Inverse Problems CW2

Student 23188618

February 2024

1 Introduction

1. (a) A blurry image, observed with some noise n (with standard deviation θ), can be represented as follows

$$g = Af_{true} + n \tag{1}$$

where A is the Gaussian convolution kernel, and f_{true} is the original, undistorted image that we aim to recover or estimate from the observed data g. For this investigation, we have chosen the image house.png[1].

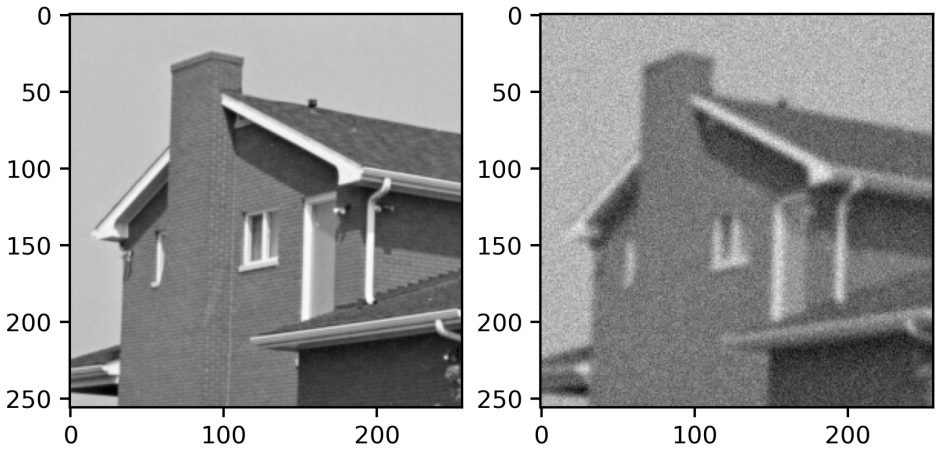

Figure 1: Greyscale image of a house that will be used as f_{true} (left) and the blurred image with added noise (right).

(b) To blur this image, I constructed a function that implicitly convolves the image f through the use of the `gaussian filter()` function from SciPy. The noise n is then added separately from a standard Normal distribution.

```
def imblur(f,sigma,w,h):
    """Blurs an image using a gaussian filter"""
    Af = sp.ndimage.gaussian_filter(f.reshape((w,h)),sigma)
    return Af.flatten()
g = imblur(ftrue,sigma,w,h)
g = g + theta*np.random.randn(g.size)
```

The original image is plotted in greyscale in 1, alongside the result of applying our blurring function to it with the following parameters: $\sigma = 2, \theta = 0.05, \alpha = 0.05$. This problem can

[1] http://www0.cs.ucl.ac.uk/staff/S.Arridge/teaching/optimisation/CW2/house.png

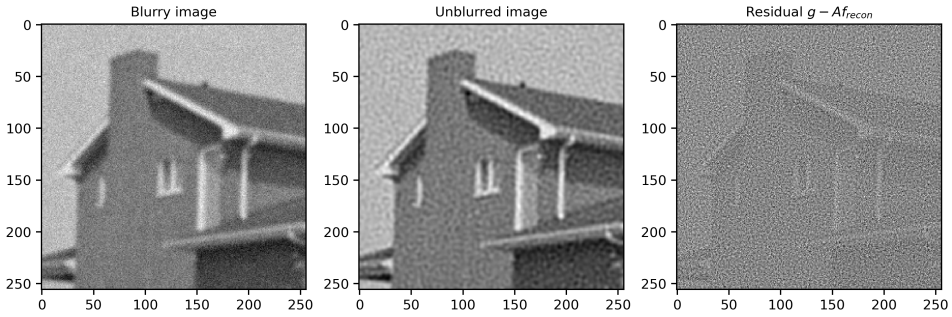

Figure 2: Left: Blurred image from 1. Center: Unblurred reconstruction obtained using the GMRES solver. Right: The residual error between the first two images.

be solved by finding the value of f_{recon} that minimises the following expression:

$$f_\alpha = \arg\min_f \|Af - g\|_2^2 + \alpha \|f\|_2^2 \tag{2}$$

The first term can be rewritten as $(Af - g)^T(Af - g)$. Expanding out gives,

$$\begin{aligned}(Af - g)^T(Af - g) &= (f^T A^T - g^T)(Af - g) \\ &= f^T A^T Af - f^T A^T g - g^T Af + g^T g\end{aligned} \tag{3}$$

Taking the derivative of this expression with respect to f and setting to 0 yields:

$$\frac{\partial}{\partial f}\left(f^T A^T Af - f^T A^T g - g^T Af + g^T g\right) \tag{4}$$

$$= 2A^T Af - 2A^T g = 0 \tag{5}$$

$$\tag{6}$$

The derivative of the regularisation term is simply αf. Thus, by combining the two, we have that

$$(A^T A + \alpha I) f_\alpha = A^T g \tag{7}$$

where f_α is the reconstructed image that satisfies this equation for a given α.

(c) GMRES is an iterative method for solving non-symmetric linear systems such as this image deblurring. When implementing GMRES to deconvolve this image, the size of the linear system being solved must be accounted for accurately. The size of this system is (S, S) where S is the number of pixels in the image. Since A is self-adjoint, meaning $A = A^T$, calculating $A^T A$ is equivalent to blurring the input twice. Then, the αf term is added to the result and the array is flattened, as the GMRES solver expects a vector input for the `matvec` argument. For the parameters specified in 1. (a), the GMRES function ran for 17 iterations before convergence. Qualitatively, it is evident that there is some information loss in the unblurred image, and this is further proven quantitatively when looking at the residual graph in 2. An ideal deblurring would not contain features from the original image (as those would all be perfectly reconstructed) but only noise, which is not the case here.

(d) Least-Squares LSQR is an alternative iterative method for solving these types of linear systems, and it does not require computation of $A^T A$. Instead, it leverages the augmented equations:

$$\begin{pmatrix} A \\ \sqrt{\alpha I} \end{pmatrix} f = \begin{pmatrix} g \\ 0 \end{pmatrix}, \tag{8}$$

As a result of stacking A and αI in this augmented form, the size of the system for this solver is $(2S, S)$. This method requires the forward equations from Equation, and the transpose of this system, where instead the

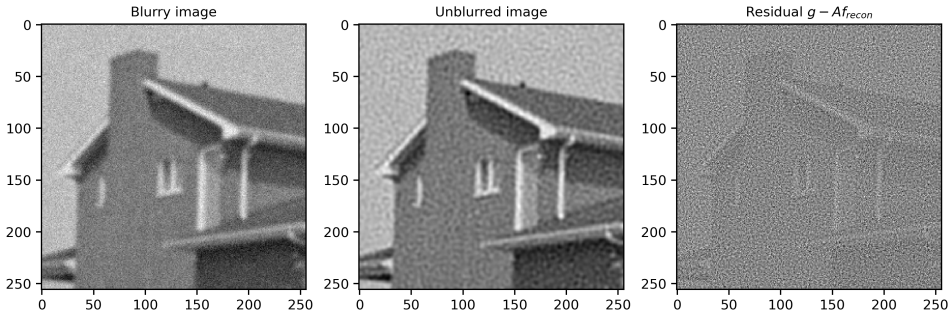

Figure 3: Left: Blurred image from 1. Center: Unblurred reconstruction obtained using the LSQR solver. Right: The residual error between the blurred image its reconstruction.

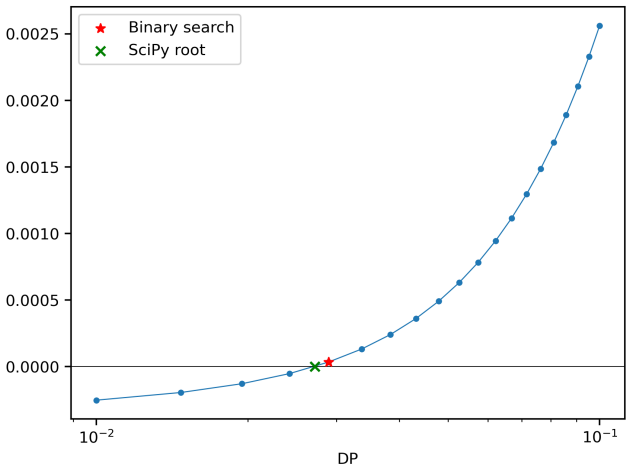

Figure 4: The result of Discrepancy Principle optimisation for the regularisation parameter on a GMRES solver.

2. (a) The optimal value for the regularisation parameter achieves a trade-off between bias and variance. To find the optimal regularisation parameter for the GMRES method, two methods were employed below.

   i. The first method is the Discrepancy Principle method. This method requires prior knowledge of noise statistics (white-noise in this context) and works on the assumption that the norm of the residual is close to the expectation of the norm of the noise, which for Gaussian noise is the standard deviation θ.

$$DP(\alpha) = \frac{1}{n}||r_\alpha||^2 - \theta^2 \tag{9}$$

   where r_α is the difference between the original blurry measurement and the blurry reconstruction, $r_\alpha = g - Af_\alpha$. For various α values, the optimal regularisation parameter is expected to be found at the zero-crossing of $DP(\alpha)$. I used binary search as well as `sp.optimize.root()` to find the best value of α. I employed binary search as a precautionary measure, as I noticed that for lower α values (ie. $< 10^{-5}$), the SciPy function was a bit slower to converge. The results of this regularisation parameter optimisation can be seen in Figure

   ii. The L-Curve method also uses the residual to find the optimal regularisation parameter but instead plots the norm of the residuals $||r_\alpha||$ against the norm of the reconstructed solutions $||f_\alpha||$ and finds the sharp corner of the L-shaped curve.

   While the exact coordinates of the sharpest point of the curve can be calculated, I

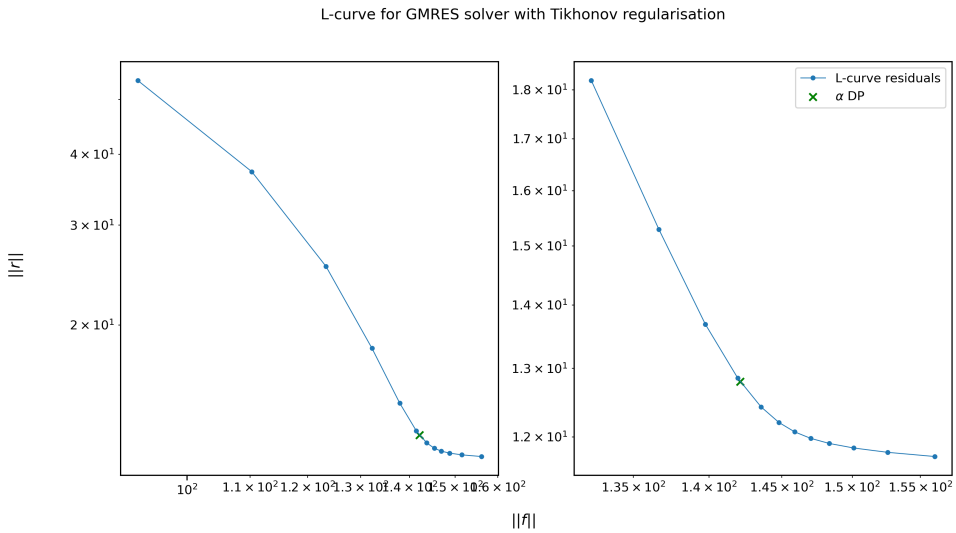

Figure 5: Two views of the L-curve for the GMRES solver, with the Discrepancy Principle result plotted as well.

assumed that it was beyond the scope of this investigation. Instead, α is estimated qualitatively; for the parameters and random noise level generated, we can see that the optimal α value is between 10^{-3} and 10^{-2}. I have also plotted α_{DP} for comparison's sake and found that it is generally near the elbow of the L-curve, though not precisely at the same point. The L-curve is a useful geometric tool for finding α, in essence it only finds the corner point in the curve that indicates a balance between the function norm and the residual norm. However, in noisy situations or when there is correlated noise, this method may break down. That being said, in the real world, it is naturally rare to have knowledge of a true noise distribution. For this reason, while both methods are useful in their own right, I believe that the Discrepancy Principle is a far more valuable tool for the task at hand, given our a priori knowledge of the noise distribution. In light of this, I shall use the Discrepancy Priniple for subsequent explorations into optimal α values in this report.

To test that the LSQR method and GMRES method produced the same α_{opt} I compared the values obtained from the Discrepancy Principle for both solvers and found that they were indeed equal, within a numerical tolerance of 10^{-4}. I have printed the outputs of my code below:

```
opt_alpha_LSQR, opt_alpha_GMRES
>>> (0.0264326536277854, 0.02643046152599466)
```

3. First-order Tikhonov regularization penalizes the first derivatives of the solution. In the discrete setting, this derivative is a finite difference matrix D that contains the differences between adjacent elements.

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

(a) As in Part 1, constructing the derivative matrix for 2D images is compute-intensive, and it is implemented as a function that acts on the image and returns the differentiated values f'. The x-derivative and y-derivative matrices are calculated separately as it is more efficient and convenient for computation in further steps. The x-derivative function is displayed below; the The y-derivative applies `np.diff()` to columns, and the boundary condition (zero gradient at the boundary) is implemented as a `np.hstack()`.

```
def applyDx(u):
```

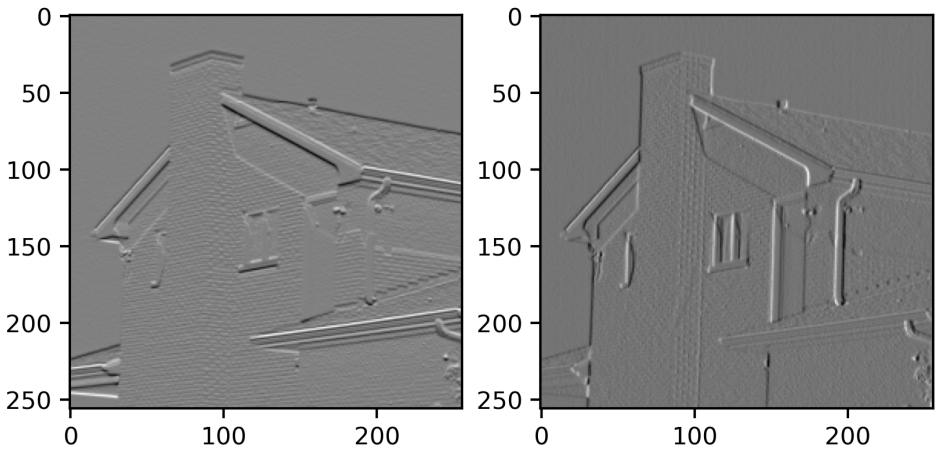

Figure 6: Left: applying x-component of gradient operator to f_{true}. Right: Applying y-component to the same image.

```
    """Applies the first derivative along rows (x)"""
    Dx = np.diff(u, axis=0)
    Dx = np.vstack([Dx, np.zeros((1, u.shape[1]))])
    return Dx

def applyDy(u):
    """Applies the first derivative along columns (y)"""
    Dy = np.diff(u, axis=1)
    Dy = np.hstack([Dy, np.zeros((u.shape[0], 1))])
    return Dy
```

I have plotted the derivatives along each direction for a visual representation of the image gradient in 6. These images demonstrate how the gradient is calculated for when stark changes occur along x (edges), and along y.

(b) The normal equations in Part 1. c) need to be modified to account for the incorporation of the derivative matrix D. The regularisation term (excluding α, as it is just a constant that I add back later) becomes

$$\psi(f) = \frac{1}{2}\|Df\|^2 = \frac{1}{2}f^\top D^\top Df \tag{10}$$

The derivative of this term WRT f is $D^\top Df$, and it can be seen that $D^\top D = \nabla^2$. This Laplacian can be computed simply as $\nabla_x^2 + \nabla_y^2$, and is used directly in the GMRES solver when computing the αI. The Laplacian Python function is shown below:

```
def laplacian(f):
    dxx = applyDxTrans(applyDx(f))
    dyy = applyDyTrans(applyDy(f))
    laplace_f = dxx + dyy

    return laplace_f.flatten()
```

For LSQR, incorporating the first-order derivative is a bit more intensive to incorporate into the linear system. The reason for this is the augmented system of equations now needs to be enlarged to incorporate both the x and y derivatives, as below:

$$\begin{pmatrix} A \\ \sqrt{\alpha}\nabla_x \\ \sqrt{\alpha}\nabla_y \end{pmatrix} f = \begin{pmatrix} g \\ 0 \\ 0 \end{pmatrix}, \tag{11}$$

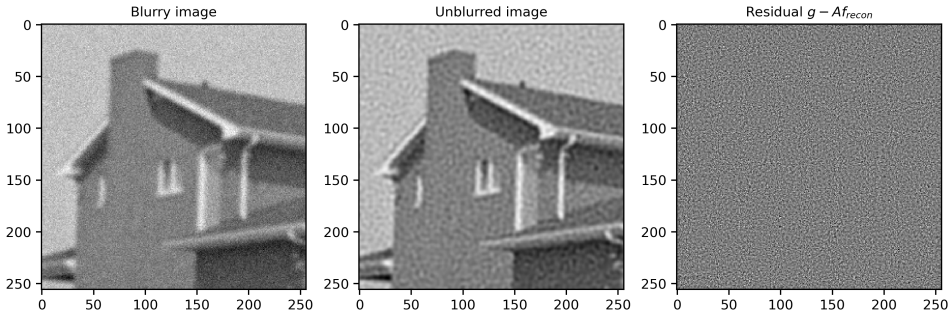

Figure 7: The reconstruction of a blurry image, calculated using GMRES with a spatial difference regularisation term.

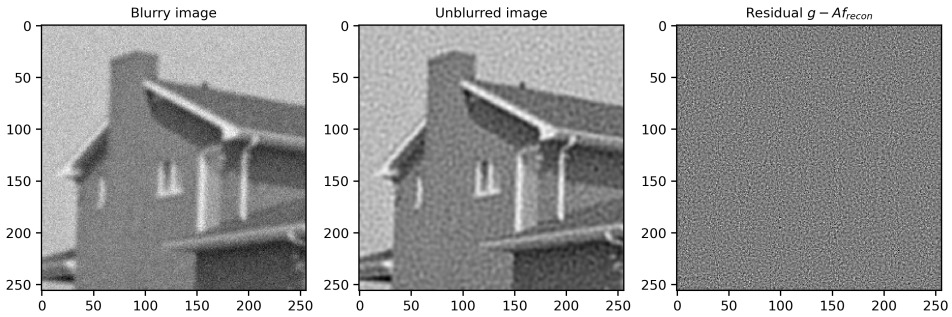

Figure 8: The reconstruction of a blurry image, calculated using GMRES with a spatial difference regularisation term.

And now, the linear system has a size of $(3S, S)$. Solving again with GMRES and LSQR, the results of the reconstruction are shown in Figure 7 and 8.

There is a large improvement from the solution obtained using zero-order Tikhonov regularisation. This can be attributed to the smoothing quality that incorporating the first derivative achieves. Observing 6 again, we can note that the points with sharp edges have a high gradient; in the reconstructions in Part 1. it was these very edges where the reconstruction failed to perform well, leading to information loss.

(c) As discussed in Part 2, given its ability to incorporate information from the noise distribution, I have chosen DP as the favourable method for calculating the optimal regularisation parameter going forward, and I used the GMRES solver for convenience and speed. With the gradient regularisation, α was higher than the zero-order regularisation, at roughly 0.49.

```
alpha_OPT_gradientreg
>>> 0.4921308316855381
```

4. A variation on the uniform first-order Tikhonov (derivative) regularisation is anisotropic regularisation. Unlike isotropic regularisation, this method smooths the solution more in certain directions, specifically where the image gradient is lower; this is done to preserve the edges and features. The Perona-Malik function defines the diffusivity matrix as follows:

$$\gamma(f) = \exp\left(-\frac{|Df|}{T}\right) = \exp\left(-\frac{\sqrt{(\nabla_x f)^2 + (\nabla_y f)^2}}{T}\right) \tag{12}$$

I constructed two versions of γ, once using `spdiags` and once as a function applied to an image.

```
def perona_malik(f):
```

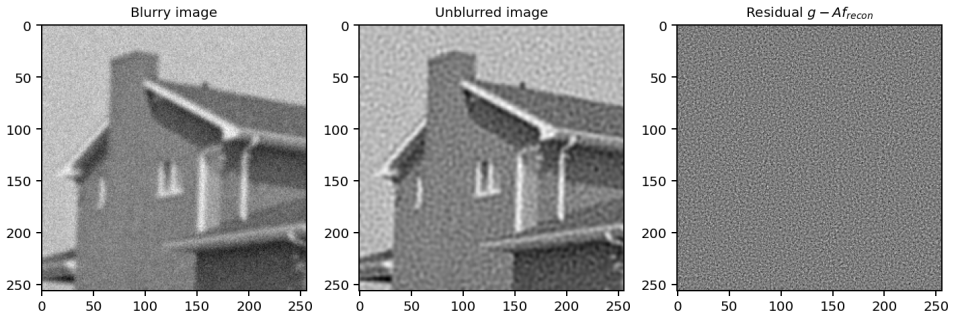

Figure 9: First iteration of deblurring with Perona-Malik diffusivity.

```
assert f.ndim > 1
n = f.shape[0]
Dx2 = np.square(applyDx(f))
Dy2 = np.square(applyDy(f))
T = np.sqrt(Dx2 + Dy2).max()
gamma = np.exp(-np.sqrt(Dx2 + Dy2)/T) ## comment out depending on need
gamma = sp.sparse.spdiags(gamma.ravel(),diags=0,m=n,n=n).toarray()
return gamma
```

5. My Laplacian function was modified to include a Gamma parameter (replaced by identity matrix when not implementing this type of regularisation).

```
def laplacian(f,gamma=0):
if not isinstance(gamma, np.ndarray):
    gamma = np.eye(f.shape[0])
dxx = np.sqrt(gamma)@ applyDxTrans(np.sqrt(gamma)@applyDx(f))
dyy = np.sqrt(gamma)@ applyDyTrans(np.sqrt(gamma)@applyDy(f))
laplace_f = dxx + dyy

return laplace_f.flatten()
```

I evaluated two representations of the Perona-Malik diffusivity γ due to undesirable results (or, at least, what I believe are undesirable results) which arose while attempting to calculate the GMRES solution using anisotropic regularisation.

I had expected to see smoothing in the non-edges (constant pixel intensity areas) and more sharpness near edges. However, irrespective of my choice of method for calculating γ, the result was not what was expected either way. Both methods resulted in reconstructions that the more iterations I performed, the more the image no longer resembled the original image, and after the 10th iteration, the reconstruction had essentially turned into noisy Gaussian circles. For this reason, I chose 5 epochs as beyond that, there was nothing more that could be derived from the images unfortunately. The result of this exploration can be seen in 9, 10 and 11.

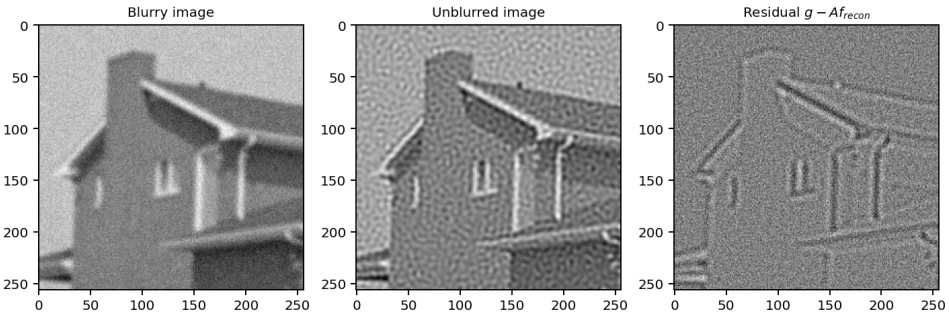

Figure 10: Second iteration of deblurring with Perona-Malik diffusivity.

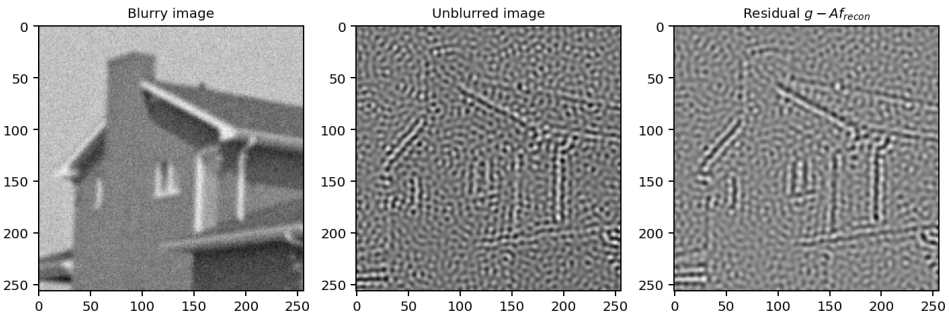

Figure 11: Fifth iteration of deblurring with Perona-Malik diffusivity.