Computational and Variational Methods for Inverse Problems - Homework 4 Solution

Mohammad Afzal Shadab (ms82697)

mashadab@utexas.edu

Due Monday, December 10, 2021

---

1 Inverse problem for Burgers' equation

Consider the inverse problem for the viscosity field $m(x)$ in the one-dimensional Burgers' equation

$$u_t + uu_x - (mu_x)_x = f \quad \text{in } (0,L)\times(0,T) \tag{1.1}$$
$$u(0,t) = u(L,t) = 0 \quad \text{for all } t \in [0,T] \tag{1.2}$$
$$u(x,0) = 0 \quad \text{for all } x \in [0,L] \tag{1.3}$$

Given observations $d = d(x,t)$ for $t \in [T_1, T]$, where $T_1 > 0$, we invert for the viscosity field $m = m(x)$ by solving the minimisation problem

$$\min_{m\in\mathcal{M}} \mathcal{F}(m) := \frac{1}{2}\int_{T_1}^{T}\int_0^L (u-d)^2 \, dx\, dt + \frac{\beta}{2}\int_0^L \left(\frac{\mathrm{d}m}{\mathrm{d}x}\right)^2 dx \tag{1.4}$$

where the space $\mathcal{M}$ is defined as $\mathcal{M} = H^1(0,L)$, regularization parameter $\beta > 0$.

1.1 Weak form of the forward problem

The weak form of the forward problem can be found by integrating with a test function $p(x,t) : [0,L]\times[0,T] \to \mathbb{R}$. Therefore, the solution space $\mathcal{U}$ considered is

$$\mathcal{U} = \{u \in H^1(0,L)\times L^2(0,T) : u(0,t) = u(L,t) = 0 \text{ and } u(x,0) = 0\} \tag{1.5}$$

Similarly, the space for the adjoint variable is $\mathcal{P}$ given by

$$\mathcal{P} = \{p \in H^1[0,L]\times L^2[0,T] : p(0,t) = p(L,t) = 0\} \tag{1.6}$$

The homogeneous Dirichlet boundary conditions gave been enforced in the weak forms. Integrating the PDE against p yields

$$\int_0^T\int_0^L p\left(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \frac{\partial}{\partial x}\left(m\frac{\partial u}{\partial x}\right) - f\right) dx\, dt = 0 \tag{1.7}$$

Now, performing the integration by parts of the diffusion term, the "weak form" of the forward problem can be yielded as follows:

$$\boxed{\text{Find } u \in \mathcal{U} \text{ such that: } \int_0^T\int_0^L p\left(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - f\right) + \frac{\partial p}{\partial x}\left(m\frac{\partial u}{\partial x}\right) dx\, dt = 0 \quad \forall p \in \mathcal{P}} \tag{1.8}$$

with homogeneous Dirichlet boundary conditions $p(0,t) = p(L,t) = 0$

1.2 Adjoint and gradient of the Lagrangian $\mathcal{L}$

Let's first form the lagrangian by clubbing the weak form of the forward problem (1.8) and the objective function (1.4) together.

$$\mathcal{L}(u,p,m) = \frac{1}{2}\int_{T_1}^{T}\int_0^L (u-d)^2\, dx\, dt + \frac{\beta}{2}\int_0^L \left(\frac{\mathrm{d}m}{\mathrm{d}x}\right)^2 dx + \int_0^T\int_0^L p\left(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - f\right) + \frac{\partial p}{\partial x}\left(m\frac{\partial u}{\partial x}\right) dx\, dt \tag{1.9}$$

(a) Adjoint equation, $\delta_u \mathcal{L} = 0 \; \forall \hat{u} \in \mathcal{U}$

The weak form of adjoint equation can be found by taking the Frechet derivative with respect to u.

$$\boxed{\text{Find } p \in \mathcal{P} \text{ such that: } \delta_u \mathcal{L} = \int_{T_1}^{T}\int_{0}^{L} \hat{u}(u-d)\, dx\, dt + \int_{0}^{T}\int_{0}^{L} p\frac{\partial \hat{u}}{\partial t} + p\hat{u}\frac{\partial u}{\partial x} + pu\frac{\partial \hat{u}}{\partial x} + \frac{\partial p}{\partial x} m \frac{\partial \hat{u}}{\partial x}\, dx\, dt = 0 \quad \forall \hat{u} \in \mathcal{U}} \tag{1.10}$$

The strong form of the adjoint equation is derived by using integration by parts

$$\int_0^T\int_0^L \mathbb{1}_{[T_1,T]}(t)\hat{u}(u-d)\, dx\, dt + \int_0^T\int_0^L \hat{u}\left[-\frac{\partial p}{\partial t} + p\frac{\partial u}{\partial x} - \frac{\partial}{\partial x}(pu) - \frac{\partial}{\partial x}\left(m\frac{\partial p}{\partial x}\right)\right] dx\, dt + \int_0^L [p\hat{u}]_0^T\, dx + \int_0^T [pu\hat{u}]_0^L\, dt + \int_0^T \left[m\frac{\partial p}{\partial x}\hat{u}\right]_0^L dt = 0$$

where $\mathbb{1}_{[T_1,T]}(t)$ is a function defined as

$$\mathbb{1}_{[T_1,T]}(t) = \begin{cases} 1, & t \in [T_1, T] \\ 0, & \text{else} \end{cases} \tag{1.11}$$

In this case, all the boundary term disappear due to the homogeneous Dirichlet boundary conditions. The initial conditions are also $u(x,0) = 0$. Furthermore, using

$$p\frac{\partial u}{\partial x} - \frac{\partial}{\partial x}(pu) = p\frac{\partial u}{\partial x} - u\frac{\partial p}{\partial x} - p\frac{\partial u}{\partial x} = -u\frac{\partial p}{\partial x} \tag{1.12}$$

and the arbitrariness of $\hat{u}$ in the spatio-temporal domain, the strong form of the adjoint equation can finally be written as follows

$$\boxed{\begin{aligned} -\frac{\partial p}{\partial t} - \frac{\partial p}{\partial x}u - \frac{\partial}{\partial x}\left(m\frac{\partial p}{\partial x}\right) &= -\mathbb{1}_{[T_1,T]}(t)(u-d) \quad \text{in } (0,L)\times(0,T) \\ p(0,t) = p(L,t) &= 0 \quad \text{for all } t \in [0,T] \\ p(x,T) &= 0 \quad \text{for all } t \in [0,T] \end{aligned}}$$

(b) Gradient w.r.t. m, $\delta_m \mathcal{L} = \langle \mathcal{G}(m), \hat{m} \rangle \; \forall \hat{m} \in \mathcal{M}$

Taking the Frechet derivative of the Lagrangian (1.9) w.r.t. m

$$\boxed{\langle \mathcal{G}(m), \hat{m} \rangle = \beta \int_0^L \frac{\mathrm{d}m}{\mathrm{d}x}\frac{\mathrm{d}\hat{m}}{\mathrm{d}x}\, dx + \int_0^T\int_0^L \hat{m}\frac{\partial p}{\partial x}\frac{\partial u}{\partial x}\, dx\, dt \quad \forall \hat{m} \in \mathcal{M}} \tag{1.13}$$

To obtain the strong form, let's perform integration by parts on the first term on RHS,

$$\langle \mathcal{G}(m), \hat{m} \rangle = \int_0^L \hat{m}\left(-\beta\frac{\mathrm{d}^2 m}{\mathrm{d}x^2} + \int_0^T \frac{\partial p}{\partial x}\frac{\partial u}{\partial x} dt\right) dx + \left[\beta\frac{\mathrm{d}m}{\mathrm{d}x}\hat{m}\right]_0^L$$

By using the arbitrariness of $\hat{m}$ in the spatio-temporal domain, we finally get the strong form of the gradient

$$\boxed{\mathcal{G}(m) = \begin{cases} -\beta\dfrac{\mathrm{d}^2 m}{\mathrm{d}x^2} + \displaystyle\int_0^T \frac{\partial p}{\partial x}\frac{\partial u}{\partial x} dt & (x,t) \in (0,L)\times(0,T) \\ -\beta\dfrac{\mathrm{d}m}{\mathrm{d}x}\Big|_{x=0} & x = 0 \\ \beta\dfrac{\mathrm{d}m}{\mathrm{d}x}\Big|_{x=L} & x = L \end{cases}}$$

2 Inverse advection-diffusion inverse problem

2.1 Hessian action

Here we consider an inverse problem for advection-diffusion-reaction equation, on the domain $\Omega = [0,1] \times [0,1]$:

$$\min_{m \in H^1(\Omega)} \mathcal{F}(m) := \frac{1}{2} \int_\Omega (u(m) - d)^2 \, dx + \frac{\beta}{2} \int_\Omega |\nabla m|^2 \, dx \tag{2.1}$$

subject to the PDE constraint

$$-\nabla \cdot (k \nabla u) + \boldsymbol{v} \cdot \nabla u + 100 \exp(m) u^3 = f \quad \text{in } \Omega \tag{2.2}$$

$$u = 0 \quad \text{on } \Omega \tag{2.3}$$

We take $f = \max\{0.5, \exp(-25(x-0.7)^2 - 25(y-0.7)^2)\}$, $k = 1$ and $\boldsymbol{v} = (1,0)^T$. The "true" reaction coefficient field m is defined as

$$m(x,y) = \begin{cases} 4 & \text{if } (x-0.5)^2 + (y-0.5)^2 < 0.2^2 \\ 8 & \text{otherwise} \end{cases} \tag{2.4}$$

Data is generated by adding noise of standard deivation of 0.01 in true values of the parameters.

Let's first define the Lagrangian

$$\mathcal{L}(u,p,m) = \frac{1}{2} \int_\Omega (u(m) - d)^2 \, dx + \frac{\beta}{2} \int_\Omega |\nabla m|^2 \, dx + \int_\Omega k \nabla u \cdot \nabla p + p \boldsymbol{v} \cdot \nabla u + 100 \exp(m) u^3 p - f p \, dx$$

where the weak form of the PDE has been derived from integration by parts against p, and the boundary terms are nil due to the homogeneous boundary condition. We then proceed to derive the gradient as follows

1. Forward problem:

$$\text{Find } u \in H_0^1(\Omega) : \delta_p \mathcal{L} = \int_\Omega k \nabla u \cdot \nabla \hat{p} + \hat{p} \boldsymbol{v} \cdot \nabla u + 100 \exp(m) u^3 \hat{p} - f \hat{p} \, dx = 0 \quad \forall \hat{p} \in H_0^1(\Omega) \tag{2.5}$$

2. Adjoint problem:

$$\text{Find } p \in H_0^1(\Omega) : \delta_u \mathcal{L} = \int_\Omega \hat{u}(u-d) \, dx + \int_\Omega k \nabla \hat{u} \cdot \nabla p + p v \cdot \nabla \hat{u} + 300 \exp(m) u^2 p \hat{u} \, dx = 0 \quad \forall \hat{u} \in H_0^1(\Omega) \tag{2.6}$$

3. Gradient evaluation w.r.t. m:

$$\mathcal{G}(m; \hat{m}) = \delta_m \mathcal{L} = \beta \int_\Omega \nabla m \cdot \nabla \hat{m} + \int_\Omega 100 \exp(m) \hat{m} u^3 p \, dx \tag{2.7}$$

For deriving the Hessian action in direction $\tilde{m}$, let's define a new Lagrangian to enforce the forward and adjoint equations

$$\mathcal{L}^H(u,p,m,\tilde{u},\tilde{p},\tilde{m}) = \delta_m \mathcal{L}(\tilde{m}) + \delta_p \mathcal{L}(\tilde{p}) + \delta_u \mathcal{L}(\tilde{u}) \tag{2.8}$$

By substituting the gradient expressions from (2.5), (2.6) and (2.7) in (2.8), we get

$$\begin{aligned} \mathcal{L}^H(u,p,m,\tilde{u},\tilde{p},\tilde{m}) = \beta \int_\Omega \nabla m \cdot \nabla \tilde{m} + \int_\Omega 100 \exp(m) \tilde{m} u^3 p \, dx + \int_\Omega k \nabla u \cdot \nabla \tilde{p} + \tilde{p} \boldsymbol{v} \cdot \nabla u + 100 \exp(m) u^3 \tilde{p} - f \tilde{p} \, dx \\ + \int_\Omega \tilde{u}(u-d) \, dx + \int_\Omega k \nabla \tilde{u} \cdot \nabla p + p v \cdot \nabla \tilde{u} + 300 \exp(m) u^2 p \tilde{u} \, dx \end{aligned} \tag{2.9}$$

For the second variations, using the calculus of variations $\mathcal{L}^H$ for variable $\alpha \to \alpha + \epsilon \hat{\alpha}$ and performing $\mathrm{d}/\mathrm{d}\epsilon$ with limit $\epsilon \to 0$:

1. **Incremental forward problem**: $\delta_p \mathcal{L}^H = 0 \; \forall \tilde{p}$

$$\boxed{\begin{aligned}\text{Find } \tilde{u} \in H_0^1(\Omega) : \; \delta_p \mathcal{L}^H = \int_\Omega & 100\exp(m)\tilde{m}u^3\hat{p}\,dx + k\nabla\tilde{u}\cdot\nabla\hat{p} \\ & + \hat{p}v\cdot\nabla\tilde{u} + 300\exp(m)u^2\hat{p}\tilde{u}\,dx = 0 \quad \forall \hat{p} \in H_0^1(\Omega)\end{aligned}} \tag{2.10}$$

For strong form, performing integration by parts on the second term on RHS to get $\hat{p}$ out and using the arbitrariness of $\hat{p}$ on Ω, we get the strong form

$$\boxed{\begin{aligned}100\exp(m)\tilde{m}u^3 - \int_\Omega k\Delta\tilde{u} + v\cdot\nabla\tilde{u} + 300\exp(m)u^2\tilde{u} = 0 \quad & \text{in } \Omega \qquad (2.11)\\ \tilde{u} = 0 \quad & \text{on } \partial\Omega \qquad (2.12)\end{aligned}}$$

2. **Incremental adjoint problem**: $\delta_u \mathcal{L}^H = 0 \; \forall \tilde{u}$

$$\boxed{\begin{aligned}\text{Find } \tilde{p} \in H_0^1(\Omega) : \; \delta_u \mathcal{L}^H = \int_\Omega & 300\exp(m)\tilde{m}u^2\hat{u}p\,dx + \int_\Omega k\nabla\hat{u}\cdot\nabla\tilde{p}\,dx + \int_\Omega \tilde{p}\boldsymbol{v}\cdot\nabla\hat{u}\,dx \\ & + \int_\Omega 300\exp(m)u^2\hat{u}\tilde{p}\,dx + \int_\Omega \tilde{u}\hat{u}\,dx + \int_\Omega 600\exp(m)up\tilde{u}\hat{u}\,dx = 0 \\ & \forall \hat{u} \in H_0^1(\Omega)\end{aligned}}$$

(2.13)

Again, integration by parts of the second and third terms and arbitrariness of $\hat{u}$ on Ω gives the strong form

$$\boxed{\begin{aligned}300\exp(m)\tilde{m}u^2p - k\Delta\tilde{p} - \nabla\cdot(\tilde{p}\boldsymbol{v}) + 300\exp(m)u^2\tilde{p} + \tilde{u} + 600\exp(m)up\tilde{u} = 0 \quad & \text{in } \Omega \qquad (2.14)\\ \tilde{p} = 0 \quad & \text{on } \partial\Omega \qquad (2.15)\end{aligned}}$$

3. **Hessian action**: We wish to derive the action of the Hessian in direction $\tilde{m}$. Using the calculus of variations the gradient is $m \to m + \epsilon\hat{m}$ and performing $\mathrm{d}/\mathrm{d}\epsilon$ with $\epsilon \to 0$, we get the weak form of the Hessian action $\mathcal{H}(\tilde{m})\hat{m} = \delta_m \mathcal{L}^H$

$$\boxed{\begin{aligned}\mathcal{H}(\tilde{m})\hat{m} = \int_\Omega & \beta\nabla\tilde{m}\cdot\nabla\hat{m} + 100\exp(m)\hat{m}\tilde{m}u^3p + 300\exp(m)\hat{m}\tilde{u}u^2p \\ & + 100\exp(m)\hat{m}u^3\tilde{p}\;\,dx \quad \forall \hat{m} \in H^1(\Omega)\end{aligned}} \tag{2.16}$$

Finally, one last integration by parts gives

$$\begin{aligned}\mathcal{H}(\tilde{m})\hat{m} = \int_\Omega & -\beta\hat{m}\Delta\tilde{m} + 100\exp(m)\hat{m}\tilde{m}u^3p + 300\exp(m)\hat{m}\tilde{u}u^2p \\ & + 100\exp(m)\hat{m}u^3\tilde{p}\;\,dx + \int_\Omega \hat{m}\beta\nabla\tilde{m}\cdot n\,\mathrm{d}s\end{aligned} \tag{2.17}$$

From the arbitrariness of $\hat{m}$ on Ω gives the strong form

$$\boxed{\mathcal{H}(\tilde{m}) = \begin{cases}-\beta\Delta\tilde{m} + 100\exp(m)\tilde{m}u^3p + 300\exp(m)\tilde{u}u^2p + 100\exp(m)u^3\tilde{p} & \text{in } \Omega \\ \beta\nabla\tilde{m}\cdot n & \text{on } \partial\Omega\end{cases}}$$

2.2 Solving the ADR inverse problem using FEniCS implemented with inexact Newton and Gauss-Newton methods

In all the inversions in the remainder of this assignment, the tolerance and maximum iterations for optimization are respectively set to 5×10^{-7} and 20 because of feasibility of

the simulations. Although the codes for the solution codes are enclosed at the end of this PDF, they can be found in the jupyter notebook 'HW5_Q2final.ipynb'.

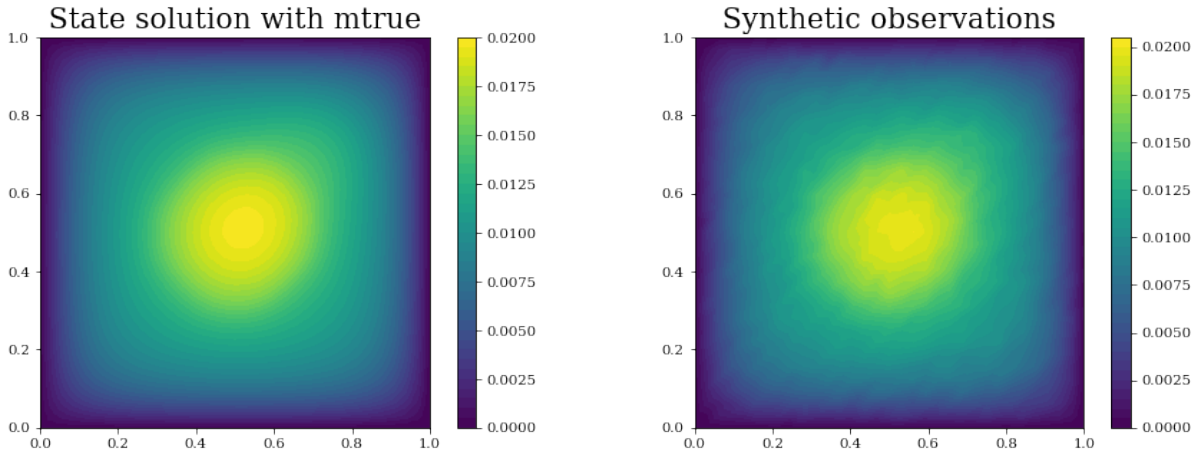


Figure 1: The true solution and noisy solution after adding 1% noise.

Table 1: Summary of all the tests performed with Newton and CG steps

Part	N	Noise SD	β	Regularization	Newton steps	CG steps
4.2	20	1%	1.00E-07	Tikhonov	4	6
			1.00E-08		6	15
			1.00E-09		5	17
			1.00E-10		5	21
			1.00E-11		5	21
	40		1.00E-07		5	11
			1.00E-08		5	15
			1.00E-09		5	16
			1.00E-10		5	21
	80		1.00E-07		5	11
			1.00E-08		5	15
			1.00E-09		5	17
			1.00E-10		5	22
4.3	40	10%	1.00E-07		5	11
			1.00E-08		5	15
			1.00E-09		5	17
			1.00E-10		5	22
4.4	20	1%	1.00E-08	TV	6	14
			1.00E-09		5	17
			1.00E-10		5	22
			1.00E-11		5	21
	40		1.00E-08		5	15
			1.00E-09		5	17
			1.00E-10		5	22
			1.00E-11		5	21
	80		1.00E-08		5	15
			1.00E-09		5	17
			1.00E-10		5	22
			1.00E-11		5	20

Using visual inspection, for 20×20 (see Figure 2) mesh the optimal β is close to 10^{-8} because of less fluctuations. Whereas for 40×40 (see Figure 3) and 80×80 (see Figure 4) both, it comes out to be $\beta = 10^{-8}$. Also, from Table 1, Newton becomes mesh independent with converging in 5 steps. However, CG steps increase with mesh size for small $\beta = 10^{-7}$ but change is minute for large β.

2.3 Optimal regularization and relative performance of nonlinear (Newton/-Gauss Newton) and total linear (CG) iterations

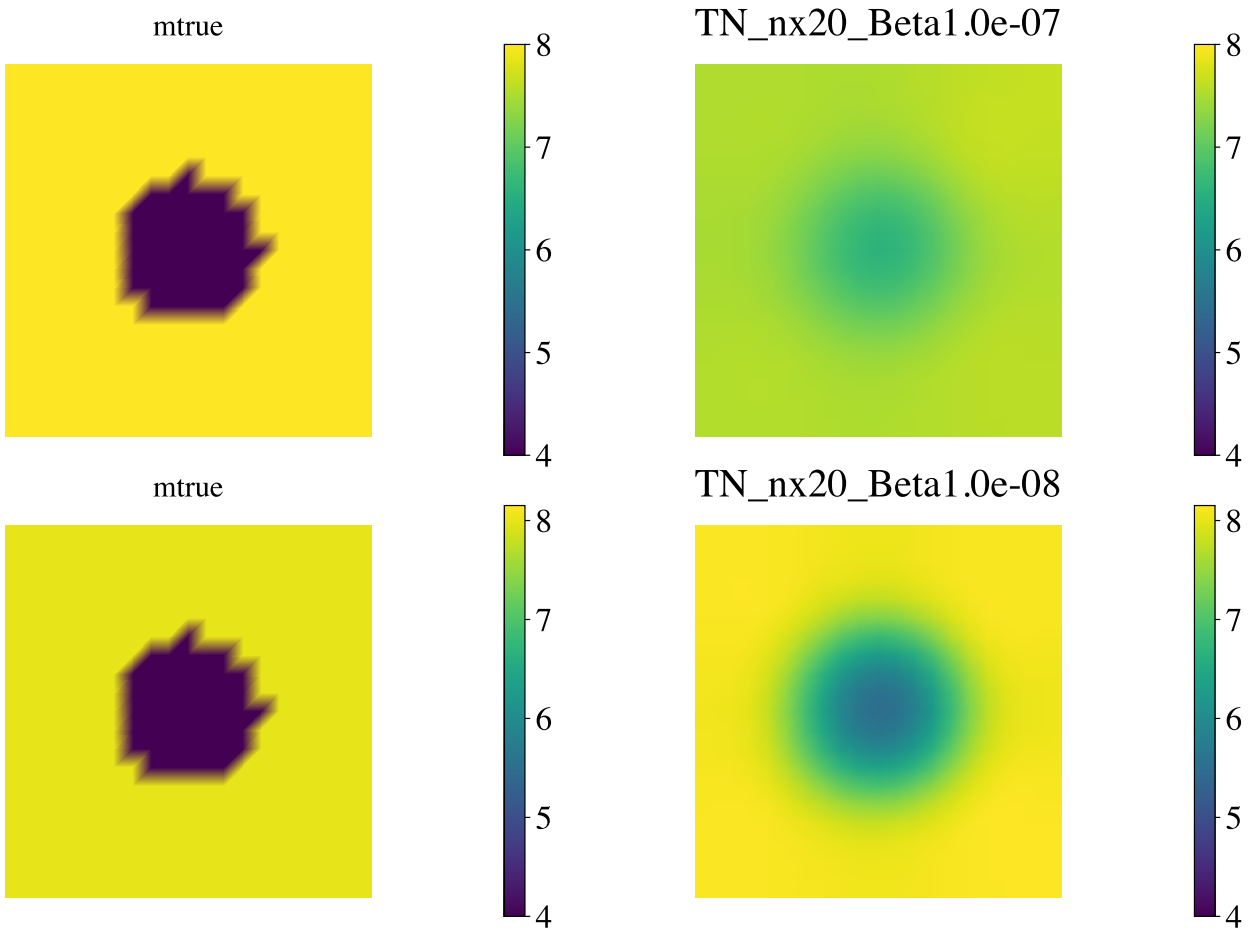
The case of 1% noise has been shown in previous section. So, we will only show for 10%.

Using visual inspection, for 40×40 mesh the optimal β is close to 10^{-8} whereas for 1% noise is (see Figure 3). Whereas for 10% noise (Figure 5), it comes out to be $\beta = 10^{-8}$. Also, from Table 1, Newton steps are same for both cases (=5) whereas CG steps for 1% noise is and 10% noise are same as well (=15).

2.4 Total variation regularization $\delta = 0.001$

Using visual inspection, for 20×20 (see Figure 6), 40×40 (see Figure 7) and 80×80 (see Figure 8) mesh sizes, the optimal β is close to 10^{-8} because of less fluctuations. Also, the inverted field is much sharper than Tikhonov case. Also, from Table 1, Newton becomes mesh independent with converging in 5 steps. However, CG steps change slightly with mesh size.

For 20 $\times$20 grid:



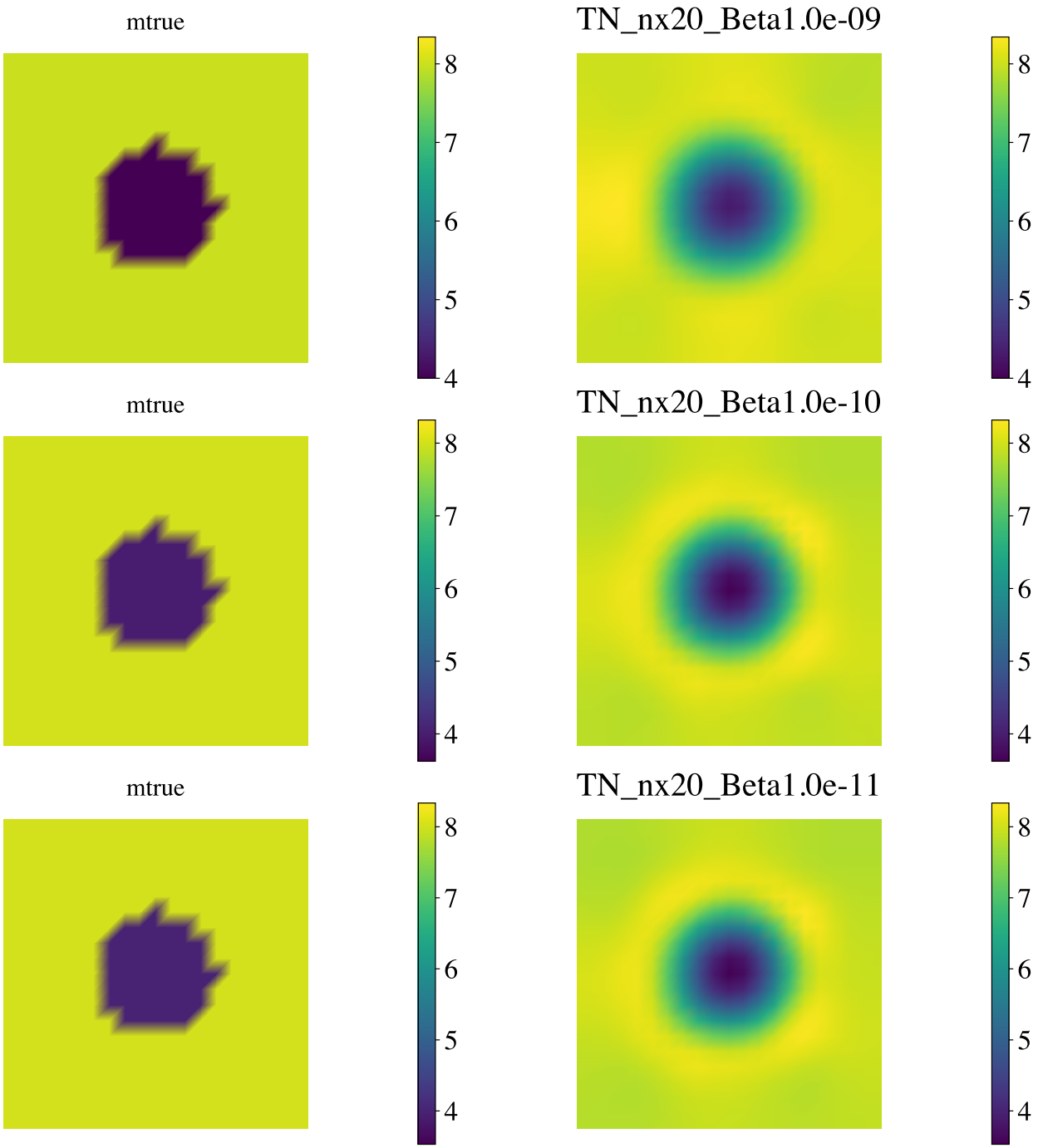


Figure 2: The inversion results for the viscosity field $m(\boldsymbol{x})$ for the cases $\beta = \{10^{-7}, 10^{-8}, 10^{-9}, 10^{-10}, 10^{-11}\}$ and $n = 20 \times 20$ after adding 1% noise.

For 40 $\times$40 and 80 $\times$80 grids:

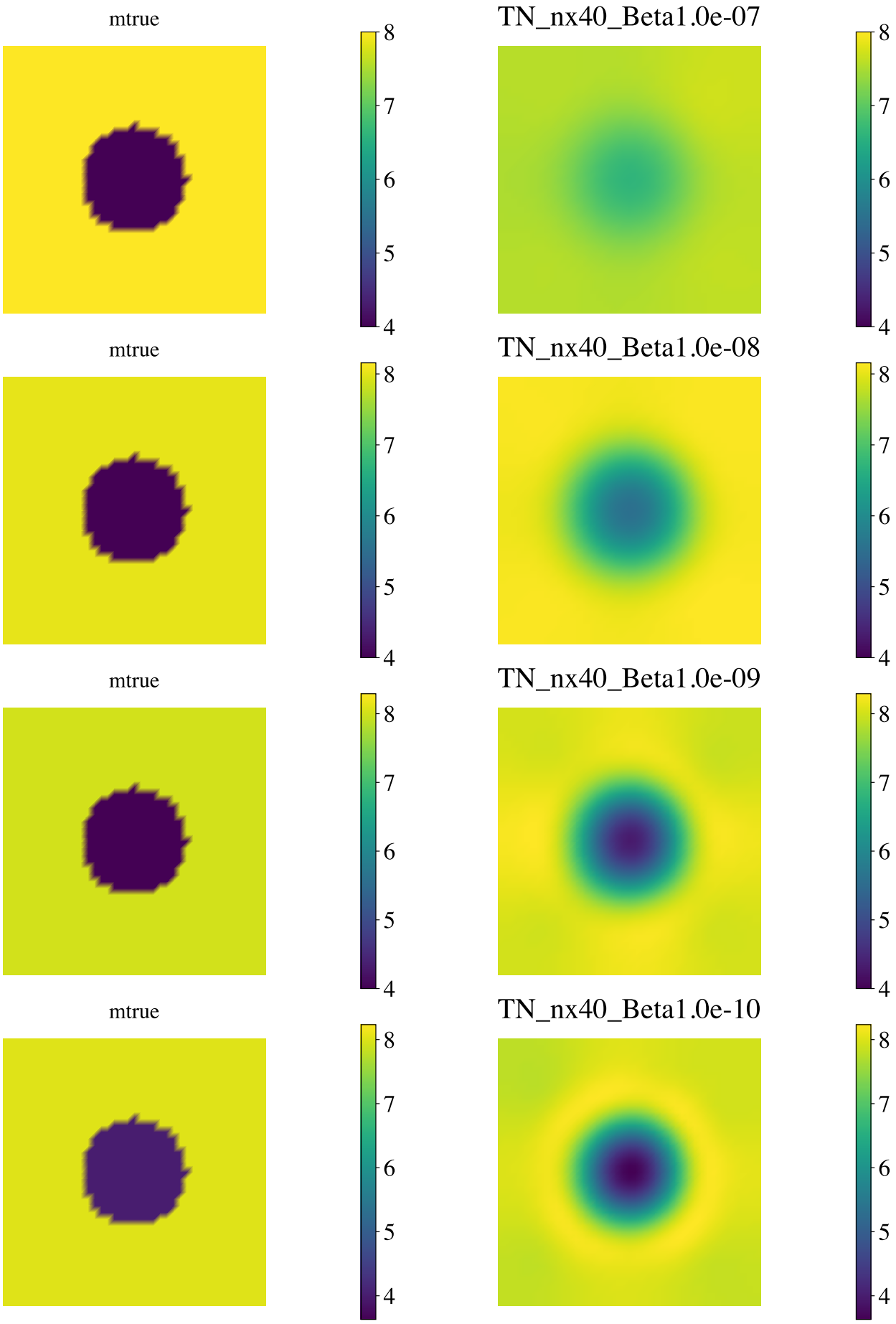


Figure 3: The inversion results for the viscosity field $m(\boldsymbol{x})$ for the cases $\beta = \{10^{-7}, 10^{-8}, 10^{-9}, 10^{-10}\}$ and $n = 40 \times 40$ after adding 1% noise.

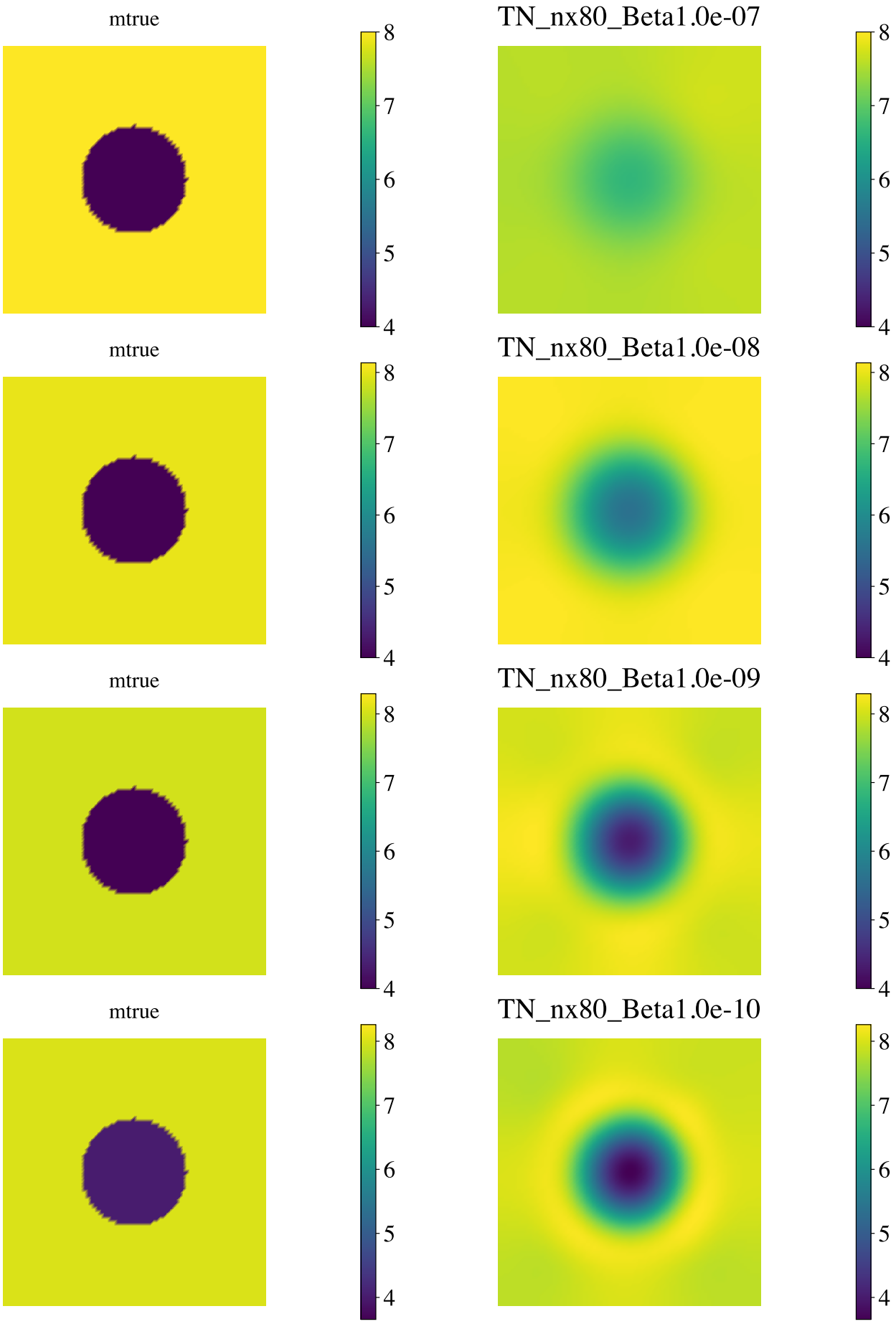


Figure 4: The inversion results for the viscosity field $m(\boldsymbol{x})$ for the cases $\beta = \{10^{-7}, 10^{-8}, 10^{-9}, 10^{-10}\}$ and $n = 80 \times 80$ after adding 1% noise.

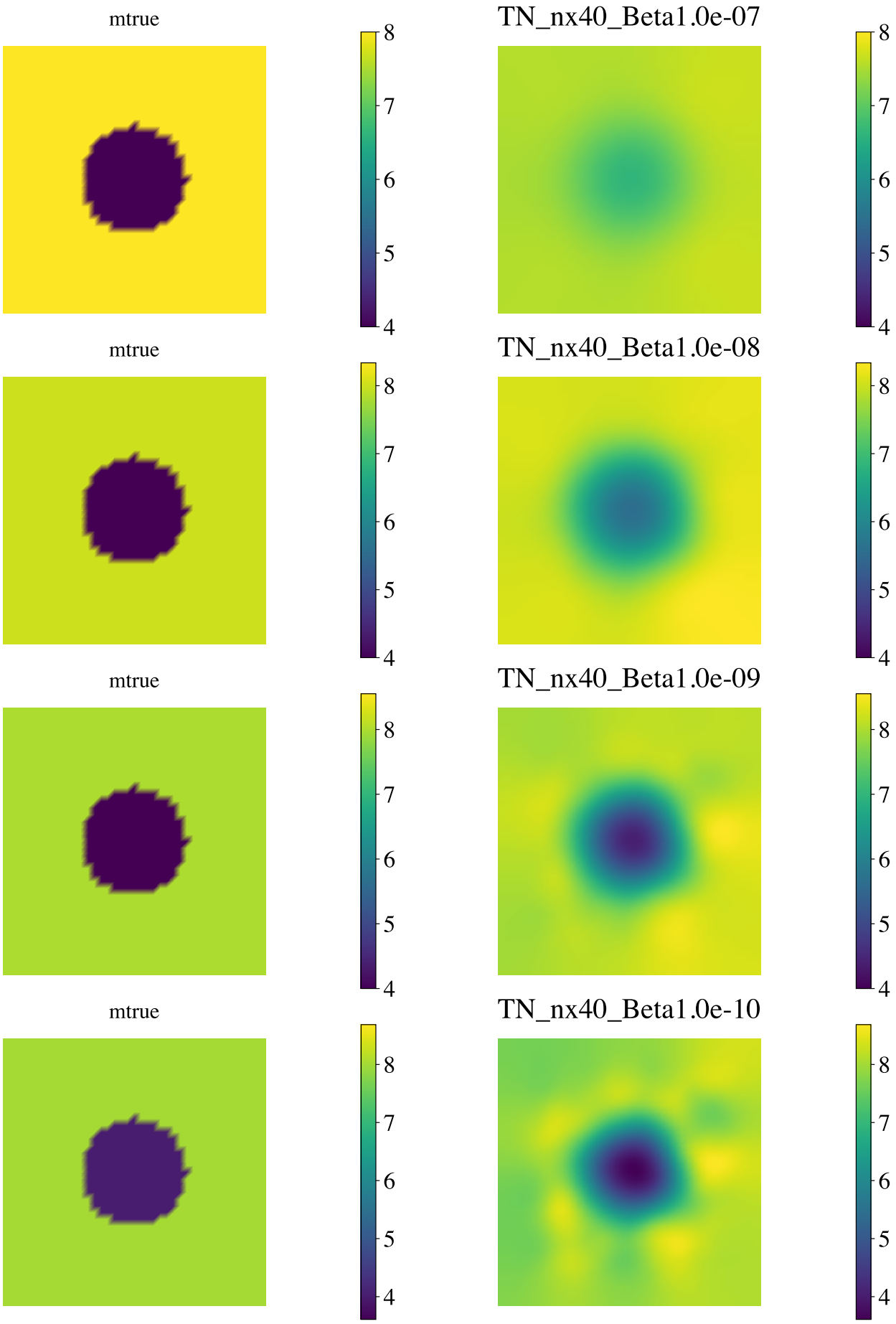


Figure 5: The inversion results for the viscosity field $m(\boldsymbol{x})$ for the cases $\beta = \{10^{-7}, 10^{-8}, 10^{-9}, 10^{-10}\}$ and $n = 40 \times 40$ after adding 10% noise.

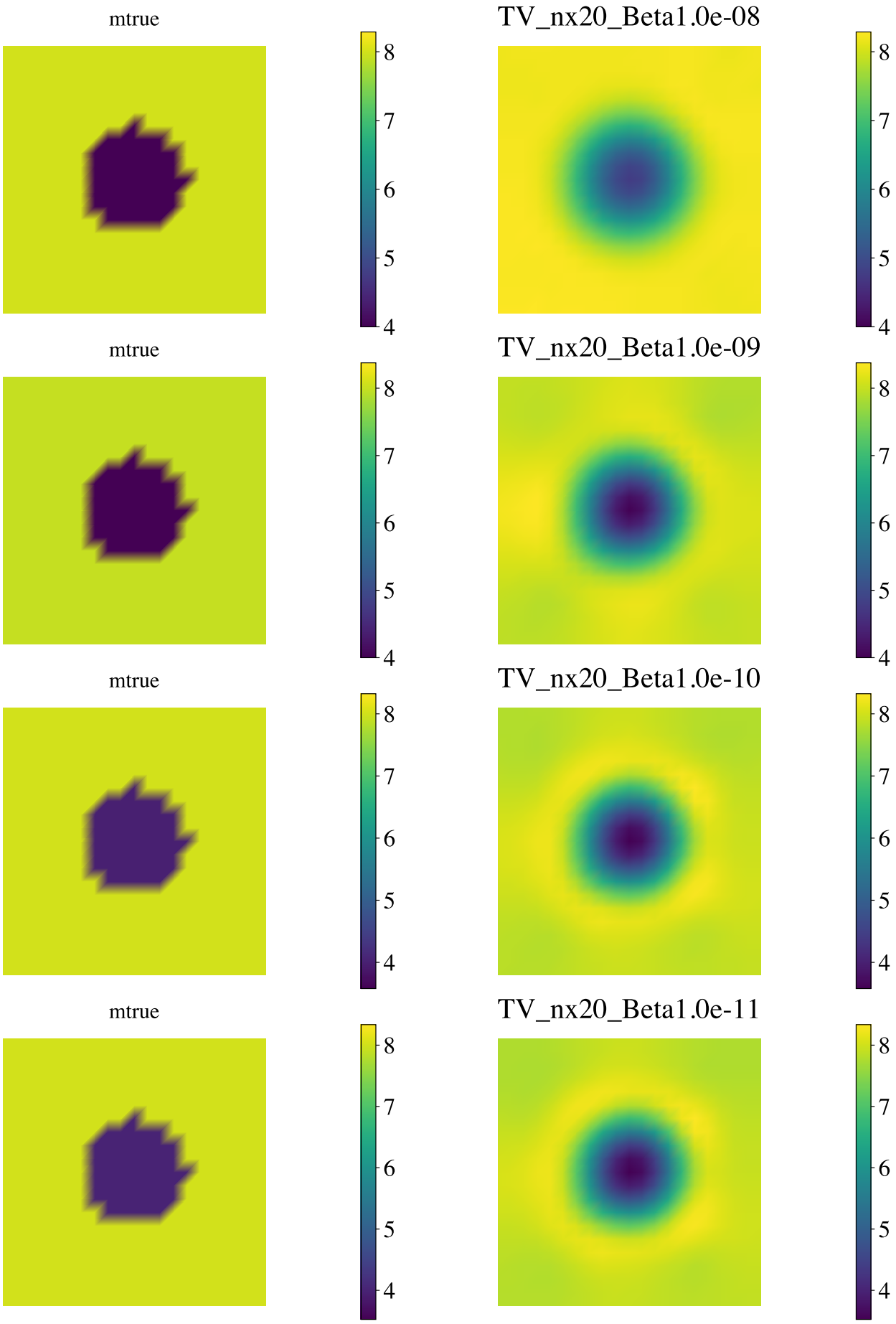


Figure 6: The inversion results for the viscosity field $m(\boldsymbol{x})$ for the cases $\beta = \{10^{-8}, 10^{-9}, 10^{-10}, 10^{-11}\}$ and $n = 20 \times 20$ after adding 1% noise. Total variation regularization has been used.

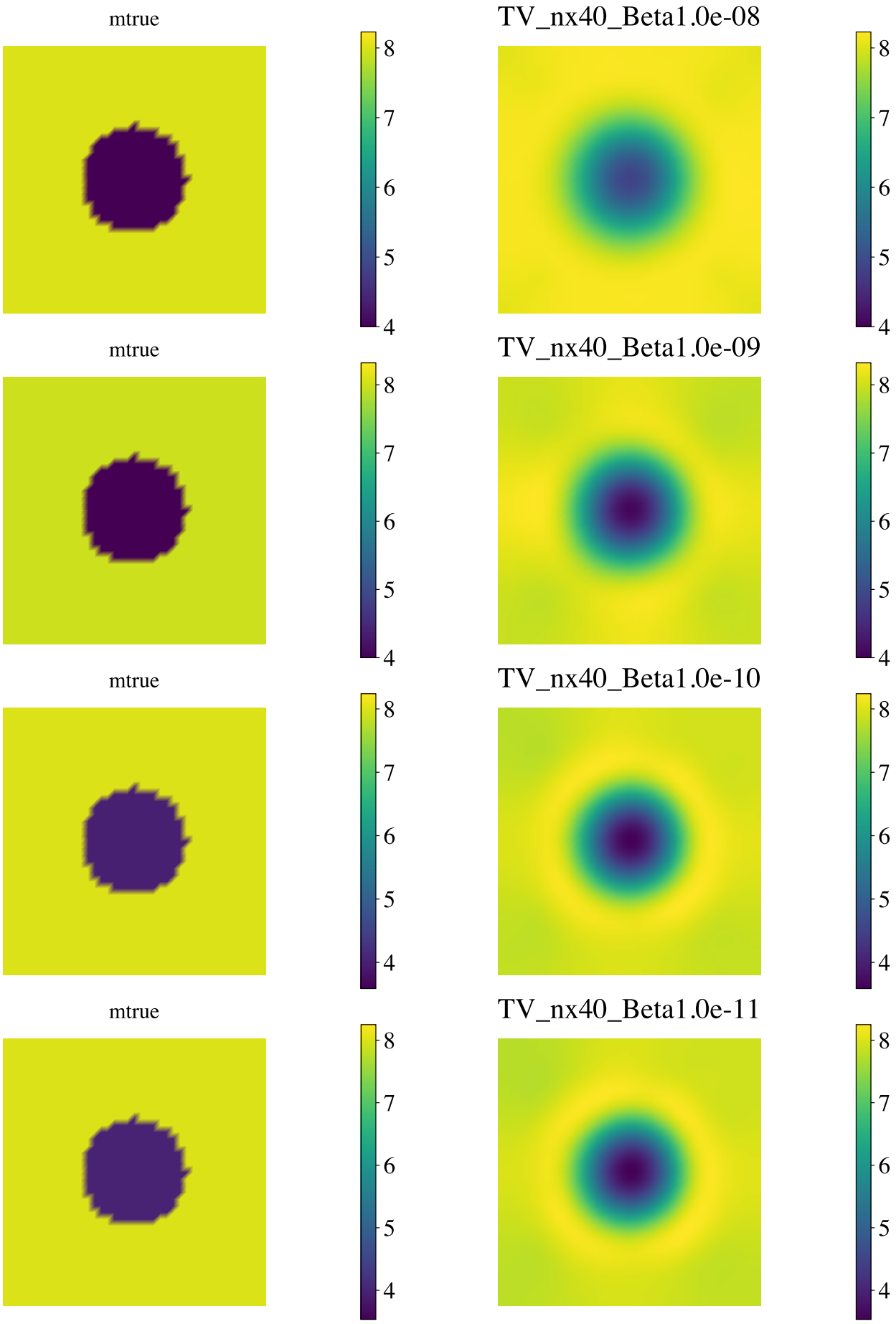


Figure 7: The inversion results for the viscosity field $m(\boldsymbol{x})$ for the cases $\beta = \{10^{-8}, 10^{-9}, 10^{-10}, 10^{-11}\}$ and $n = 40 \times 40$ after adding 1% noise. Total variation regularization has been used.

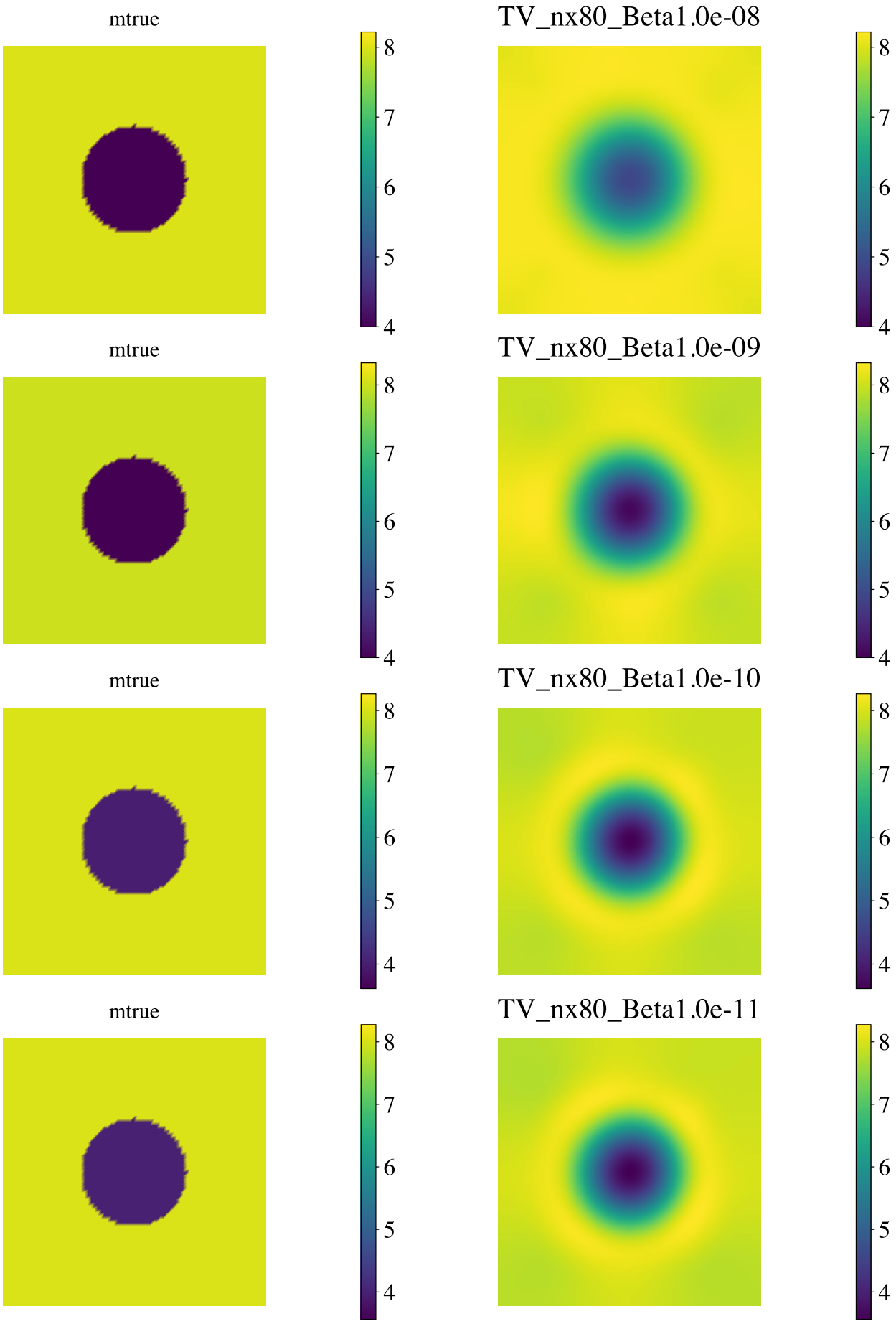


Figure 8: The inversion results for the viscosity field $m(\boldsymbol{x})$ for the cases $\beta = \{10^{-8}, 10^{-9}, 10^{-10}, 10^{-11}\}$ and $n = 80 \times 80$ after adding 1% noise. Total variation regularization has been used.

```
import dolfin as dl
import numpy as np

import sys
import os
sys.path.append( os.environ.get('HIPPYLIB_DIR', "../../") )

from hippylib import *

import logging

import matplotlib.pyplot as plt
%matplotlib inline
plt.rcParams.update({'font.size': 22})
plt.rcParams.update({'font.family': 'Times'})

logging.getLogger('FFC').setLevel(logging.WARNING)
logging.getLogger('UFL').setLevel(logging.WARNING)
dl.set_log_active(False)


nx = 64
ny = 64
mesh = dl.UnitSquareMesh(nx, ny)
Vm = dl.FunctionSpace(mesh, 'Lagrange', 1)
Vu = dl.FunctionSpace(mesh, 'Lagrange', 2)

# The true and initial guess inverted parameter
mtrue = dl.interpolate(dl.Expression('8. - 4.*(pow(x[0] - 0.5,2) + pow(x[1] - 0.5,2) <
                                          pow(0.2,2) )', degree=5), Vm)

# define function for state and adjoint
u = dl.Function(Vu)
m = dl.Function(Vm)
p = dl.Function(Vu)

# define Trial and Test Functions
u_trial, m_trial, p_trial = dl.TrialFunction(Vu), dl.TrialFunction(Vm), dl.TrialFunction
                                          (Vu)
u_test,  m_test,  p_test  = dl.TestFunction(Vu),  dl.TestFunction(Vm),  dl.TestFunction(
                                          Vu)

# initialize input functions
f = dl.interpolate(dl.Expression('std::max(0.5, exp(-25*(x[0]-0.7)*(x[0]-0.7) - 25 * (x[
                                          1]-0.7)*(x[1]-0.7)))', degree=2), Vu)
v  = dl.Constant((1.0, 0.0))#advective velocity vector
u0 = dl.Constant(0.0)       #BC
k  = dl.Constant(1.0)       #Diffusion parameter

# plot
plt.figure(figsize=(15,5))
nb.plot(mesh,subplot_loc=121, mytitle="Mesh", show_axis='on')
nb.plot(mtrue,subplot_loc=122, mytitle="True parameter field")
plt.show()

# set up dirichlet boundary conditions
def boundary(x,on_boundary):
    return on_boundary

bc_state = dl.DirichletBC(Vu, u0, boundary)
bc_adj = dl.DirichletBC(Vu, dl.Constant(0.), boundary)

# Class HessianOperator to perform Hessian apply to a vector
class HessianOperator():
    cgiter = 0
    def __init__(self, R, Wmm, C, A, adj_A, W, Wum, bc0, use_gaussnewton=False):
        self.R = R
        self.Wmm = Wmm
        self.C = C
        self.A = A
        self.adj_A = adj_A
        self.W = W
```

```
        self.Wum = Wum
        self.bc0 = bc0
        self.use_gaussnewton = use_gaussnewton

        # incremental state
        self.du = dl.Vector()
        self.A.init_vector(self.du,0)

        #incremental adjoint
        self.dp = dl.Vector()
        self.adj_A.init_vector(self.dp,0)

        # auxiliary vector
        self.Wum_du = dl.Vector()
        self.Wum.init_vector(self.Wum_du, 1)

    def init_vector(self, v, dim):
        self.R.init_vector(v,dim)

    # Hessian performed on v, output as generic vector y
    def mult(self, v, y):
        self.cgiter += 1
        y.zero()
        if self.use_gaussnewton:
            self.mult_GaussNewton(v,y)
        else:
            self.mult_Newton(v,y)

    # define (Gauss-Newton) Hessian apply H * v
    def mult_GaussNewton(self, v, y):

        #incremental forward
        rhs = -(self.C * v)
        self.bc0.apply(rhs)
        dl.solve (self.A, self.du, rhs)

        #incremental adjoint
        rhs = - (self.W * self.du)
        self.bc0.apply(rhs)
        dl.solve (self.adj_A, self.dp, rhs)

        # Misfit term
        self.C.transpmult(self.dp, y)

        if self.R:
            Rv = self.R*v
            y.axpy(1, Rv)

    # define (Newton) Hessian apply H * v
    def mult_Newton(self, v, y):

        #incremental forward
        rhs = -(self.C * v)
        self.bc0.apply(rhs)
        dl.solve (self.A, self.du, rhs)

        #incremental adjoint
        rhs = -(self.W * self.du) -  self.Wum * v
        self.bc0.apply(rhs)
        dl.solve (self.adj_A, self.dp, rhs)

        #Misfit term
        self.C.transpmult(self.dp, y)

        self.Wum.transpmult(self.du, self.Wum_du)
        y.axpy(1., self.Wum_du)

        y.axpy(1., self.Wmm*v)

        #Reg/Prior term
        if self.R:
            y.axpy(1., self.R*v)
```

```
def AddDiffInverseProblem(nx, ny, gamma, v, morozov = False, plot = True, noise_level=0.
                                                    01, useTV=False):
    np.random.seed(seed=1)
    mesh = dl.UnitSquareMesh(nx, ny)
    Vm = dl.FunctionSpace(mesh, 'Lagrange', 1)
    Vu = dl.FunctionSpace(mesh, 'Lagrange', 2)

    # The true and initial guess inverted parameter
    mtrue = dl.interpolate(dl.Expression('8. - 4.*(pow(x[0] - 0.5,2) + pow(x[1] - 0.5,2)
                                                     < pow(0.2,2) )', degree=5), Vm)

    # define function for state and adjoint
    u = dl.Function(Vu)
    m = dl.Function(Vm)
    p = dl.Function(Vu)

    # define Trial and Test Functions
    u_trial, m_trial, p_trial = dl.TrialFunction(Vu), dl.TrialFunction(Vm), dl.
                                                    TrialFunction(Vu)
    u_test,  m_test,  p_test  = dl.TestFunction(Vu),  dl.TestFunction(Vm),  dl.
                                                    TestFunction(Vu)

    # initialize input functions
    f = dl.interpolate(dl.Expression('std::max(0.5, exp(-25*(x[0]-0.7)*(x[0]-0.7) - 25 *
                                                     (x[1]-0.7)*(x[1]-0.7)))', degree=2), Vu
                                                    )
    v = dl.Constant((1.0, 0.0))
    k = dl.Constant(1.0)
    u0 = dl.Constant(0.0)

    # plot
    plt.figure(figsize=(15,5))
    nb.plot(mesh,subplot_loc=121, mytitle="Mesh", show_axis='on')
    nb.plot(mtrue,subplot_loc=122, mytitle="True parameter field")
    plt.show()

    # set up dirichlet boundary conditions
    def boundary(x,on_boundary):
        return on_boundary

    bc_state = dl.DirichletBC(Vu, u0, boundary)
    bc_adj = dl.DirichletBC(Vu, dl.Constant(0.), boundary)


    a_true = dl.inner( dl.exp(mtrue) * dl.grad(u_trial), dl.grad(u_test)) * dl.dx \
           + dl.dot(v, dl.grad(u_trial))*u_test*dl.dx
    L_true = f * u_test * dl.dx
    A_true, b_true = dl.assemble_system(a_true, L_true, bc_state)

    utrue = dl.Function(Vu)

    F = k*dl.inner(dl.grad(utrue), dl.grad(u_test)) * dl.dx + \
    dl.inner(v, dl.grad(utrue)) * u_test * dl.dx + \
    100*dl.exp(mtrue)*utrue*utrue*utrue*u_test * dl.dx - \
    f*u_test*dl.dx

    dl.solve(F == 0, utrue, bc_state, solver_parameters={"newton_solver": \
                                                    {"relative_tolerance": 1e-6}})

    ud = dl.Function(Vu)
    ud.assign(utrue)

    # perturb state solution and create synthetic measurements ud
    # ud = u + ||u||/SNR * random.normal
    MAX = ud.vector().norm("linf")
    noise = dl.Vector()
    A_true.init_vector(noise,1)
    noise.set_local( noise_level * MAX * np.random.normal(0, 1, len(ud.vector().
                                                    get_local())) )
    bc_adj.apply(noise)

    ud.vector().axpy(1., noise)
```

```
# define cost function
def cost(u, ud, m,gamma):
    if useTV:
        reg = gamma * dl.assemble( dl.sqrt(dl.inner(dl.grad(m), dl.grad(m)) + TVeps)
                                                    *dl.dx )
    else:
        reg = 0.5* gamma * dl.assemble( dl.inner(dl.grad(m), dl.grad(m))*dl.dx )

    misfit = 0.5 * dl.assemble( (u-ud)**2*dl.dx)
    return [reg + misfit, misfit, reg]

# weak form for setting up the state equation

F_state = dl.inner(k * dl.grad(u) , dl.grad(p_test)) * dl.dx + \
      dl.dot(v, dl.grad(u)) * p_test * dl.dx + \
      dl.Constant(100)*dl.exp(m) * u**3 * p_test * dl.dx - \
      f * p_test * dl.dx

# weak form for gradient
CTvarf = dl.inner(dl.Constant(100) * m_trial* dl.exp(m) * u**3, p_test) * dl.dx
gradRvarf = dl.Constant(gamma)*dl.inner(dl.grad(m), dl.grad(m_test))*dl.dx

# L^2 weighted inner product
M_varf    = dl.inner(m_trial, m_test) * dl.dx
M = dl.assemble(M_varf)

# weak form for setting up the adjoint equation

F_adj = (u - ud) * u_test * dl.dx + k * dl.inner(dl.grad(u_test), dl.grad(p)) * dl.
                                                dx + \
dl.dot(v, dl.grad(u_test)) * p * dl.dx + dl.Constant(300) * dl.exp(m) * u * u *
                                                u_test * p * dl.dx


# weak form for gradient
TVeps = 0.001

CTvarf = dl.Constant(100) * m_test * dl.exp(m) * u**3 * p * dl.dx
if useTV:
    gradRvarf = ( dl.Constant(gamma)/dl.sqrt(dl.inner(dl.grad(m), dl.grad(m)) +
                                                TVeps) )* \
                dl.inner(dl.grad(m), dl.grad(m_test))*dl.dx
else:
    gradRvarf = dl.Constant(gamma)*dl.inner(dl.grad(m), dl.grad(m_test))*dl.dx

# L^2 weighted inner product
M_varf    = dl.inner(m_trial, m_test) * dl.dx
M = dl.assemble(M_varf)

m0 = dl.interpolate(dl.Constant(4.), Vm )
m.assign(m0)

# solve state equation
dl.solve(F_state == 0, u, bc_state, solver_parameters={"newton_solver": {"
                                                relative_tolerance": 1e-6}})
uk = dl.Function(Vu)

# evaluate cost
[cost_old, misfit_old, reg_old] = cost(u, ud, m, gamma)

#Hessian varfs
W_varf    = dl.inner(u_trial, u_test) * dl.dx
R_varf    = dl.Constant(gamma) * dl.inner(dl.grad(m_trial), dl.grad(m_test)) * dl.dx

#C_varf    = dl.inner(dl.Constant(100) * dl.exp(m) * m_trial * u**3, u_test) * dl.dx
C_varf = dl.inner(dl.Constant(100) * m_trial* dl.exp(m) * u**3, p_test) * dl.dx
Wum_varf = dl.inner(dl.Constant(300) * dl.exp(m) * m_trial * u * u * u_test, p) * dl
                                                .dx
Wmm_varf = dl.inner(dl.Constant(100) * dl.exp(m) * m_trial * m_test *  u**3,  p) *
                                                dl.dx

a_adj = dl.inner(k * dl.grad(u_test), dl.grad(p_trial)) * dl.dx + \
```

```
        dl.inner(dl.dot(v, dl.grad(u_test)), p_trial) * dl.dx + \
        dl.inner(dl.Constant(300) * dl.exp(m) * u * u * u_test, p_trial) * dl.dx


a_state = dl.inner(k * dl.grad(u_trial), dl.grad(p_test)) * dl.dx + \
          dl.inner(dl.dot(v, dl.grad(u_trial)), p_test) * dl.dx + \
          dl.inner(dl.Constant(300) * dl.exp(m) * u * u * u_trial, p_test) * dl.dx

# Assemble constant matrices
W = dl.assemble(W_varf)
R = dl.assemble(R_varf)

# define parameters for the optimization
tol = 5e-7
c = 1e-4
maxiter = 20

# initialize iter counters
iter = 1
total_cg_iter = 0
converged = False

# initializations
g, m_delta = dl.Vector(), dl.Vector()
R.init_vector(m_delta,0)
R.init_vector(g,0)

m_prev = dl.Function(Vm)

print( "Nit   CGit   cost          misfit        reg           sqrt(-G*D)     ||grad|
                                          |       alpha  tolcg" )

while iter <  maxiter and not converged:

    # solve the adoint problem

    dl.solve(F_adj == 0, p, bc_adj, solver_parameters={"newton_solver": {"
                                            relative_tolerance": 1e-6}})

    #state_A, state_b = dl.assemble_system (a_state, L_state, bc_state)
    a_adj = dl.inner(k * dl.grad(u_test), dl.grad(p_trial)) * dl.dx + \
        dl.inner(dl.dot(v, dl.grad(u_test)), p_trial) * dl.dx + \
        dl.inner(dl.Constant(300) * dl.exp(m) * u * u * u_test, p_trial) * dl.dx


    adjoint_A , _= dl.assemble_system(a_adj, dl.Constant(0.) * p_test * dl.dx,
                                            bc_adj)

    a_state = dl.inner(k * dl.grad(u_trial), dl.grad(p_test)) * dl.dx + \
          dl.inner(dl.dot(v, dl.grad(u_trial)), p_test) * dl.dx + \
          dl.inner(dl.Constant(300) * dl.exp(m) * u * u * u_trial, p_test) * dl.dx


    state_A , _= dl.assemble_system(a_state, dl.Constant(0.) * u_test * dl.dx,
                                            bc_state)

    # evaluate the  gradient
    MG = dl.assemble(CTvarf + gradRvarf)

    # calculate the L^2 norm of the gradient
    dl.solve(M, g, MG)
    grad2 = g.inner(MG)
    gradnorm = np.sqrt(grad2)

    # set the CG tolerance (use Eisenstat[U+FFFD]Walker termination criterion)
    if iter == 1:
        gradnorm_ini = gradnorm
    tolcg = min(0.5, np.sqrt(gradnorm/gradnorm_ini))

    # assemble W_um and W_mm
    C   = dl.assemble(C_varf)
    Wum = dl.assemble(Wum_varf)
    Wmm = dl.assemble(Wmm_varf)
```

```
        # define the Hessian apply operator (with preconditioner)
        Hess_Apply = HessianOperator(R, Wmm, C, state_A, adjoint_A, W, Wum, bc_adj,
                                                    use_gaussnewton=(iter<6) )
        P = R + 0.1*gamma * M
        Psolver = dl.PETScKrylovSolver("cg", amg_method())
        Psolver.set_operator(P)

        solver = CGSolverSteihaug()
        solver.set_operator(Hess_Apply)
        solver.set_preconditioner(Psolver)
        solver.parameters["rel_tolerance"] = tolcg
        solver.parameters["zero_initial_guess"] = True
        solver.parameters["print_level"] = -1

        # solve the Newton system H a_delta = - MG
        solver.solve(m_delta, -MG)
        total_cg_iter += Hess_Apply.cgiter

        # linesearch
        alpha = 1
        descent = 0
        no_backtrack = 0
        m_prev.assign(m)
        while descent == 0 and no_backtrack < 10:
            m.vector().axpy(alpha, m_delta )

            # solve the state/forward problem
            dl.solve(F_state == 0, u, bc_state, solver_parameters={"newton_solver": {"
                                                        relative_tolerance": 1e-5}})
            a_state = dl.inner(k * dl.grad(u_trial), dl.grad(p_test)) * dl.dx + \
              dl.inner(dl.dot(v, dl.grad(u_trial)), p_test) * dl.dx + \
              dl.inner(dl.Constant(300) * dl.exp(m) * u * u * u_trial, p_test) * dl.dx

            state_A = dl.assemble(a_state)

            # evaluate cost
            [cost_new, misfit_new, reg_new] = cost(u, ud, m, gamma)

            # check if Armijo conditions are satisfied
            if cost_new < cost_old + alpha * c * MG.inner(m_delta):
                cost_old = cost_new
                descent = 1
            else:
                no_backtrack += 1
                alpha *= 0.5
                m.assign(m_prev)  # reset a

        # calculate sqrt(-G * D)
        graddir = np.sqrt(- MG.inner(m_delta) )

        sp = ""
        print( "%2d %2s %2d %3s %8.5e %1s %8.5e %1s %8.5e %1s %8.5e %1s %8.5e %1s %5.2f
                                                    %1s %5.3e" % \
            (iter, sp, Hess_Apply.cgiter, sp, cost_new, sp, misfit_new, sp, reg_new, sp,
                                                                \
             graddir, sp, gradnorm, sp, alpha, sp, tolcg) )

        # check for convergence
        if gradnorm < tol and iter > 1:
            converged = True
            print( "Newton's method converged in ",iter,"  iterations" )
            print( "Total number of CG iterations: ", total_cg_iter )

        iter += 1

    if not converged:
        print( "Newton's method did not converge in ", maxiter, " iterations" )

    if plot:
        nb.multi1_plot([mtrue, m], ["mtrue", "m"])
        if useTV:
            plt.title("TV_nx%.f_Beta%.1e" %(nx,gamma))
```

```
            plt.savefig("HW5_Q2_TV_%.f_%.1e_.pdf" %(nx,gamma))
        else:
            plt.title("TN_nx%.f_Beta%.1e" %(nx,gamma))
            plt.savefig("HW5_Q2_TN_%.f_%.1e_.pdf" %(nx,gamma))
        plt.show()

    Mstate = dl.assemble(u_trial*u_test*dl.dx)
    noise_norm2 = noise.inner(Mstate*noise)

    if not morozov:
        Hmisfit = HessianOperator(None, Wmm, C, state_A, adjoint_A, W, Wum, bc_adj,
                                                    use_gaussnewton=True)
        k = 50
        p = 20

        Omega = MultiVector(m.vector(), k+p)
        parRandom.normal(1., Omega)
        lmbda, evecs = doublePassG(Hmisfit, P, Psolver, Omega, k)

        plt.plot(range(0,k), lmbda, 'b*', range(0,k+1), np.ones(k+1), '-r')
        plt.yscale('log')
        plt.xlabel('number')
        plt.ylabel('eigenvalue')
        plt.show()

        nb.plot_eigenvectors(Vm, evecs, mytitle="Eigenvector", which=[0,1,2,5,10,15])
        plt.show()



    return Vm.dim(), iter, total_cg_iter, noise_norm2, cost_new, misfit_new, reg_new
# Part 2.2: FEniCS implementation of the inexact Newton and Gauss-Newton methods
n = 20
gammas = [1e-7, 1e-8, 1e-9, 1e-10, 1e-11]
#gammas = [1e-8]
misfits = []

for gamma in gammas:
    print('****** Computing solution with gamma = ', gamma, '******')
    ndof, niter, ncgiter, noise_norm2, cost,misfit,reg = AddDiffInverseProblem(n, n,
                                                gamma, v = dl.Constant((1.0, 0.0)),
                                                morozov =True, plot = True,noise_level=0
                                                .01, useTV=False)


n = 40
gammas = [1e-7, 1e-8, 1e-9, 1e-10]
#gammas = [1e-8]
misfits = []

for gamma in gammas:
    print('****** Computing solution with gamma = ', gamma, '******')

    ndof, niter, ncgiter, noise_norm2, cost,misfit,reg = AddDiffInverseProblem(n, n,
                                                gamma, v = dl.Constant((1.0, 0.0)),
                                                morozov = True, plot = True,noise_level=
                                                0.01,useTV=False)
n = 80
gammas = [1e-7, 1e-8, 1e-9, 1e-10]
#gammas = [1e-8]
misfits = []

for gamma in gammas:
    print('****** Computing solution with gamma = ', gamma, '******')

    ndof, niter, ncgiter, noise_norm2, cost,misfit,reg = AddDiffInverseProblem(n,n,
                                                gamma, v = dl.Constant((1.0, 0.0)),
                                                morozov = True, plot = True,noise_level=
                                                0.01, useTV=False)
# Part 2.3: Consider two noise levels: low noise (1%) and high noise (10%) and find the
                                            optimal parameter

n = 40
```

```
gammas = [1e-7, 1e-8, 1e-9, 1e-10]
#gammas = [1e-8]
misfits = []

for gamma in gammas:
    print('****** Computing solution with gamma = ', gamma, '******')
    ndof, niter, ncgiter, noise_norm2, cost,misfit,reg = AddDiffInverseProblem(n,n,
                                                gamma, v = dl.Constant((1.0, 0.0)),
                                                morozov = True, plot = True,noise_level=
                                                0.10, useTV=False)

n = 40
gammas = [1e-7, 1e-8, 1e-9, 1e-10]
#gammas = [1e-8]
misfits = []

for gamma in gammas:
    print('****** Computing solution with gamma = ', gamma, '******')
    ndof, niter, ncgiter, noise_norm2, cost,misfit,reg = AddDiffInverseProblem(n,n,
                                                gamma, v = dl.Constant((1.0, 0.0)),
                                                morozov = True, plot = True,noise_level=
                                                0.01,useTV=False)

# Part 2.4: Tikhonov regularization with total variation regularization $\delta=0.001$

n = 20
gammas = [1e-8,1e-9, 1e-10,1e-11]
misfits = []

for gamma in gammas:
    print('****** Computing solution with gamma = ', gamma, '******')
    ndof, niter, ncgiter, noise_norm2, cost,misfit,reg = AddDiffInverseProblem(n, n,
                                                gamma, v = dl.Constant((1.0, 0.0)),
                                                morozov = True, plot = True,noise_level=
                                                0.01,useTV=True)

n = 40
gammas = [1e-8,1e-9, 1e-10,1e-11]
#gammas = [1e-8]
misfits = []

for gamma in gammas:
    print('****** Computing solution with gamma = ', gamma, '******')
    ndof, niter, ncgiter, noise_norm2, cost,misfit,reg = AddDiffInverseProblem(n,n,
                                                gamma, v = dl.Constant((1.0, 0.0)),
                                                morozov = True, plot = True,noise_level=
                                                0.01,useTV=True)

n = 80
gammas = [1e-8,1e-9, 1e-10,1e-11]
#gammas = [1e-8]
misfits = []

for gamma in gammas:
    print('****** Computing solution with gamma = ', gamma, '******')

    ndof, niter, ncgiter, noise_norm2, cost,misfit,reg = AddDiffInverseProblem(n,n,
                                                gamma, v = dl.Constant((1.0, 0.0)),
                                                morozov = True, plot = True,noise_level=
                                                0.01, useTV=True)
```