpha \nabla^2 f + D\lambda = 0.$$

Since D is a sparse matrix, we can solve for λ using a sparse linear system solver. Substituting this back into the previous equation, we get:

$$(A^T A + \alpha D^T D) f = A^T g.$$

This is the normal equation for the gradient regularized problem.

In [ ]:
```python
def ATA_operator(f, sigma, alpha):
    # Apply A^T A + alpha*D^T D operator to f
    Af = gaussian_filter(f, sigma)
    ATAf = gaussian_filter(Af, sigma)
    Df = gradient_operator(f)
    DT_Df = gradient_operator(Df, is_transpose=True)
    return ATAf + alpha * DT_Df

def deconvolve_normal_equations(g, sigma, alpha):
    # Set up linear operator for ATA
    M, N = g.shape
    A = sparse.linalg.LinearOperator((M*N, M*N), matvec=lambda x: np.ravel(ATA_operator(x.reshape(g.shape), sigma, alpha)))

    # Compute ATg
    ATg = np.ravel(g)

    # Solve linear system using CG
    f_alpha, info = splinalg.cg(A, ATg)

    return f_alpha.reshape((M, N))
```

b) Augmented equation solver

Solve the gradient regularised problem with augmented equation solvers.

To use the augmented Lagrangian method to solve the regularized least squares problem with gradient regularization, we can introduce a Lagrange multiplier λ and rewrite the problem as:

$$\min_f \frac{1}{2}||Af - g||_2^2 + \frac{\alpha}{2}||Df||_2^2 + \lambda^T(Df - y) + \frac{\mu}{2}||Df - y||_2^2,$$

where y is an intermediate variable and μ is a penalty parameter that controls the degree of constraint violation. We can then apply the alternating direction method of multipliers (ADMM) to solve this problem iteratively.

At each iteration, we update f by solving the following least squares problem:

$$(A^TA + \alpha D^TD + \mu I)f = A^Tg + D^T(y - \lambda/\mu),$$

where I is the identity matrix. We can use any linear system solver to solve this equation, including LSQR.

We then update y by projecting $Df + \lambda/\mu$ onto the constraint set $Df = y$:

$$y = \max(0, |Df + \lambda/\mu| - \alpha/\mu) \cdot \text{sign}(Df + \lambda/\mu).$$

Finally, we update the Lagrange multiplier λ by:

$$\lambda = \lambda + \mu(Df - y).$$

We repeat these steps until convergence. The penalty parameter μ can be increased at each iteration to improve the convergence rate. However, if μ is chosen too large, the method may become unstable. Choosing an appropriate value for μ can be tricky and often requires some trial and error.

In [ ]:
```python
def augmented_deconvolve(g, A, alpha, gamma, mu, max_iter=100, tol=1e-5):
    # Compute D and D^T
    Dx, Dy = gradient_operator(g)
    D = sparse.vstack([sparse.csr_matrix(Dx), sparse.csr_matrix(Dy)])
    DT = D.T

    # Set up linear operator for (A^T A + alpha D^T D + mu I) matrix
    M, N = g.shape
    AAT = A.T @ A
    DTD = DT @ D
    I = sparse.eye(M*N, M*N, format='csr')
    L = AAT + alpha * DTD + mu * I
    linear_op = sparse.linalg.LinearOperator((M*N, M*N), matvec=lambda x: L @ x)

    # Compute ATg
    ATg = A.T @ g.flatten()

    # Initialize variables
    f = np.zeros((M, N))
    y = np.zeros((M, N))
    lambd = np.zeros((2*M, N))

    # Iterate until convergence
    for i in range(max_iter):
        # Solve for f
```

```
        rhs = ATg + DT @ (y - lambd/mu)
        f_new, _ = splinalg.lsqr(linear_op, rhs, atol=tol, btol=tol)
        f_new = f_new.reshape((M, N))

        # Update y
        z = D @ f_new + lambd/mu
        y_new = np.maximum(0, np.abs(z) - gamma/mu) * np.sign(z)

        # Update Lagrange multiplier
        lambd_new = lambd + mu * (D @ f_new - y_new)

        # Check convergence
        if np.linalg.norm(f_new - f) < tol and np.linalg.norm(y_new - y) < tol:
            break

        # Update variables
        f = f_new
        y = y_new
        lambd = lambd_new

        # Increase penalty parameter
        mu *= 1.5

    return f
```

c) Chose a value for α, explain your choice

Week 3

4. Anisotropic derivative filter

To perform image denoising with an anisotropic derivative filter, we add weights to the gradient term to obtain the following regularized least squares problem:

$$f_\alpha = \arg\min_f ||Af - g||_2^2 + \alpha||\sqrt{\gamma}Df||_2^2,$$

where A is the data matrix, g is the observation vector, D is the gradient operator, γ is the diffusivity matrix, and α is the regularization parameter. We construct γ as a diagonal matrix with entries between 0 and 1, where 0 indicates regions of the image where we do not want to smooth. One example diffusivity function is the Perona-Malik function:

$$\gamma(f) = \exp(-|Df|/T) = \exp(-\sqrt{(\partial_x f)^2 + (\partial_y f)^2/T}),$$

where T is a threshold based on the maximum expected edge values in the image, which can be estimated from the norm of the image gradient. Note that $\sqrt{\cdot}$ here is an element-wise operation on the diagonal matrix γ, so to calculate $\sqrt{\gamma}D$, we use

$$\sqrt{\gamma}D = \begin{pmatrix} \sqrt{\gamma}\partial_x \\ \sqrt{\gamma}\partial_y \end{pmatrix}.$$

To solve the regularized least squares problem, we can use the normal equation:

$$(A^T A + \alpha D^T \gamma D) f = A^T g.$$

We can solve this equation using any linear system solver, including LSQR.

Alternatively, we can use the augmented Lagrangian method to solve the problem iteratively. At each iteration, we update f by solving the following least squares problem:

$$(A^T A + \alpha D^T \gamma D + \mu I) f = A^T g + D^T \gamma \sqrt{\gamma}(y - \lambda/\mu),$$

where y is an intermediate variable, μ is a penalty parameter, and λ is the Lagrange multiplier. We can use any linear system solver to solve this equation, including LSQR.

We then update y by projecting $\sqrt{\gamma}(Df + \lambda/\mu)$ onto the constraint set $Df = y$:

$$y = \max(0, |\sqrt{\gamma}(Df + \lambda/\mu)| - \alpha/\mu) \cdot \text{sign}(\sqrt{\gamma}(Df + \lambda/\mu)).$$

Finally, we update the Lagrange multiplier λ by:

$$\lambda = \lambda + \mu\sqrt{\gamma}(Df - y).$$

We repeat these steps until convergence. The penalty parameter μ can be increased at each iteration to improve the convergence rate. However, if μ is chosen too large, the method may become unstable. Choosing an appropriate value for μ can be tricky and often requires

some trial and error.

In [ ]:
```python
def anisotropic_derivative_filter(g, alpha, max_iter=100, eps=1e-5):
    """
    Solve an optimization problem with an anisotropic derivative filter using the Perona-Malik diffusivity function.

    Args:
        g (ndarray): the input image
        alpha (float): the regularization strength
        max_iter (int): the maximum number of iterations (default: 100)
        eps (float): the stopping criterion (default: 1e-5)

    Returns:
        f (ndarray): the denoised image
    """

    # Construct the gradient operator
    Dx = spdiags([-1*np.ones(g.shape[0]), np.ones(g.shape[0])], [0, 1], g.shape[0]-1, g.shape[0])
    Dx = Dx.todense()
    Dx[-1, -1] = -1
    Dx = np.kron(np.eye(g.shape[1]), Dx)

    Dy = spdiags([-1*np.ones(g.shape[1]), np.ones(g.shape[1])], [0, 1], g.shape[1]-1, g.shape[1])
    Dy = Dy.todense()
    Dy[-1, -1] = -1
    Dy = np.kron(Dy, np.eye(g.shape[0]))

    D = np.vstack((Dx, Dy))

    # Initialize f
    f = g.copy().flatten()

    # Iterate until convergence or maximum number of iterations
    for i in range(max_iter):

        # Compute diffusivity
        gradient_norm = np.sqrt((Dx @ f)**2 + (Dy @ f)**2)
        T = 0.2 * np.max(gradient_norm)
        gamma = np.exp(-gradient_norm / T)
        gamma = spdiags(gamma, 0, len(gradient_norm), len(gradient_norm)).todense()
```

```
        # Compute denoised image
        f_old = f.copy()
        grad = 2 * (D.T @ (A @ f - g)) + 2 * alpha * (gamma @ D @ f)
        f -= np.linalg.norm(grad) * grad / np.linalg.norm(D @ grad)**2

        # Project f onto [0, 1]
        f[f < 0] = 0
        f[f > 1] = 1

        # Check convergence
        if np.linalg.norm(f - f_old) < eps:
            print("Algorithm converged after", i+1, "iterations.")
            break

        # Check maximum number of iterations
        if i+1 == max_iter:
            print("Maximum number of iterations reached.")

    return f.reshape(g.shape)
```

5. Iterative Deblurring with Anisotropic Derivative Filter

To deblur the image iteratively with this method, we can follow these steps:

1. Take an initial blurry image f_0 and set $i = 0$.
2. Compute the diffusivity $\gamma(f_i)$.
3. Compute the denoised image f_{i+1} using the anisotropic derivative filter with the current value of $\gamma(f_i)$.
4. Increase i by 1 and repeat from step 2 until convergence.

To check for convergence, monitor the relative change in the solution between iterations:

$$\frac{||f_{i+1} - f_i||_2}{||f_i||_2} < \epsilon,$$

where ϵ is a small tolerance parameter. If this condition is satisfied, it can be assumed that the solution has converged.

Alternatively, it can be stopped by setting a maximum number of iterations $N_{\max}$ and terminating the algorithm when this limit is reached, in case the algorithm fails to converge or in cases where the convergence criteria may be too strict.

In [ ]:
```python
def iterative_deblurring(g, A, alpha, max_iter=100, eps=1e-5):
    """
    Perform iterative deblurring with anisotropic derivative filter using the Perona-Malik diffusivity function.

    Args:
        g (ndarray): the observed blurry image
        A (ndarray): the blurring operator
        alpha (float): the regularization strength
        max_iter (int): the maximum number of iterations (default: 100)
        eps (float): the stopping criterion (default: 1e-5)

    Returns:
        f (ndarray): the deblurred image
    """

    # Compute maximum gradient magnitude of blurred image
    Dx, Dy = gradient_operator(g.shape)
    gradient_norm = np.sqrt((Dx @ g.flatten())**2 + (Dy @ g.flatten())**2)
    max_gradient_norm = np.max(gradient_norm)

    # Set threshold value for Perona-Malik function
    T = 0.2 * max_gradient_norm

    # Initialize variables
    f = g.copy()

    # Iterate until convergence
    for i in range(max_iter):
        # Compute diffusivity
        gradient_norm = np.sqrt((Dx @ f.flatten())**2 + (Dy @ f.flatten())**2)
        gamma = np.exp(-gradient_norm / T)
        gamma = spdiags(gamma, 0, len(gradient_norm), len(gradient_norm)).todense()

        # Compute denoised image
        f_old = f.copy()
        f = spsolve(A.T @ A + alpha * D.T @ gamma @ D, A.T @ g.flatten())
```

```
        f = f.reshape(g.shape)

        # Check convergence
        if np.linalg.norm(f - f_old) < eps:
            print("Algorithm converged after", i+1, "iterations.")
            break

        # Check maximum number of iterations
        if i+1 == max_iter:
            print("Maximum number of iterations reached.")

    return f
```