Assignment 2

Callum Lau, 19102521

Inverse Problems

February 27, 2020

1 Convolution and Deconvolution

1.1 a.)

The greyscale 256 x 256 ($N \times N$) cameraman image f_{true} is given below:

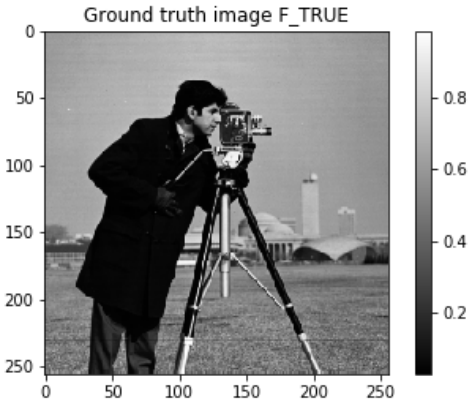


Figure 1: f_{true}

1.2 b.)

We then define the convolution mapping A as a function `imblur` and add Gaussian noise. We will use fixed values of $\sigma = 3.0$ and $\theta = 0.05$ for all the experiments.

```
from scipy.ndimage.filters import gaussian_filter
# Defining blurring function for a given sigma
def imblur(f):
    return scipy.ndimage.filters.gaussian_filter(f, SIGMA)

g = imblur(F_TRUE) + THETA*np.random.randn(N,N)
```

The convolved image with noise using the parameter values $\sigma = 3.0$ and $\theta = 0.02$ is given below:

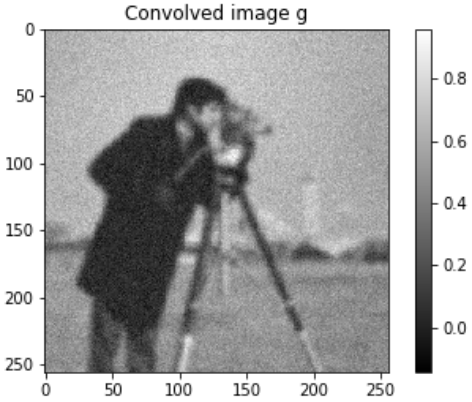


Figure 2: $g = Af_{\text{true}} + n$

1.3 c.)

The normal equations are given by:

$$(A^\top A + \alpha I) f_\alpha = A^\top g$$

Using the fact that $A^\top = A$, we represent $(A^\top A + \alpha I) f_\alpha$ as a function:

```
ATA_I = lambda f, alpha: imblur(imblur(f.reshape((N,N)))) + alpha*(f.reshape((N,N)))
```

and then define a function `gmres_solution` which creates the sparse representation of the operator, and uses the GMRES method to output a deconvolved solution f_{recon}:

```
from functools import partial
# GMRES method to solve Ax = b
def gmres_solution(_alpha, operator):

      A = scipy.sparse.linalg.LinearOperator((N*N,N*N), partial(operator, alpha = _alpha))
      ATg = imblur(g).reshape(N*N, 1)
      gmresOutput = scipy.sparse.linalg.gmres(A, ATg,  callback=counter)
      return gmresOutput

f_recon = gmres_solution(ALPHA, ATA_I)[0]
```

Using the value of $\alpha = 0.1$ as defined above gave the solution below:

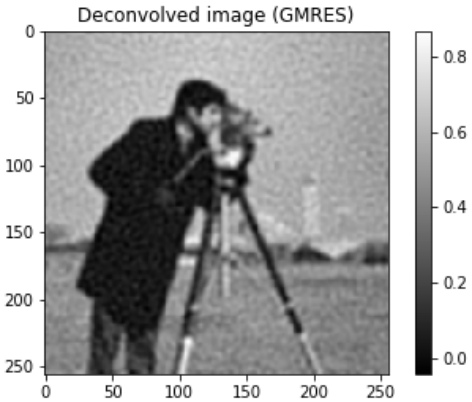


Figure 3: f_{recon}

1.4 d.)

For the LSQR method the equation we wish to solve is

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

where

- f is a $(N^2 \times 1)$ vector (solution image)
- g is a $(2N^2 \times 1)$ vector (augmented data)
- M is a $(2N^2 \times N)$ matrix (augmented operator)

and hence we first define the functions performing the operations Mf and $M^T b$ respectively:

```
def M_f(f, alpha):
    x = imblur(f.reshape((N,N))).reshape((N*N,1))
    y = alpha*f.reshape((N*N,1))
    return np.vstack((x, y))


def MT_b(f, alpha):
    mid = f.size//2
    x = imblur(f[:mid].reshape((N,N))).reshape((N*N,1))
    y = alpha*f[mid:].reshape((N*N,1))
    return x + y
```

We then form the augmented data vector b and the sparse linear operator M using the functions defined above, and pass these to the LSQR solver:

```
b = np.vstack([np.reshape(g,(g.size,1)),np.zeros((g.size,1))])
M = scipy.sparse.linalg.LinearOperator((2*N*N,N*N), matvec = partial(M_f, alpha = ALPHA),
                                       rmatvec = partial(MT_b, alpha = ALPHA))
f_recon = scipy.sparse.linalg.lsqr(M,b)
```

The LSQR solution using $\alpha = 0.1$ is given below:

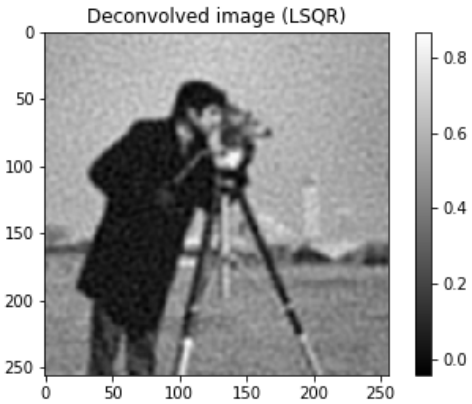


Figure 4: f_{recon}

The performance of the LSQR method compared to the GMRES method was very similar. When both methods were used with the same tolerance of $1e-08$, both methods took exactly 25 iterations. In fact, when comparing different tolerance levels and α values, the performance of both algorithms in terms iof iterations was almost identical. Therefore, no strong conclusions about the suitability of either method over the other for this problem can be made.

2 Choose a regularisation parameter α

2.1 i.)

The discrepancy principle finds the optimal value of α by solving the non-linear optimisation problem:

$$\mathrm{DP}(\alpha) \doteq \frac{1}{n}||\boldsymbol{r}_\alpha||^2 - \theta^2 = 0$$

where

$$\boldsymbol{r}_\alpha = g - Af_\alpha$$

and therefore we first write a function to compute $\mathrm{DP}(\alpha)$ using the GMRES solver:

```
def DP(alpha, operator):
    f_alpha = gmres_solution(alpha, operator)[0].reshape(N,N)
    r_alpha = (1/g.size)*np.linalg.norm(g.flatten()-imblur(f_alpha).flatten())**2-THETA**2
    return r_alpha
```

and use a library root finder to find the solution:

```
dp_solution = scipy.optimize.root(partial(DP, operator=ATA_I), 0.1)
```

which gave an answer of $\alpha \approx 0.0283$. The DP curve is visualised below:

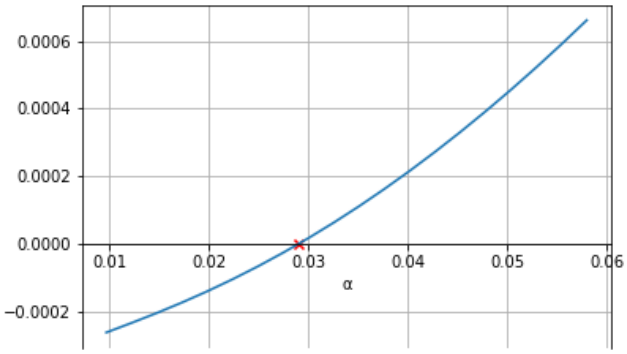


Figure 5: Discrepancy Solution plot

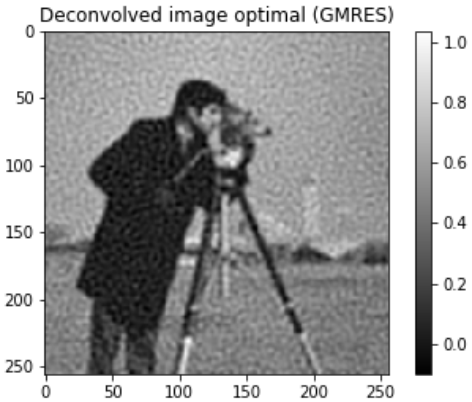


Figure 6: f_α with $\alpha = 0.0283$

2.2 ii.)

For the L-curve we plot the graph of the points:

$$\{||\boldsymbol{r}_\alpha||^2, \Psi(f_\alpha)\}$$

Note we are solving the regularised least squares problem with $\Psi(f) = ||f||_2$ the regulariser. This is accomplished using the code below:

```
alphas = 10**(np.linspace(-3, -1, 40))
llhd = np.zeros(alphas.size)
pr = np.zeros(alphas.size)
for i, a in enumerate(alphas):
    f_alpha = gmres_solution(a, ATA_I)[0].reshape(N,N)
    llhd[i] = np.linalg.norm(g.flatten() - imblur(f_alpha).flatten())**2
    pr[i] = np.linalg.norm(f_alpha.flatten())**2
```

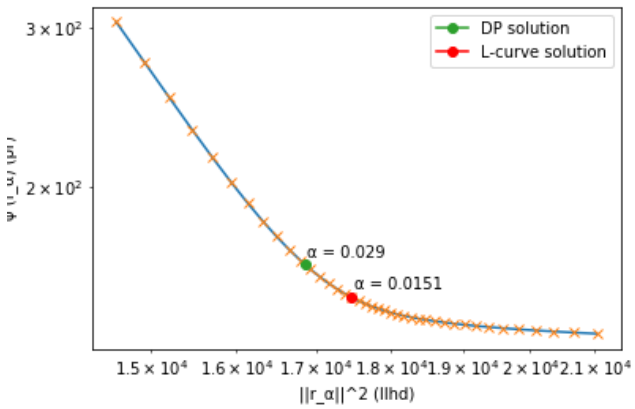


Figure 7: L-curve

From this curve we can eyeball the solution $\alpha = 0.0151$ where the gradient of the curve seems to be changing the most. This is lower than the DP solution. The f_α solution with $\alpha = 0.0151$ is shown below:

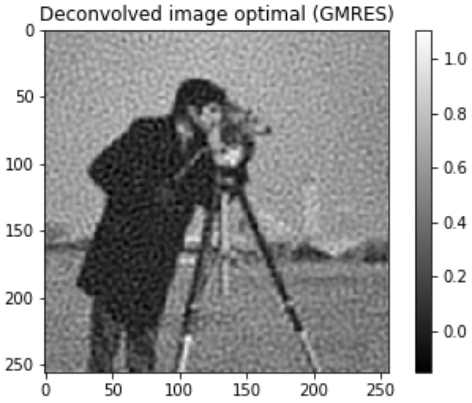


Figure 8: f_α with $\alpha = 0.0151$

which does seem to be slightly under-regularised.

3 Using a regularisation term based on the spatial derivative

3.1 a.)

For an $(N \times N)$ image g, the gradient operator $D = \begin{pmatrix} \nabla_x \\ \nabla_y \end{pmatrix}$ is constructed in a sparse matrix form using forward differences and a zero boundary condition:

$$(D_x g)_{i,j} = \begin{cases} g_{i+1,j} - g_{i,j}, & i < N \\ 0, & i = N \end{cases} \qquad (D_y g)_{i,j} = \begin{cases} g_{i,j+1} - g_{i,j}, & j < N \\ 0, & j = N \end{cases}$$

Firstly we form the regular forward difference operator $N \times N$ D_1:

$$D_1 = \begin{bmatrix} -1 & 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & -1 & 1 & \dots & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 0 & -1 & 1 \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 \end{bmatrix}$$

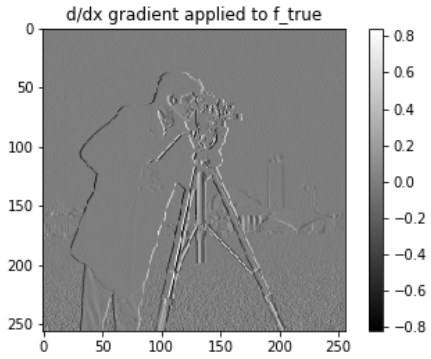
where we have enforced the boundary condition. Then note that we have for an $N \times N$ image g, then $D_1 g = D_x g$ and $g D_1 = D_y g$. Now we wish to form the large sparse $N^2 \times N^2$ matrix which can compute the gradient by taking the vectorised image and left multiplying a gradient operator, i.e. a matrix with two-non zero diagonals for each operator ∇_x and ∇_y. For each gradient this can be computed succinctly as:

$$\nabla_x = I \otimes D_1 \qquad \nabla_y = D_1 \otimes I$$

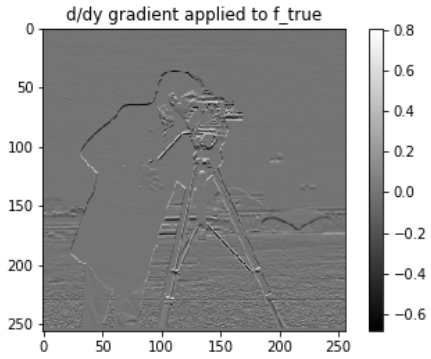
where I is the $N \times N$ identity matrix. The code to accomplish this is given below:

```
from scipy.sparse import spdiags
from scipy.sparse import eye
from scipy.sparse import kron
from scipy.sparse import vstack

x = np.vstack((-np.ones((N)), np.ones((N))))
D1 = spdiags(x, [0, 1], N, N)

# modification to enforce boundary conditions
D1 = scipy.sparse.lil_matrix(D1)
D1[-1,:] = 0

# form large sparse gradient operators
D1y2d = kron(D1, eye(N))
D1x2d = kron(eye(N), D1)
grad_op = vstack((D1x2d, D1y2d))
```

the results of applying the gradient operators to f_{true} are shown below:



(a) Gradient $\nabla_x f_{\text{true}}$



(b) Gradient $\nabla_y f_{\text{true}}$

3.2 b.)

The experiments formed in section 1 are repeated with the new regularisation term $\Psi(f_\alpha) = ||Df||_2$. This means the new normal equation is derived as follows:

$$
\begin{aligned}
f_\alpha &= \underset{f}{\operatorname{argmin}} \, ||Af - g||_2^2 + \alpha ||Df||_2^2 \\
\frac{\partial}{\partial f} ||Af - g||_2^2 + \alpha ||Df||_2^2 &= \frac{\partial}{\partial f} (Af - g)^T (Af - g) + \alpha (Df)^T (Df) \\
&= \frac{\partial}{\partial f} f^T A^T A f - 2 f^T A^T g + g^T g + \alpha f^T D^T D f \\
&= 2A^T Af + 2\alpha D^T Df - 2A^T g = 0 \\
\implies (A^T A + \alpha D^T D) f_\alpha &= A^T g
\end{aligned}
$$

and therefore for the GMRES method we simply form the sparse operator `ATA_DTD` and use the `gmres_solution` function as defined in 1c.:

```
# GMRES solution
ATA_DTD = lambda f, alpha: imblur(imblur(f.reshape((N,N)))) \
    + alpha*(grad_op.T @ grad_op @ f.reshape(N*N,1)).reshape(N,N)
f_alpha = gmres_solution(ALPHA, ATA_DTD)
```

For a value of $\alpha = 0.1$ this gave the solution f_α as below:

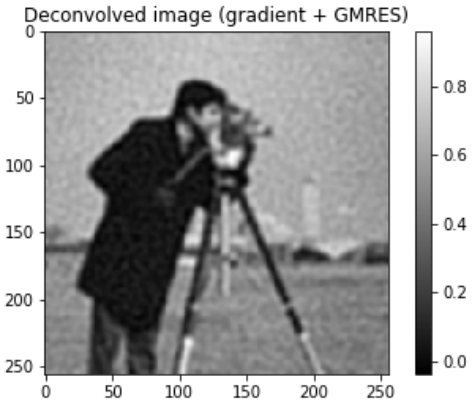


Figure 10: Deconvolution based on gradient regularsiation

For the LSQR method we note that we want to minimise

$$\boldsymbol{f}_{\alpha,D} = \underset{f}{\operatorname{argmin}} \left\| \begin{pmatrix} A \\ \sqrt{\alpha}\nabla_x \\ \sqrt{\alpha}\nabla_y \end{pmatrix} \boldsymbol{f} - \begin{pmatrix} \boldsymbol{g} \\ 0 \\ 0 \end{pmatrix} \right\|^2 \doteq \underset{f}{\operatorname{argmin}} ||M\boldsymbol{f} = \boldsymbol{b}||^2 \tag{1}$$

where:

- $\boldsymbol{f}$ is an $(N^2 \times 1)$ vector (solution image)
- $\boldsymbol{b}$ is a $(3N^2 \times 1)$ vector (augmented data)
- M is a $(3N^2 \times N^2)$ matrix

and therefore, as we did in 1d.), form the matrix M as a sparse linear operator capable of performing the operation Mf and M^Tb:

```
def M_f_grad(f, alpha):
    x = imblur(f.reshape((N,N))).reshape((N*N,1))
    y = np.sqrt(alpha)*(grad_op @ f.reshape(N*N,1))
    return np.vstack((x, y))

def MT_b_grad(b, alpha):
    mid = b.size//3
    x = imblur(b[:mid].reshape((N,N))).reshape((N*N,1))
    y = np.sqrt(alpha)*(grad_op.T @ b[mid:].reshape(2*N*N,1))
    return x + y

M = scipy.sparse.linalg.LinearOperator((3*N*N,N*N), matvec = partial(M_f_grad, alpha =
                                                    ALPHA),
                                       rmatvec = partial(MT_b_grad, alpha = ALPHA))
b = np.vstack([np.reshape(g,(g.size,1)),np.zeros((g.size,1)), np.zeros((g.size,1))])
```

and therefore the solution for $\alpha = 0.1$ is simply given by

```
f_alpha = scipy.sparse.linalg.lsqr(M,b)
```

which is visuliased below:

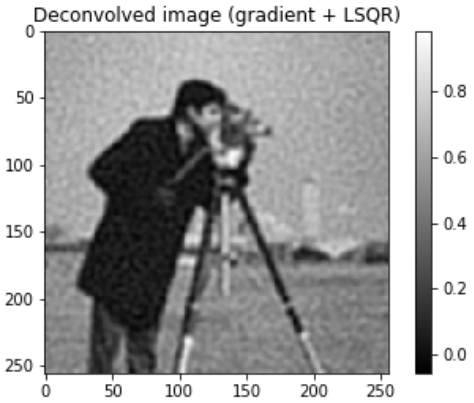
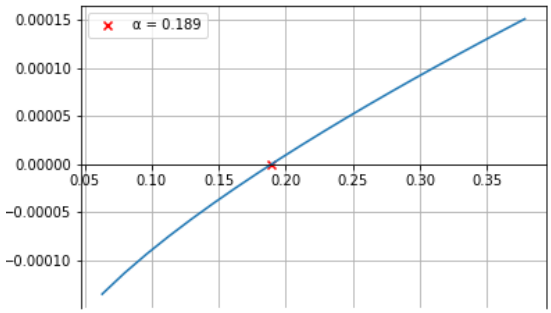


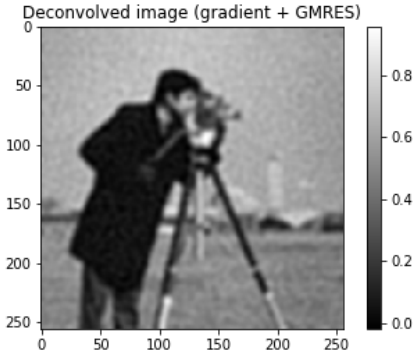
Figure 11: Deconvolution based on gradient regularsiation

3.3 c.)

We can also apply the discrepancy principle method as we did in 2i.) using the gradient operator as the regulariser, as well as the GMRES method to find the solution $||r_\alpha||^2$. Doing this we found the optimal α value to be $\alpha = 0.183$, and both DP solution and the f_{recon} solution are shown below:



(a) Discrepancy Solution



(b) GMRES solution $\alpha = 0.183$

4 Construct an anisotropic derivative filter

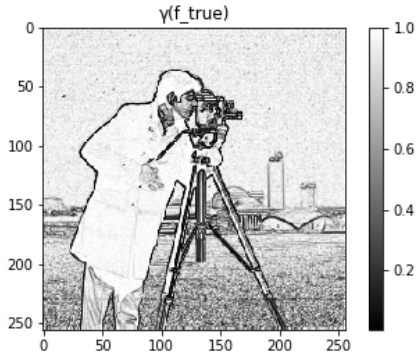
The Perona-Malik diffusivity function $\gamma(f) = \exp(-\sqrt{(\nabla_x f)^2 + (\nabla_y f)^2}/T$ is first contructed using the gradient operators defined before and setting the value of T as some fraction of the maximum absolute value of the image gradient. Specifically, we set $T = 0.2 * \max|Df|$. Note that it takes in an $(N \times N)$ image f and applies the filter pointwise - i.e outputs an $(N \times N)$ image.

```
def gamma(f):
  T = np.max(abs(grad_op @ f.reshape(N*N,1)))*0.2
  ga = np.exp(-np.sqrt((D1x2d @ f.reshape(N*N,1))**2 + (D1y2d @ f.reshape(N*N,1))**2)/T)
  return ga.reshape((N,N))
```

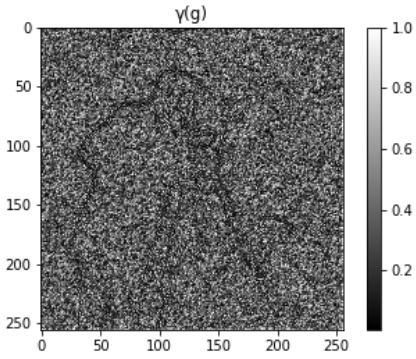
This $(N \times N)$ matrix is then shaped into a large sparse $(N^2 \times N^2)$ matrix with every element now on the main diagonal. This is the form of the Perona-Malik filter we want, and enables us to construct the full anisotropic filter via pointwise multiplication with the gradient operator matrices:

```
perona_malik_filter = spdiags((gamma(g).flatten()), [0], N*N, N*N)
anisotropic_filter = vstack((np.sqrt(perona_malik_filter) * D1x2d, np.sqrt(
                                        perona_malik_filter) * D1y2d))
```

The diffusivity of $\gamma(f_{\text{true}})$ and $\gamma(g)$ is shown in the plots below

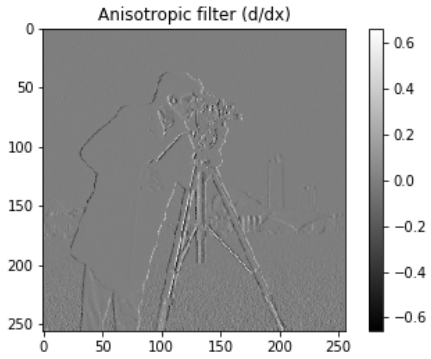


(a) $\gamma(f_{\text{true}})$

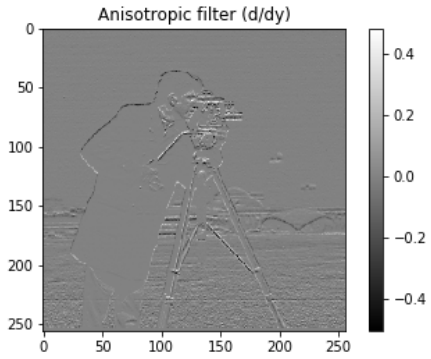


(b) $\gamma(g)$

Furthermore the anisotropic filter applied to f_{true} are also given-



(a) Gradient $\sqrt{\gamma} \cdot \nabla_x f_{\text{true}}$



(b) Gradient $\sqrt{\gamma} \cdot \nabla_y f_{\text{true}}$

We can then performing the deblurring of g using this anisotropic filter as the regulariser as well as the GMRES method for the solution of the system. Again, we note that since $\gamma(f)$ is considered 'fixed' the normal equations now take the form:

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

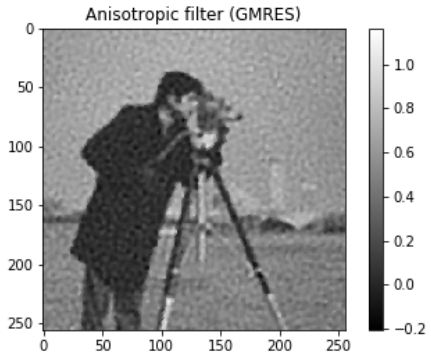
which we represent as an operator `ATA_G_DTD` taking a fixed value of $\gamma(f)$, an α parameter and an input image f:

```
ATA_G_DTD = lambda f, alpha, filt: imblur(imblur(f.reshape((N,N)))) \
  + alpha*(filt.T @ filt @ f.reshape(N*N,1)).reshape(N,N)
```

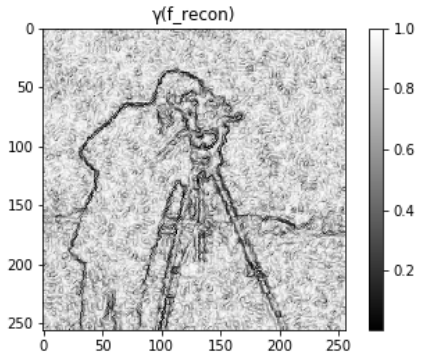
We can then form a sparse linear operator for a given value of $\gamma(f)$ and α and solve the normal equations once again using the GMRES method:

```
curr_filter = partial(ATA_G_DTD, alpha = ALPHA, filt = anisotropic_filter)
A = scipy.sparse.linalg.LinearOperator((N*N,N*N), curr_filter)
ATg = imblur(g).reshape(N*N, 1)
gmresOutput = scipy.sparse.linalg.gmres(A, ATg)
```

This deblurring procedure produced the solution f_{recon} and diffusivity $\gamma(f_{\text{recon}})$ below:



(a) GMRES solution, $\alpha = 0.1$



(b) Diffusitiy $\gamma(f_{\text{recon}})$

5 Iterative deblurring

For the iterative deblurring task, we follow the algorithm as given. The adaptation from Task 4 is to simply recompute $\gamma(f_i)$ every iteration, form the operator `ATA_G_DTD` using this new value of γ and solve the corresponding normal equation using the GMRES method. We can define a function `iterative_deblur` that

does precisely this. The stopping condition is when the norm of the difference between forward projection of the deblurred image Af_i and the original data g stops changing by some small amount; i.e. if

$$\texttt{tol} > ||Af_i - g||_2$$

for e.g $\texttt{tol} = 0.001$

```
def iterative_deblur(g, alp):

    f_i = np.copy(g)
    diff = 1
    alp = 0.2

    while(diff > 0.001):

        # constuct matrix (ATA + 2*alpha*D^T gamma D)
        perona_i = spdiags((gamma(f_i).flatten()), [0], N*N, N*N)
        filter_i = vstack((np.sqrt(perona_i) * D1x2d, np.sqrt(perona_i) * D1y2d))
        curr_filter = partial(ATA_G_DTD, alpha = alp, filt = filter_i)

        A = scipy.sparse.linalg.LinearOperator((N*N,N*N), curr_filter)
        ATg = imblur(g).reshape(N*N, 1)

        f_i = scipy.sparse.linalg.gmres(A, ATg)[0].reshape((N,N))
        diff = np.linalg.norm(imblur(f_i) - g)
```

We then run this deblurring function, given a specific value of $\alpha = 0.15$ and the tolerance given above, which produced the sequence of iterative solutions and diffusivities:

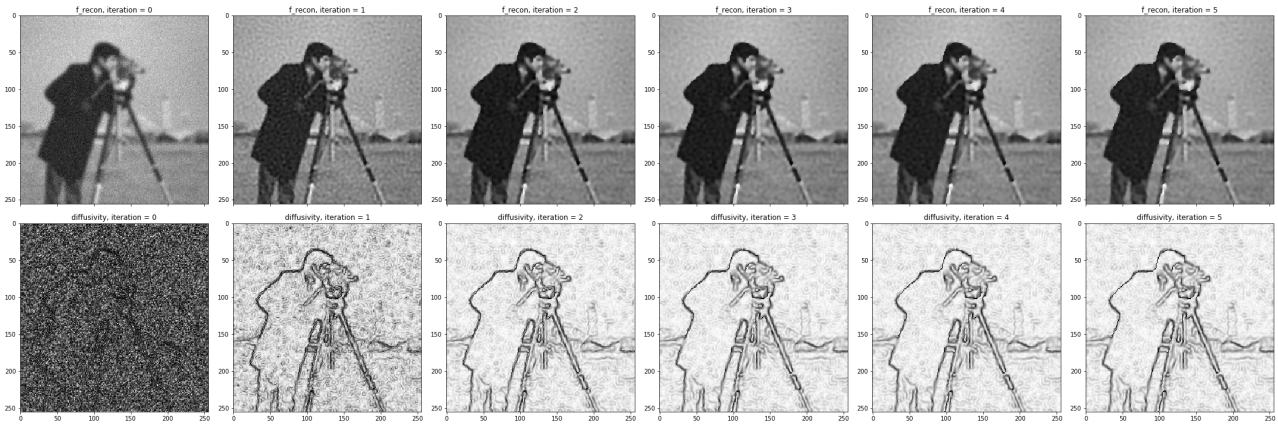


Figure 16: Iterative deblurring

Note how the algorithm essentially starts to produce its best solution after just 3 iterations, and any improvements after that are almost undetectable.