Spring 2024: Compututational and Variational Methods for Inverse Problems

Assignment-04: Assignment 4: Inverse Problems Governed by PDE Forward Models

Shreshth Saini (SS223464)
saini.2@utexas.edu
Due April 29, 2024

Problem-1: Frequency-domain inverse wave propagation

Given:

The inverse problem with Tikhonov regularization is given as:

$$\min_n \Phi(m) := \frac{1}{2}\sum_i^{N_f}\sum_j^{N_s}\int_\tau (u_{ij}(m) - d_{ij})^2 ds + \frac{\beta}{2}\int_\Omega \nabla m.\nabla m dx \tag{1.1}$$

where N_f, N_s are the number of frequencies and sources, respectively. d_{ij} denotes given measurements for frequency i and source j. u_{ij} is acoustic wavefield. $\beta > 0$ is the regularization parameter.

Furthermore, u_{ij} depends on medium parameter field $m(x) = 1 - c_0^2/c(x)^2$ through the solution of the Helmholtz equation:

$$-\Delta u_{ij} - k_{0,i}^2(1-m)u_{ij} = k_{0,i}^2 m u_{ij}^{inc} \text{ in } \Omega, i = 1, 2, \ldots, N_f, j = 1, 2, \ldots, N_s$$

$$\frac{\partial u_{ij}}{\partial n} = 0, \text{ on } \tau \tag{1.2}$$

where $k_{0,i} = \frac{w_i}{c_0}$ is the wavenumber.

(a) Infinite Dimensional Gradient of Φ:

Let $u \in H^1(\Omega)$ and a corresponding variation $\hat{u} \in H^1\Omega$.

Give the single source and frequency case, the inverse problem can be written as:

$$\min_n \Phi(m) := \frac{1}{2}\int_\tau (u(m) - d)^2 ds + \frac{\beta}{2}\int_\Omega \nabla m.\nabla m dx \tag{1.3}$$

where d denotes given measurements. u depends on medium parameter field $m(x) = 1 - c_0^2/c(x)^2$ through the solution of the Helmholtz equation:

$$-\Delta u - k_0^2(1-m)u = k_0^2 m u^{inc} \text{ on } \Omega \tag{1.4}$$

$$\frac{\partial u}{\partial n} = 0, \text{ on } \tau \tag{1.5}$$

The weak form of the PDE can be obtained by multiplying with test function $p \in H_0^1(\Omega)$:

$$\int_\Omega (-\Delta u - k_0^2(1-m)u)p dx = k_0^2\int_\Omega m u^{inc} p dx \tag{1.6}$$

Integrating by parts, we get:

$$\int_\Omega \nabla u.\nabla p dx - k_0^2\int_\Omega (1-m)up dx - \int_\tau p\nabla u.n ds = k_0^2\int_\Omega m u^{inc} p dx \tag{1.7}$$

Here we can use the boundary condition to simplify the equation. $p = 0$ on τ, Weak form becomes:

Find $u \in H_0^1(\Omega)$ such that:

$$\int_\Omega \nabla u.\nabla p - k_0^2((1-m)u + mu^{inc})pdx = 0, \forall p \in H_0^1(\Omega) \tag{1.8}$$

Gradient of Φ:

Gradient of Φ can be obtained by defining a Langrangian functional using inner problem's objective function and weak form of the PDE (sum of the two equations):

$$\mathcal{L}(m,u,p) = \frac{1}{2}\int_\tau (u-d)^2 ds + \frac{\beta}{2}\int_\Omega \nabla m.\nabla m dx + \int_\Omega \nabla u.\nabla p - k_0^2((1-m)u + mu^{inc})pdx \tag{1.9}$$

gradient wrt $o \in m, u, p$, using the calculus of variations the gradient o -> $o + \epsilon\hat{o}$ and performing $d/d\epsilon$ with $\epsilon = 0$, the Weak form of graident is:

- Forward problem:

$$Find u \in H_0^1(\Omega) \text{ such that } : \delta_p\mathcal{L}(p;\hat{p}) = \int_\Omega \nabla u.\nabla \hat{p} - k_0^2((1-m)u + mu^{inc})\hat{p}dx = 0, \forall \hat{p} \in H$$

For the strong form of the gradient, we perform intergration by parts in the first term of above equation, that gives us the storm form of gradient using test function $\hat{p}$

$$-\Delta u - k_0^2(1-m)u = k_0^2 m u^{inc} \text{ in } \Omega \tag{1.11}$$

$$\frac{\partial u}{\partial n} = 0, \text{ on } \tau \tag{1.12}$$

- Adjoint problem:

Find $p \in H_0^1(\Omega)$ such that:

$$\delta_u\mathcal{L}(u;\hat{u}) = \int_\Omega \hat{u}(u-d)ds + \int_\Omega \nabla\hat{u}.\nabla p - k_0^2((1-m)\hat{u})pdx = 0, \forall \hat{u} \in H_0^1(\Omega) \tag{1.13}$$

Integrating the weak form by parts gives:

$$\int_\Omega (u-d)\hat{u}dx + \int_\Omega -\Delta p - k_0^2(1-m)p\hat{u}dx = 0 \tag{1.14}$$

Herem the boundary term vanishes due to the boundary condition on $\hat{u} = 0$ on τ. Using the arbitrariness of $\hat{u}$, we get the strong form:

$$-\Delta p - k_0^2(1-m)p = u - d \text{ in } \Omega \tag{1.15}$$

$$\frac{\partial p}{\partial n} = 0, \text{ on } \tau \tag{1.16}$$

Finally, the gradient of Φ wrt m is given by:

$$\delta_m\mathcal{L}(m;\hat{m}) = \beta\int_\Omega \nabla m.\nabla\hat{m} + k_0^2(u-u^{inc})\hat{m}pdx, \forall \hat{m} \in H_0^1(\Omega) \tag{1.17}$$

For the strong form of the gradient, we perform intergration by parts in the first term of above equation, that gives us the storm form of gradient using test function $\hat{m}$:

$$\delta_m\mathcal{L}(m;\hat{m}) = \int_\Omega -\beta\hat{m}\Delta m + k_0^2(u-u^{inc})\hat{m}pdx + \beta\int_\tau \hat{m}.\nabla m.nds \tag{1.18}$$

Thus the strong form of the gradient is:

$$-\beta\Delta m + k_0^2(u-u^{inc})p = 0, \text{ in } \Omega \tag{1.19}$$

$$\beta\nabla m n = 0, \text{ on } \tau \tag{1.20}$$

(b) Infinite dimesional Hessian action :

Hessian action in direction of $\tilde{m}$; we define the new langrangian functional for forward and adjoint problem as:

$$\mathcal{L}^H(m,u,p,\tilde{m},\tilde{u},\tilde{p}) = \delta_m\mathcal{L}(\tilde{m}) + \delta_u\mathcal{L}(\tilde{u}) + \delta_p\mathcal{L}(\tilde{p}) \tag{1.21}$$

using the gradients from previous part of the question-1 (1.10, 1.13, 1.17), we get the Hessian action as:

$$\mathcal{L}^H(m,u,p,\tilde{m},\tilde{u},\tilde{p}) = \beta\int_\Omega \nabla m\nabla\tilde{m} + k_0^2(u-u^{inc})\tilde{m}pdx + \int_\tau \nabla u\nabla\tilde{p} - k_0^2(1-m)u\tilde{p} - k_0^2mu^{inc}\tilde{p}d$$

Using the calculus of variations the gradient is m -> $m+\epsilon\tilde{m}$ and performing $d/d\epsilon$ with $\epsilon=0$, we get the weak form of the Hessian action is:

$$\mathcal{H}(\tilde{m})\hat{m} = \delta_m\mathcal{L}^H = \beta\int_\Omega \nabla\tilde{m}.\nabla\hat{m} + k_0^2\hat{m}u\tilde{p} - k_0^2\hat{m}u^{inc}\tilde{p} - k_0^2(-\hat{m})\tilde{u}pdx, \forall\hat{m}\in H_0^1(\Omega) \tag{1.23}$$

For the strong form of the Hessian action, we perform intergration by parts in the first term of above equation, that gives us the storm form of Hessian action using test function $\hat{m}$:

$$\mathcal{H}(\tilde{m})\hat{m} = \int_\Omega -\beta\Delta\tilde{m}\hat{m} + k_0^2\hat{m}u\tilde{p} - k_0^2\hat{m}u^{inc}\tilde{p} - k_0^2(-\hat{m})\tilde{u}pdx + \beta\int_\tau \nabla\tilde{m}.\hat{m}.nds \tag{1.24}$$

Thus the strong form of the Hessian action is:

$$-\beta\Delta\tilde{m} + k_0^2u\tilde{p} - k_0^2u^{inc}\tilde{p} + k_0^2\tilde{u}p = 0, \text{ in } \Omega \tag{1.25}$$

$$\beta\nabla\tilde{m}.n, \text{ on } \tau \tag{1.26}$$

(c) Infinite Dimensional graident (General):

Gradient for arbitrary number of sources and frequencies, we use the langrangian functional from original form 1.1 and 1.2 in weak form:

$$\mathcal{L}(m,u_{ij},p_{ij}) = \frac{1}{2}\sum_i^{N_f}\sum_j^{N_s}\int_\tau (u_{ij}-d_{ij})^2ds + \frac{\beta}{2}\int_\Omega \nabla m.\nabla mdx + \sum_i^{N_f}\sum_j^{N_s}\int_\Omega \nabla u_{ij}.\nabla p_{ij} - k_{0,i}^2((1-m$$

- We can see from to equation that there are total N_fXN_s constraints on PDE (all having same form).

Getting the weak form :

- Forward problem:

Find $u_{ij}\in H_0^1(\Omega)$ such that:

$$\delta_p\mathcal{L}(p_{ij};\hat{p}_{ij}) = \int_\Omega \nabla u_{ij}.\nabla\hat{p}_{ij} - k_{0,i}^2((1-m)u_{ij} + mu_{ij}^{inc})\hat{p}_{ij}dx = 0, \forall\hat{p}_{ij}\in H_0^1(\Omega) \tag{1.28}$$

following same structure as in 1.1 and 1.2, the stron form is:

$$-\Delta u_{ij} - k_{0,i}^2(1-m)u_{ij} = k_{0,i}^2mu_{ij}^{inc} \text{ in } \Omega \tag{1.29}$$

$$\frac{\partial u_{ij}}{\partial n} = 0, \text{ on } \tau \tag{1.30}$$

- Adjoint problem:

Find $p_{ij}\in H_0^1(\Omega)$ such that:

$$\delta_u\mathcal{L}(u_{ij};\hat{u}_{ij}) = \int_\Omega \hat{u}_{ij}(u_{ij}-d_{ij})ds + \int_\Omega \nabla\hat{u}_{ij}.\nabla p_{ij} - k_{0,i}^2((1-m)\hat{u}_{ij})p_{ij}dx = 0, \forall\hat{u}_{ij}\in H_0^1(\Omega) \tag{1.3}$$

The strong form is:

$$-\Delta p_{ij} - k_{0,i}^2(1-m)p_{ij} = u_{ij} - d_{ij} \text{ in } \Omega \tag{1.32}$$

$$\frac{\partial p_{ij}}{\partial n} = 0, \text{ on } \tau \tag{1.33}$$

- Gradient of Φ wrt m is given by (weak form):

$$\delta_m \mathcal{L}(m;\hat{m}) = \beta \int_\Omega \nabla m . \nabla \hat{m} - \int_\Omega \sum\sum k_{0,i}^2 (1-\hat{m}) u_{ij} p_{ij} dx, \forall \hat{m} \in H_0^1(\Omega) \tag{1.34}$$

The strong form is:

$$-\beta \Delta m - \sum\sum k_{0,i}^2 (1-m) u_{ij} p_{ij}, \text{ in } \Omega \tag{1.35}$$

$$\beta \nabla m . n, \text{ on } \tau \tag{1.36}$$

For each gradient $N_f X N_s$ forwards and same number of adjoint equations needs to be solved.

Problem-2: The steepest descent method for solving the Helmholtz inverse problem:

Given:

We are using single source and frequency case, and the medium properties is such that $\tanh(m(x)) = 1 - \frac{c_0^2}{c(x)^2}$. The final inverse problem is given as:

$$\min_m \Phi(m) := \frac{1}{2} \int_\tau (u(m) - d)^2 ds + \frac{\beta}{2} \int_\Omega \nabla m . \nabla m dx \tag{2.1}$$

u depends on medium parameter field $m(x)$ through the solution of the Helmholtz equation:

$$-\Delta u - k_0^2 (1 - \tanh(m)) u = k_0^2 \tanh(m) u^{inc} \text{ in } \Omega \tag{2.2}$$

$$\frac{\partial u}{\partial n} = 0, \text{ on } \tau \tag{2.3}$$

Note that the domain $\Omega = \backslash \mathrm{R}^2 ||| x ||^2 <= 1$ is a unit circle, $\Omega = [0,1] \times [0,1]$. $k_0 = 5$ is wave number. u_{inc} is the incident wavefield, which is given as the superposition of three spherical waves with wave number generated by point like sources:

$$u^{inc}(x) = \sum_{i=1}^{3} -e_i \frac{cos(k_0 |x - x_i|)}{4\pi |x - x_i|} \text{ for } x \in \Omega \tag{2.4}$$

where x_i denote the location of three point-like sources and e_i $N(0,1)$ are randomly chosen coefficients.

measurement data d is generated by solving the Helmholtz equation with the true medium properties m_{true}, further corrupted by Gaussian noise with standard deviation $\sigma = 0.01$.:

$$m(x,y) = 0.2, \text{ if } \sqrt{(x-0.1)^2 + 2(y+0.2)^2} < 0.5, 0.1 \text{ otherwise} \tag{2.5}$$

NOTE: for reference, equations 2.2-2.5 are implemented in CVIPS/assignment folder on FEniCS server.

NOTE: for reference, steepest decent method implementation is also provided - 05_Poisson_SD/Poisson_SD.ipynb

- Langrangian functional based on given conditions:

$$\mathcal{L}(m,u,p) = \frac{1}{2} \int_\tau (u-d)^2 ds + \frac{\beta}{2} \int_\Omega \nabla m . \nabla m dx + \int_\Omega \nabla u . \nabla p - k_0^2 ((1 - \tanh(m)) u + \tanh(m) u^{in}$$

Note that the boundary conditions are zero due to homogeneous neumann boundary conditions.

1. Forward problem:

   Find $u \in H_0^1(\Omega)$ such that:

$$\delta_p \mathcal{L} = \int_\Omega \nabla u . \nabla \hat{p} - k_0^2 ((1 - \tanh(m)) u + \tanh(m) u^{inc}) \hat{p} dx = 0, \forall p \in H_0^1(\Omega) \tag{2.7}$$

2. Adjoint problem:

   Find $p \in H_0^1(\Omega)$ such that:

$$\delta_u \mathcal{L} = \int_\Omega \hat{u}(u-d)ds + \int_\Omega \nabla p.\nabla\hat{u} - k_0^2((1-\tanh(m))p)\hat{u}dx = 0, \forall \hat{u} \in H_0^1(\Omega) \qquad (2.8)$$

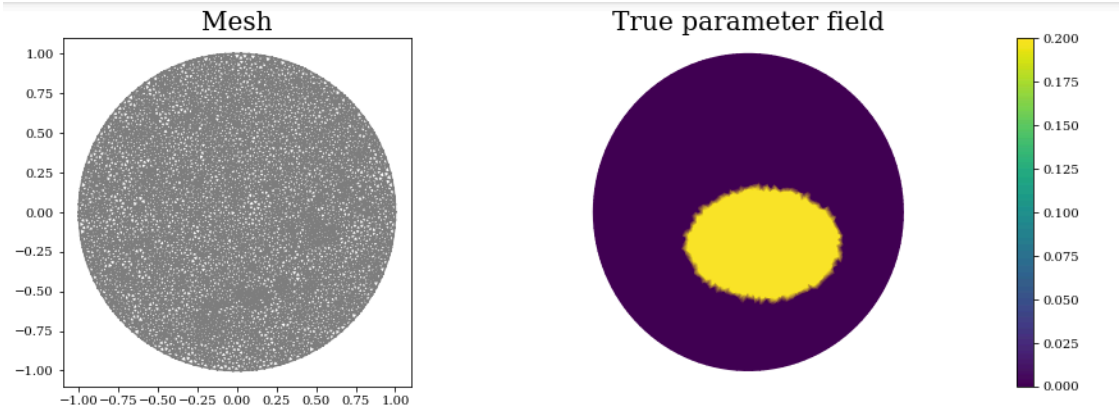
3. Gradient of Φ wrt m is given by (weak form):

$$\mathcal{G}(m,\hat{m}) = \delta_m \mathcal{L} = \beta \int_\Omega \nabla m.\nabla\hat{m} - \int_\Omega k_0^2(-(1-\tanh^2(m)) * u + (1-tanh^2(m)) * u^{inc})\hat{m}pdx, \forall \hat{m}$$

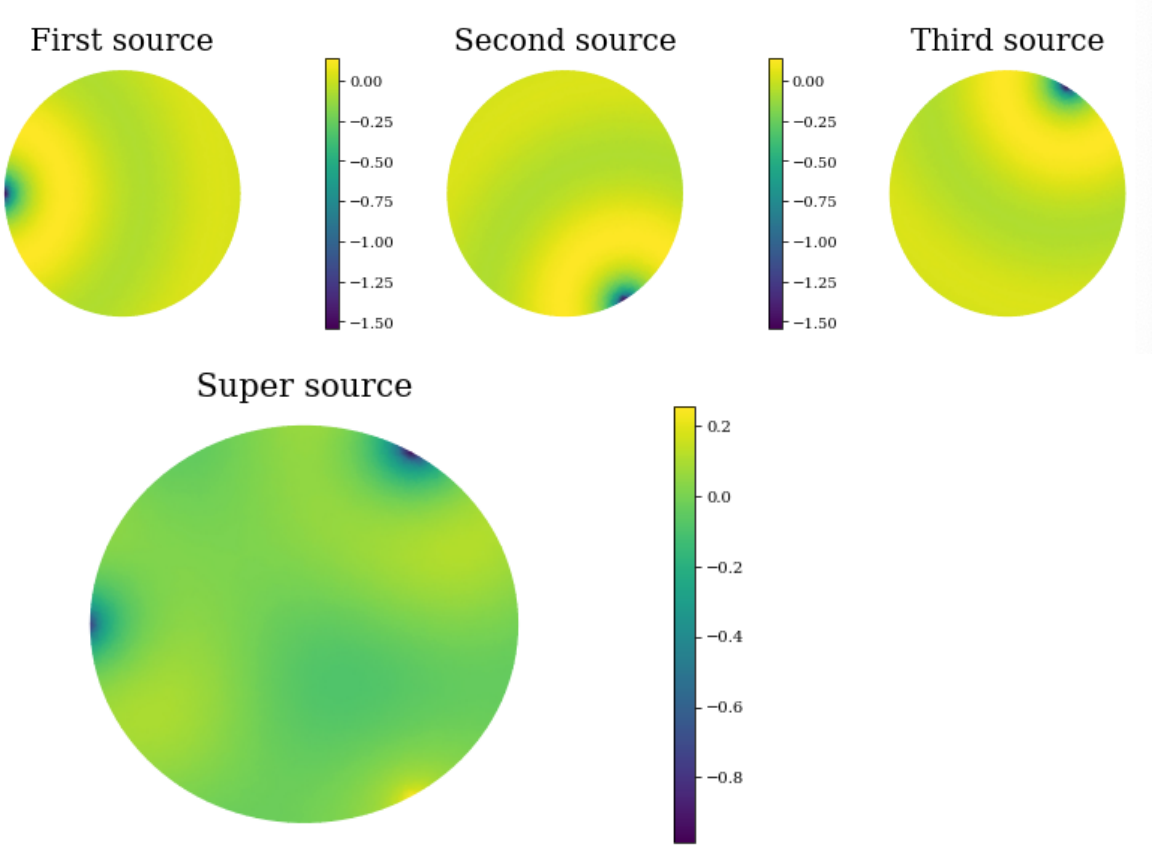
(1) Solving the Inverse Problem with H^1 Regularization

General Plots:

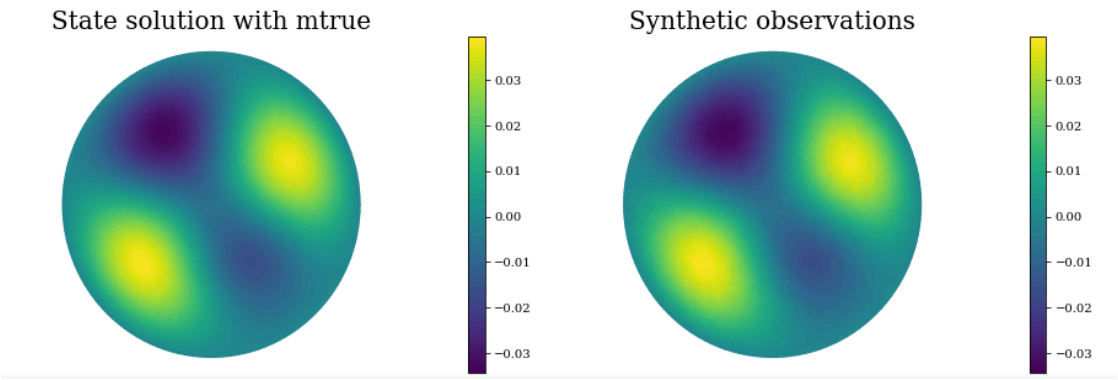
1. Mesh and True Parametere Field:



2. Discrete sources and Super Source:



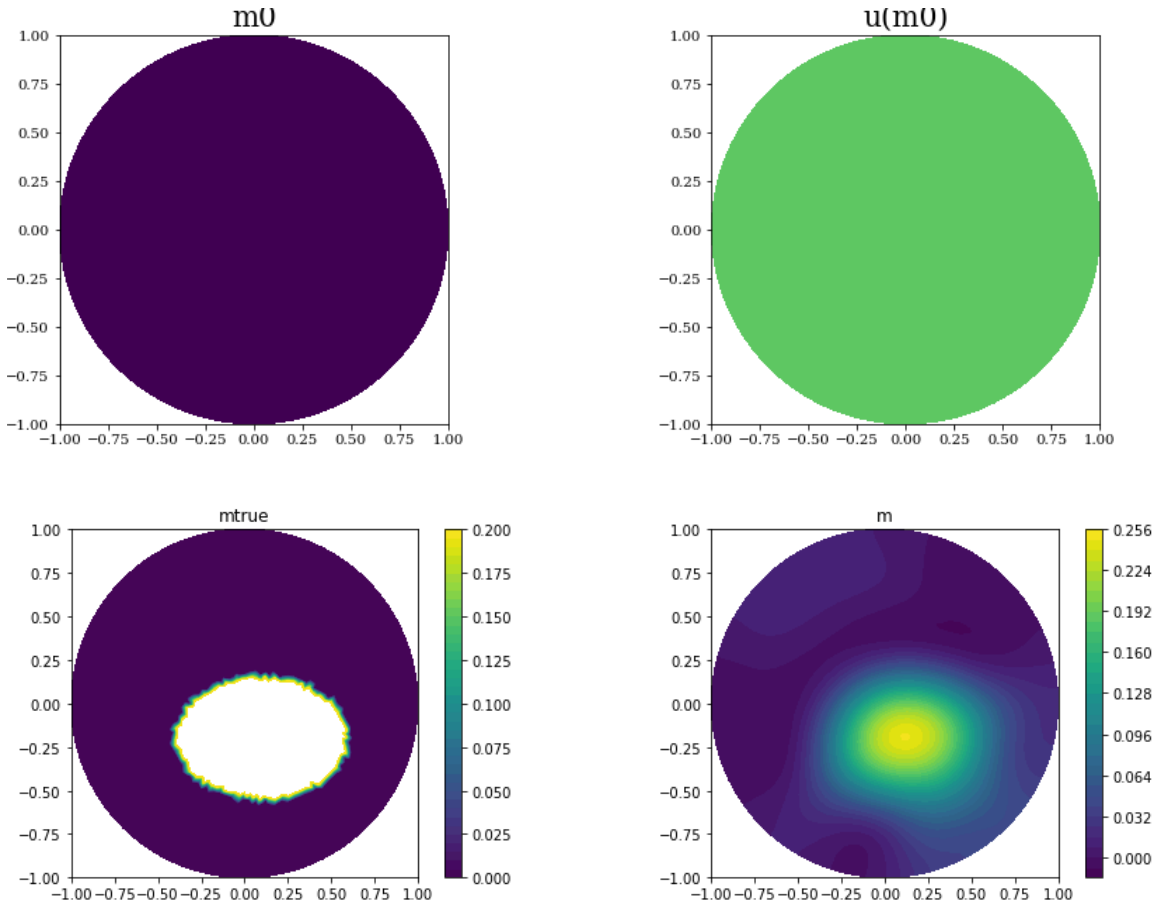
3. State Solution with m_true and Synthetic Observations with $\sigma = 0.01$:

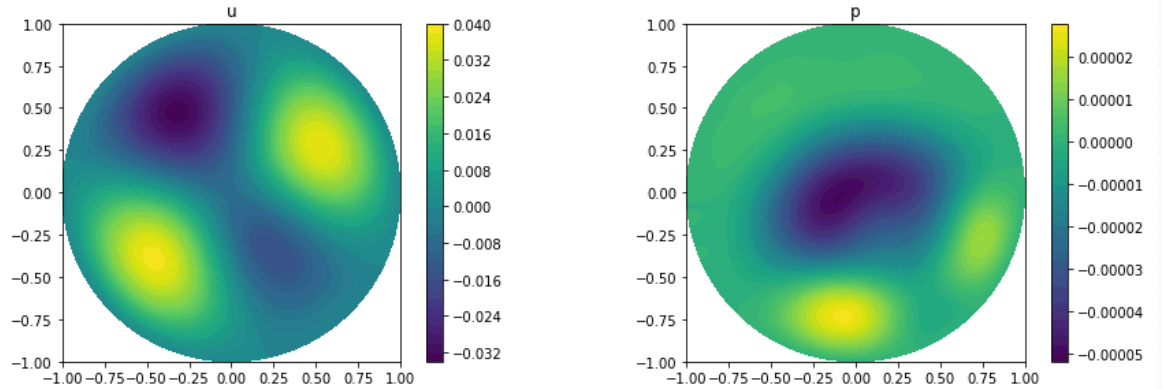


NOTE: Please see the appendix for full code.

(a) Implementing the steepest descent method with $\beta = 10^{-5}, m(x) = 0, \alpha = 1$:

- Converged at 1110 iteration.
- Figure for Inversion results with $\beta = 10^{-5}$, $m(x) = 0$, and $\alpha = 1$:



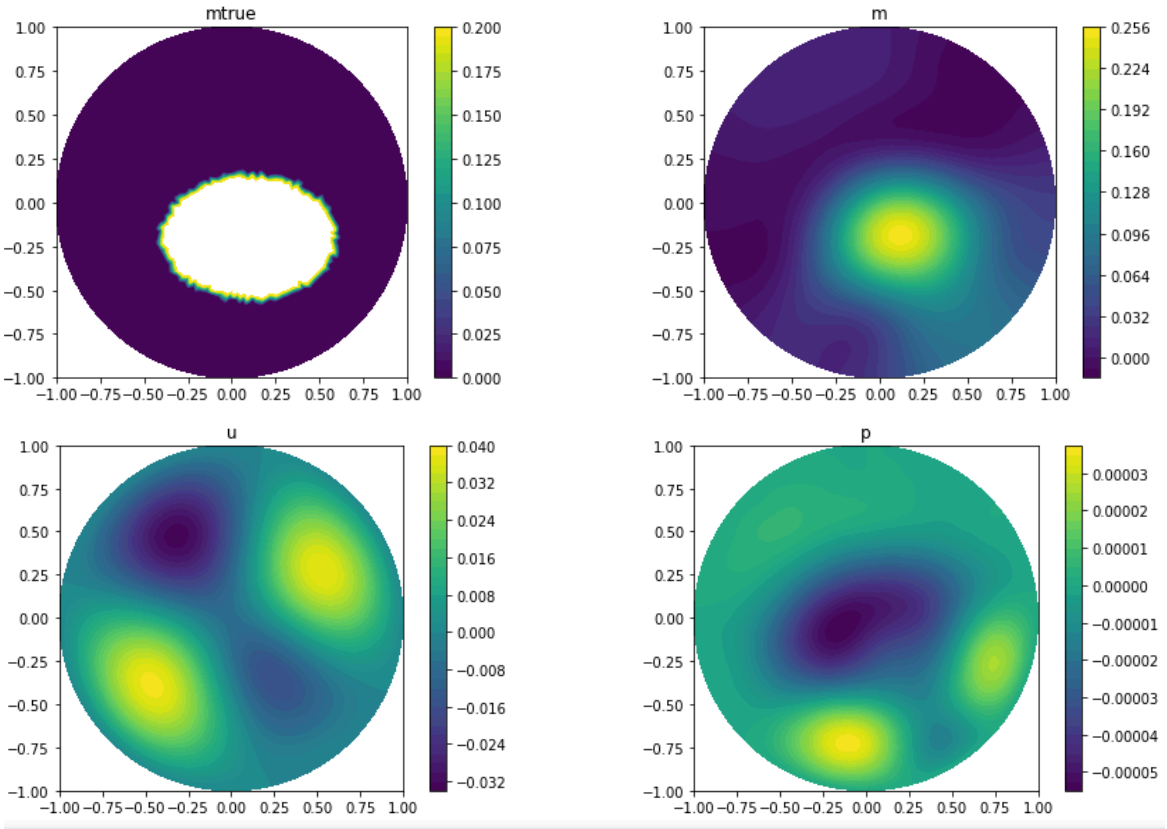


We keep the parameters as recommended in the question. The convergence is achieved at 1110 iterations. The inversion results are shown in the figures above. The parameters are recovered decently; The inverted m is diffused this is because pf the Tikhonov regularization H^1.

In [ ]:
```python
# Part 2.1.b

# main function

if __name__ == "__main__":
    # input parameters
    mesh_file = "circle.xml"
    beta = 1e-5
    noise_level = 0.01
    nsources = 3
    k0 = 5.0
    iters = 2000
    alpha = 1
    regul = "h1"
    start_point = 0.0 # "low" or float # meaning at m = 0; else takes 8; or give float value
    graph_name = "Inversion results with Precondition for case beta =1e-5, m(x)=0"
    precon = True

    # initialize the class
    helm = Helmholtz(start_point, mesh_file, beta, noise_level, nsources, k0, iters, alpha, reg
    # solve the problem
    helm.solution_flow()
```

(b) Implementing the steepest descent with H1-inner product to precondition with $\beta = 10^{-5}, m(x) = 0, \alpha = 1$:

- Steepest descent converged in 438 iterations

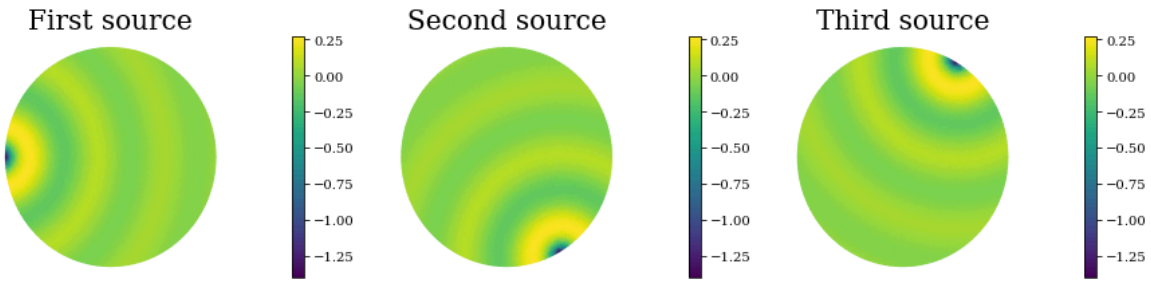


The convergence is achieved at 438 iterations. The inversion results are shown in the figures above. Using the preconditioning with H1-inner product, the convergence is faster than the previous case.

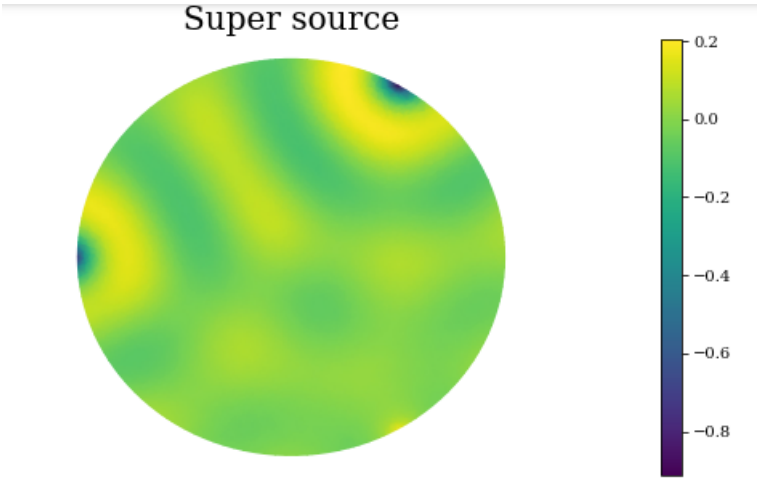
In [ ]:
```
# Part 2.1.c
# main function

if __name__ == "__main__":
    # input parameters
    mesh_file = "circle.xml"
    beta = 1e-5
    noise_level = 0.01
    nsources = 3
    k0 = 10.0
    iters = 2000
    alpha = 1
    regul = "h1"
    start_point = 0.0 # "low" or float # meaning at m = 0; else takes 8; or give float value
    graph_name = "Inversion results for case beta =1e-5, m(x)=0"
    precon = True

    # initialize the class
    helm = Helmholtz(start_point, mesh_file, beta, noise_level, nsources, k0, iters, alpha, reg
    # solve the problem
    helm.solution_flow()
```

(C) Implementing the steepest descent with H1-inner product to precondition with $\beta = 10^{-5}, m(x) = 0, \alpha = 1$, and $k_0 = 10$:

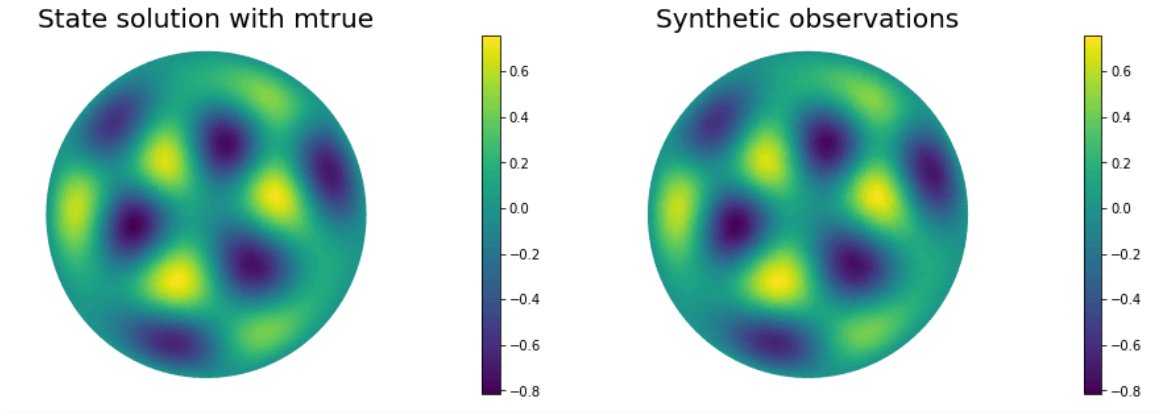
- Sources



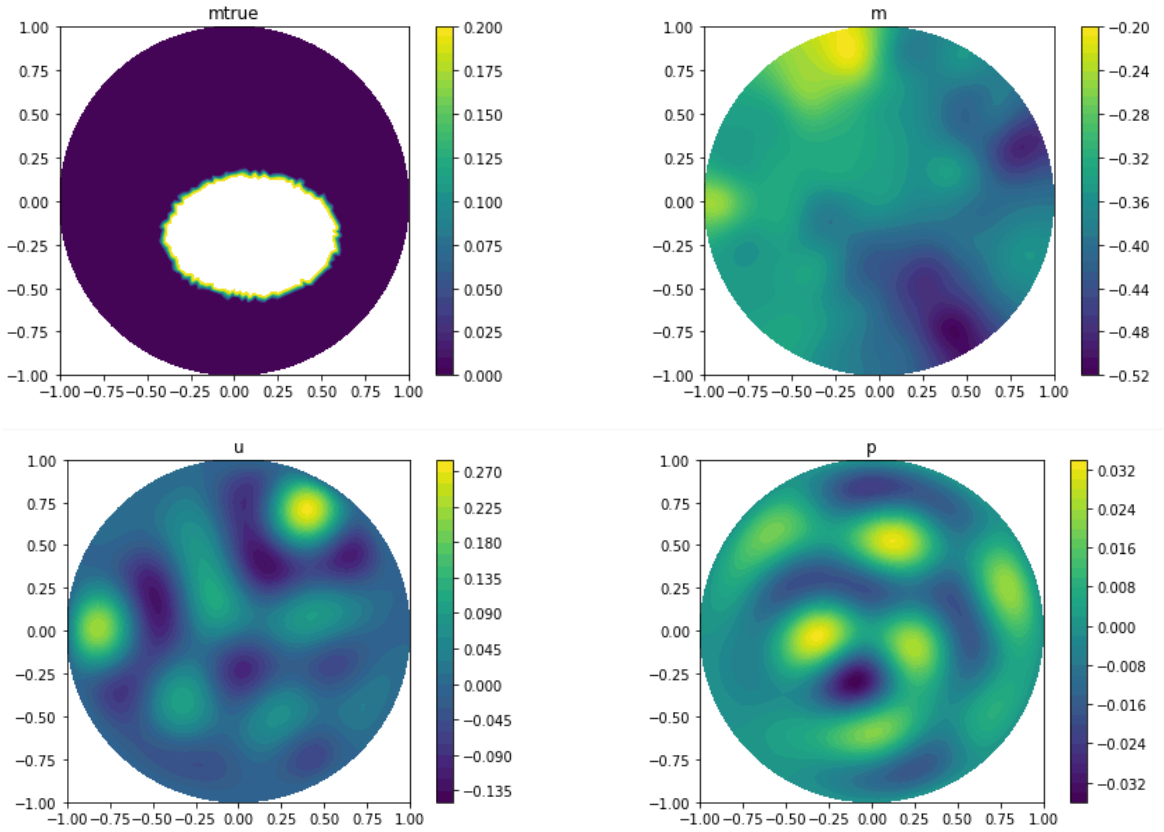
- Super Source



- State Solution with m_true and Synthetic Observations with $\sigma = 0.01$:



- Inversion results

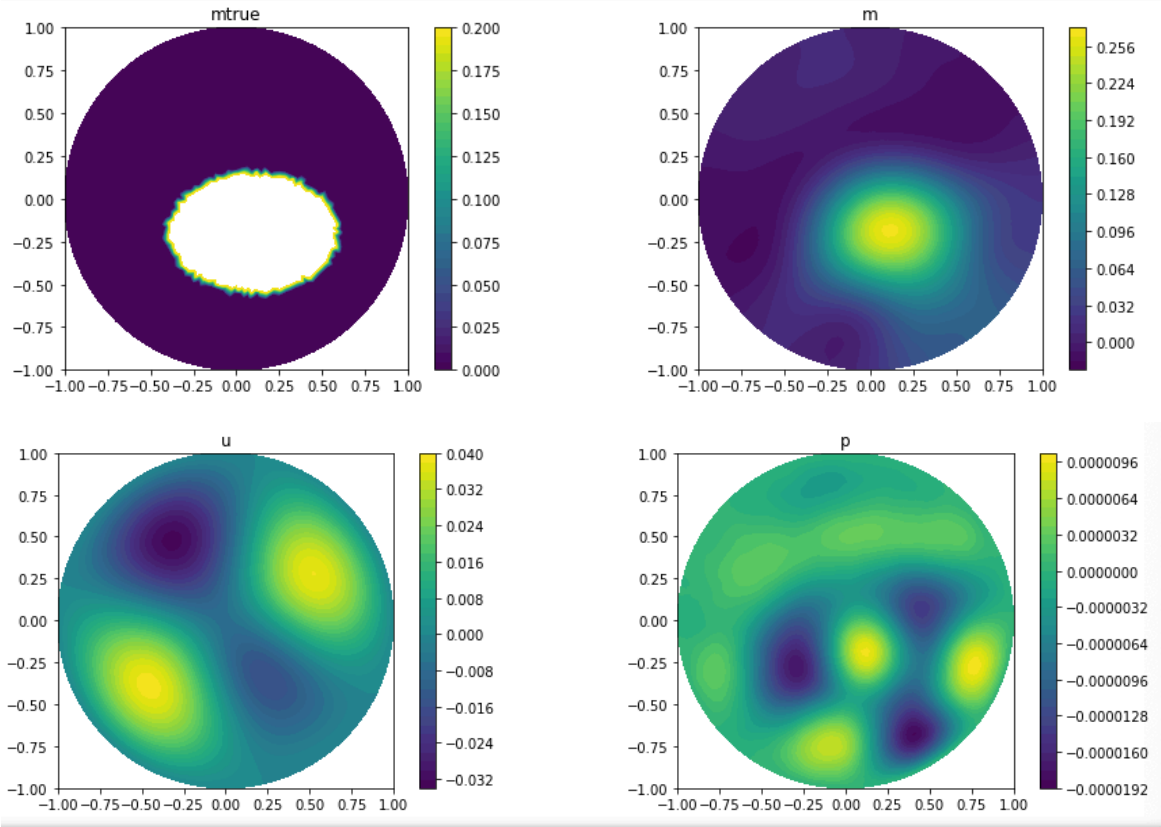


- Steepest descent did not converge in 1200 iterations for $k_0 = 10$. Parameters are not reconvered well. There is a significant difference between the parameters, and this we can safely assume that even with more iterations it would not converge.

(2) Solution of the inverse problem (TV regularization)

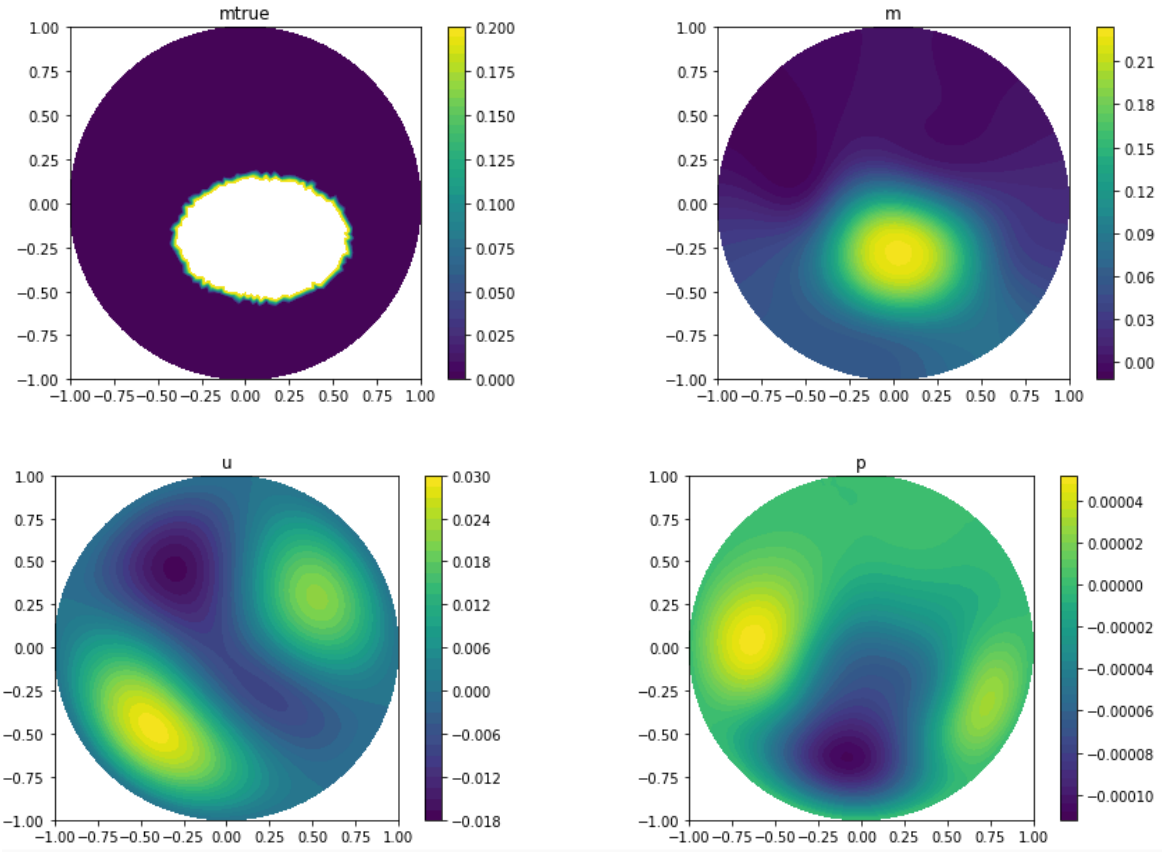
In [ ]:
```
# main function

if __name__ == "__main__":
    # input parameters
    mesh_file = "circle.xml"
    beta = 1e-9 # explore this here for 2nd question
    noise_level = 0.01
    nsources = 3
    k0 = 5#10.0
    iters = 1200
    alpha = 1
    regul = "tv"
    start_point = 0.0 # "low" or float # meaning at m = 0; else takes 8; or give float value
    graph_name = "Inversion results for case beta =1e-5, m(x)=0"
    precon = True

    # initialize the class
    helm = Helmholtz(start_point, mesh_file, beta, noise_level, nsources, k0, iters, alpha, reg
    # solve the problem
    helm.solution_flow()
```

- Steepest descent converged in 720 iterations



With $\delta = 0.01$ and $\beta = 10 - 9$, we initialize with an initial guess of m = 0. From the results, it is clear that the true reaction coefficient field is recovered very well with least oscillations, i.e. convergence in 720 iterations. The range of m is well captured. With better \beta value using Morozov discrepency criterion we can obtain even better convergence. Also, the edges are preserved due to preferential diffusion compared with isotropic diffusion of H1 regularization.

(3) Solution of the inverse problem with multiple source inverse problem



Overall better resolution, due to different sources acting independently. Furthermore, results seems more robust to noise, thus better convergence and results.

NOTE:please see the appendix for detailed code.

Problem-3: Inexact Newton-CG method for solving the Helmholtz inverse problem:

Same as Problem-2 but with Inexact Newton-CG method.

- $k_0 = 10$

From equation 1.22 and 1.23, but using the terms from question-2:

$$\mathcal{L}^H(m,u,p,\tilde{m},\tilde{u},\tilde{p}) = \beta\int_\Omega \nabla m\nabla\tilde{m} + k_0^2(u-u^{inc})(1-tanh^2(m))\tilde{m}pdx + \int_\tau \nabla u\nabla\tilde{p} - k_0^2(1-tanh$$
$$- k_0^2(1-tanh(m))\tilde{u}pdx$$

- Derivatie wrt to m :

$$\mathcal{H}(\tilde{m})\hat{m} = \delta_m\mathcal{L}^H = \beta\int_\Omega \nabla\tilde{m}.\nabla\hat{m} - k_0^2(u-u^{inc})(2tanh(m)(1-tanh^2(m)))\hat{m}\tilde{m}pdx + k_0^2(1-tanh$$
$$\in H_0^1(\Omega)$$

- Incremental Forward problem : Find $\tilde{m} \in H_0^1(\Omega)$:

$$\delta_p\mathcal{L}^H = \int_\Omega k_0^2(u-u^{inc})(1-tanh^2(m))\tilde{m}\hat{p}dx + \int_\Omega \nabla\tilde{u}\nabla\hat{p} - k_0^2(1-tanh(m))\tilde{u}\hat{p}dx; \forall\hat{p} \in H_0^1(\Omega$$

- Incremental Adjoint problem : Find $\tilde{p} \in H_0^1(\Omega)$:

$$\delta_u\mathcal{L}^H = \int_\Omega k_0^2(1-tanh^2(m))\tilde{m}\hat{u}pdx + \int_\tau \nabla\hat{u}\nabla\tilde{p} - k_0^2(1-tanh(m))\hat{u}\tilde{p} + \int_\Omega \tilde{u}\hat{u}dx; \forall\hat{u} \in H_0^1(\Omega$$

NOTE:Please see the appendix for detailed code implementation.

In [ ]:
```
# 3.1
# main function

if __name__ == "__main__":
    # input parameters
    mesh_file = "circle.xml"
    noise_level = 0.01
    nsources = 3
    k0 = 10
    iters = 120
    alpha = 1
    regul = "h1"
    start_point = 4.0 # "low" or float # meaning at m = 0; else takes 8; or give float value
    graph_name = "Inversion results for case beta =1e-5, m(x)=0"
    precon = True
    morozov = True

    for i in [0,1,2]:
        for beta in [1e-3, 1e-4, 1e-5, 1e-6, 1e-7,1e-8]:
            # initialize the class
            print("Beta value: ", beta)
            helm = Helmholtz_InCG(start_point, mesh_file, beta, noise_level, nsources, k0, iter
            # solve the problem
            helm.solution_flow()
```

(a) Implementing the Inexact Newton-CG method with for three refinement levels:

1. Table below summarizes the itertaions required for different beta values and refinement level:

Table-1: Iterations for different beta values and refinement levels

Refinement Level	Noise SD	β	Regularization	Newton Steps	CG Steps
0	1%	1.00E-03	Tikhonov	9	11
0	1%	1.00E-04	Tikhonov	9	10
0	1%	1.00E-05	Tikhonov	9	11
0	1%	1.00E-06	Tikhonov	9	10
0	1%	1.00E-07	Tikhonov	6	6
0	1%	1.00E-08	Tikhonov	4	4
1	1%	1.00E-03	Tikhonov	10	13
1	1%	1.00E-04	Tikhonov	9	10
1	1%	1.00E-05	Tikhonov	9	10
1	1%	1.00E-06	Tikhonov	9	10
1	1%	1.00E-07	Tikhonov	6	6
1	1%	1.00E-08	Tikhonov	4	4
2	1%	1.00E-03	Tikhonov	9	11
2	1%	1.00E-04	Tikhonov	9	11
2	1%	1.00E-05	Tikhonov	9	11
2	1%	1.00E-06	Tikhonov	6	8
2	1%	1.00E-07	Tikhonov	6	7
2	1%	1.00E-08	Tikhonov	4	5

The results observed in the table reflect several key trends concerning the influence of mesh refinement and regularization parameter β on the performance of an iterative solution approach, such as the Newton-CG method used for solving inverse problems. Increased Refinement Leads to More CG Iterations: As the refinement level increases from 0 to 2, the number of Conjugate Gradient (CG) steps tends to increase or remains roughly stable. This increase is expected because refining the mesh increases the number of degrees of freedom in the problem, leading to larger system matrices. Whereas, Newton Steps Remain Stable, The number of Newton steps required to achieve convergence does not show a consistent trend with increased mesh refinement. It tends to stabilize across different β values. This might indicate that the problem's nonlinearity (handled by Newton steps) is not significantly affected by the mesh density in terms of convergence, which is primarily governed by how well the linear systems (solved in each Newton step) are handled. HIgher β Values Lead to More Iterations, as β increases, the number of CG steps tends to increase.

NOTE: Please the see appendix for plots.

In [ ]:
```python
# 3.2.a
if __name__ == "__main__":
    # input parameters
    mesh_file = "circle.xml"
    noise_level = 0.01
    nsources = 3
    k0 = 10
    iters = 50
    alpha = 1
    regul = "h1"
    start_point = 4.0 # "low" or float # meaning at m = 0; else takes 8; or give float value
    graph_name = "Inversion results for case beta =1e-5, m(x)=0"
    precon = True
    refine_level = 2
    morozov = False


    betas = [1e-4]
    for i in [0,1,2]:
        for beta in betas:
            # initialize the class
            print("Beta value: ", beta)

            helm = Helmholtz_InCG(start_point, mesh_file, beta, noise_level, nsources, k0, iter
            # solve the problem
```

```
            helm.solution_flow()

    for k0 in [5,10,12.5]:
        for i in [0]:
            for beta in betas:
                # initialize the class
                helm = Helmholtz_InCG(start_point, mesh_file, beta, noise_level, nsources, k0,
                # solve the problem
                helm.solution_flow()
```

(b) Computing eigen values:

1. we fix the beta here and compute the eigen values for different refinement levels.

As the mesh is refined (i.e., more elements or finer discretization), the spectrum of the Hessian matrix generally becomes broader. This means that the eigenvalues spread over a larger range, with both very small and very large eigenvalues becoming more prominent. Finer meshes capture more details, leading to increased complexity in the Hessian's spectral properties.

In case of Coarser Meshes (level - 0), coarser meshe tend to simplify the problem, leading to a more condensed spectrum of eigenvalues. The Hessian matrix in coarser meshes, that does not capture all the nuances of smaller-scale variations, and thus make the problem numerically more stable but less accurate in capturing fine details (high cost).

With increasing refinement, the lowest eigenvalues of the Hessian (which are most problematic in terms of causing instability in the inverse problem) are typically increased by the regularization, thus making the problem more stable. However, as the mesh refines, new low eigenvalues emerges due to the higher resolution capturing more detailed modes of the system.

2. For varying wave number k_0:

As we increase the wavenumber, the obtaining the parameters becomes more challenging, same can be observed across all experiments, where convergence delays and parameters are poorly obtained. This is due to the fact that higher wave numbers can represent higher frequencies, the Hessian of such systems have more variability in its spectrum, potentially leading to larger and smaller eigenvalues which can affect the stability and convergence of numerical methods used to solve the inverse problem.

With higher wavenumber, we observe a broader spread in the eigenvalues of the Hessian. This spread indicates more extreme conditioning of the matrix, which could necessitate stronger regularization or different numerical treatment to ensure stability and accuracy.

Starting the solution of the inverse problem for a larger wavenumber with the solution from a smaller wavenumber as an initial guess speeds up the convergence. This is because of the continuity in the behavior of the system across different scales represented by wavenumber.

Furthermore, we observe that larger wavenumber introduces more rapid oscillations that are difficult to capture on a coarse mesh. And thus we observe more instability and slow convergence

In [ ]:
```
# Q3.3

# main function

if __name__ == "__main__":
    # input parameters
    mesh_file = "circle.xml"
    noise_level = 0.01
    nsources = 3
    k0 = 10
    iters = 120
    alpha = 1
    regul = "tv"
    start_point = 4.0 # "low" or float # meaning at m = 0; else takes 8; or give float value
    graph_name = "Inversion results for case beta =1e-5, m(x)=0"
    precon = True
    refine_level = 0
```

```
morozov = True

betas = [1e-3, 1e-4, 1e-5, 1e-6, 1e-7,1e-8]
for i in [0,1,2]:
    print("Refinement level:",i)
    for beta in betas:
        # initialize the class
        print("Beta value: ", beta)
        helm = Helmholtz_InCG(start_point, mesh_file, beta, noise_level, nsources, k0, iter
        # solve the problem
        helm.solution_flow()
```

(C) Implementing the Inexact Newton-CG method with for three refinement levels WITH TV:

1. Similar to part-1, we show table below that summarizes the itertaions required for different beta values and refinement level for TV:

Table-2: Iterations for different beta values and refinement levels with TV regularization

Refinement Level	Noise SD	β	Regularization	Newton Steps	CG Steps
0	1%	1.00E-03	TV	11	12
0	1%	1.00E-04	TV	13	14
0	1%	1.00E-05	TV	20	21
0	1%	1.00E-06	TV	16	16
0	1%	1.00E-07	TV	19	19
0	1%	1.00E-08	TV	14	16
1	1%	1.00E-03	TV	11	13
1	1%	1.00E-04	TV	13	14
1	1%	1.00E-05	TV	16	17
1	1%	1.00E-06	TV	18	18
1	1%	1.00E-07	TV	19	19
1	1%	1.00E-08	TV	19	19
2	1%	1.00E-03	TV	11	13
2	1%	1.00E-04	TV	9	10
2	1%	1.00E-05	TV	20	20
2	1%	1.00E-06	TV	16	16
2	1%	1.00E-07	TV	16	16
2	1%	1.00E-08	TV	15	15

TV Regularization require more iterations to converge, particularly for finer discretizations or lower values of the regularization parameter, as indicated by the generally higher number of conjugate gradient (CG) steps, we can observe the similar pattern in our tabel above. Furthermore there's more oscillations in the TV regularization, can ber observed in the plots. Convergence in case TV, is more sensitive to he choice of β. Optimal tuning of β is crucial in TV, as it heavily influences the balance between noise suppression and edge preservation.

Appendix has plots.

Problem-4: Inverse Problem for Burger's Equation:

Given:

We have 1-D Burgers' equation with viscosity field $m(x)$:

$$u_t + uu_x - (mu_x)_x = f \text{ in } (0, L) \times (0, T) \tag{4.1}$$

$$u(0, t) = u(L, t) = 0 \text{ for all } t \in [0, T] \tag{4.2}$$

$$u(x,0) = 0 \text{ for all } x \in [0,L] \tag{4.3}$$

We are given the observaton $d = d(x,t)$ for $t \in [T_1, T]$, and $T_1 > 0$. The inversion for $m(x)$ is given by the minimization problem:

$$\min_{m \in \mathcal{M}} \mathcal{F}(m) := \frac{1}{2}\int_{T_1}^{T}\int_0^L (u(x,t;m) - d(x,t))^2 dxdt + \frac{\beta}{2}\int_0^L (\frac{dm}{dx})^2 dx \tag{4.4}$$

where $\mathcal{M}$ is the set of admissible viscosity fields, $\mathcal{M} = H^1(0,L)$, and $\beta > 0$.

(a) Weak form (forward problem)$:

We can write the weak form of the Burgers' equation (forward) by integrating with a test function $p(x,t) : [0,L]X[0,T]$ -> $\in \mathbb{R}$. The solution space $\mathcal{U}$ becomes:

$$\mathcal{U} = \{u \in H^1(0,L) \times L^2(0,T) : u(0,t) = u(L,t) = 0 \text{ and } u(x,0) = 0\} \tag{4.5}$$

And space for the adjoint variable $\mathcal{P}$:

$$\mathcal{P} = \{p \in H^1[0,L] \times L^2[0,T] : p(0,t) = p(L,t) = 0\} \tag{4.6}$$

With homogenous dirichlet boundary conditions, the weak form of the Burgers' equation is (integrating PDE with p):

$$\int_0^T \int_0^L p(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - \frac{\partial}{\partial x}(m\frac{\partial u}{\partial x}) - f)dxdt = 0 \tag{4.7}$$

Integrating by parts for the diffusion terms, we get:

Find $u \in \mathcal{U}$ such that:

$$\int_0^T \int_0^L p(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - f) + \frac{\partial p}{\partial x}(m\frac{\partial u}{\partial x})dxdt = 0, \forall p \in \mathcal{P} \tag{4.8}$$

Note that the homogenous dirichlet boundary conditions are $p(0,t) = p(L,t) = 0$

(b) Adjoint and Gradient of Langrangian $\mathcal{L}$:

The Langrangian functional for the Burgers' equation is given by combining the objective function and the weak form of the PDE:

$$\mathcal{L}(m,u,p) = \frac{1}{2}\int_{T_1}^{T}\int_0^L (u(x,t) - d(x,t))^2 dxdt + \frac{\beta}{2}\int_0^L (\frac{dm}{dx})^2 dx + \int_0^T \int_0^L p(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} - f) +$$

- For Adjoint problem:

$$\delta_u \mathcal{L} = 0, \forall \hat{u} \in \mathcal{U} \tag{4.10}$$

We know that weak form of adjoint problem can be obtained by frechet derivative of the Langrangian wrt u and setting it to zero.

Find $p \in \mathcal{P}$ such that:

$$\delta_u \mathcal{L}(u;\hat{u}) = \int_{T_1}^{T}\int_0^L \hat{u}(u-d)dxdt + \int_0^T \int_0^L p(\frac{\partial \hat{u}}{\partial t} + \hat{u}\frac{\partial u}{\partial x} + u\frac{\partial \hat{u}}{\partial x}) + \frac{\partial p}{\partial x}(m\frac{\partial \hat{u}}{\partial x})dxdt = 0, \forall \hat{u} \in$$

For the strong form of the adjoint problem, we integrate by parts:

$$\int_{T_1}^{T}\int_0^L \hat{u}(u-d)dxdt + \int_0^T \int_0^L \hat{u}[-\frac{\partial p}{\partial t} + p\frac{\partial u}{\partial x} - \frac{\partial (pu)}{\partial x} - \frac{\partial}{\partial x}(m\frac{\partial p}{\partial x})]dxdt + \int_0^T [p\hat{u}]_0^T dx + \int_0$$

Now we use the dirichlet boundary conditions to simplify the equation, (note that the initial condition is also $u(x,0) = 0$):

$$p\frac{\partial u}{\partial x} - \frac{\partial (pu)}{\partial x} \implies p\frac{\partial u}{\partial x} - p\frac{\partial u}{\partial x} - u\frac{\partial p}{\partial x} \implies -u\frac{\partial p}{\partial x} \tag{4.13}$$

Now for the strong form of adjoint problem, we get:

$$-\frac{\partial p}{\partial t} - u\frac{\partial p}{\partial x} - \frac{\partial}{\partial x}(m\frac{\partial p}{\partial x}) = -(u-d) \text{ in } (0,L)X(T_1,T) \tag{4.14}$$

$$p(0,t) = p(L,t) = 0 \text{ for all } t \in [0,T] \tag{4.15}$$

$$p(x,T) = 0 \text{ for all } x \in [0,L] \tag{4.16}$$

- Gradient of $\mathcal{L}$ wrt m:

$$\delta_m \mathcal{L} = (\mathcal{G}(m), \hat{m}), \forall \hat{m} \in \mathcal{M} \tag{4.17}$$

We the same langranian as in 4.9, and take frechet derivative wrt m:

$$(\mathcal{G}, \hat{m}) = \beta \int_0^L \frac{dm}{dx}\frac{d\hat{m}}{dx}dx + \int_0^T \int_0^L \hat{m}\frac{\partial p}{\partial x}\frac{\partial u}{\partial x}dxdt = 0, \forall \hat{m} \in \mathcal{M} \tag{4.18}$$

For the strong form of the gradient, we integrate by parts for the first term:

$$(\mathcal{G}, \hat{m}) = \int_0^L \hat{m}(-\beta\frac{d^2m}{dx^2} + \int_0^T \frac{\partial p}{\partial x}\frac{\partial u}{\partial x}dt)dx + [\beta\frac{dm}{dx}\hat{m}]_0^L \tag{4.19}$$

Thus the strong form of the gradient is:

$$-\beta\frac{d^2m}{dx^2} + \int_0^T \frac{\partial p}{\partial x}\frac{\partial u}{\partial x}dt, (x,t) \text{ in } (0,L) \times (0,T) \tag{4.20}$$

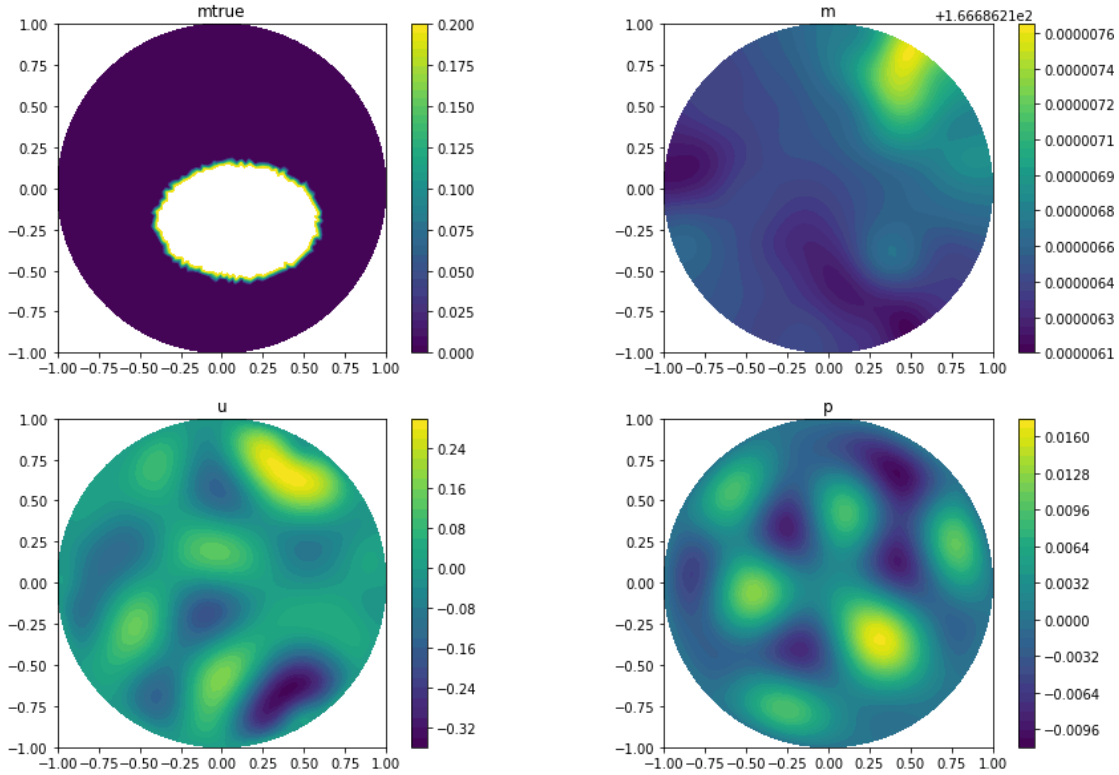
$$\beta\frac{dm}{dx}, \text{ on } x = 0, L \tag{4.21}$$

Appendix

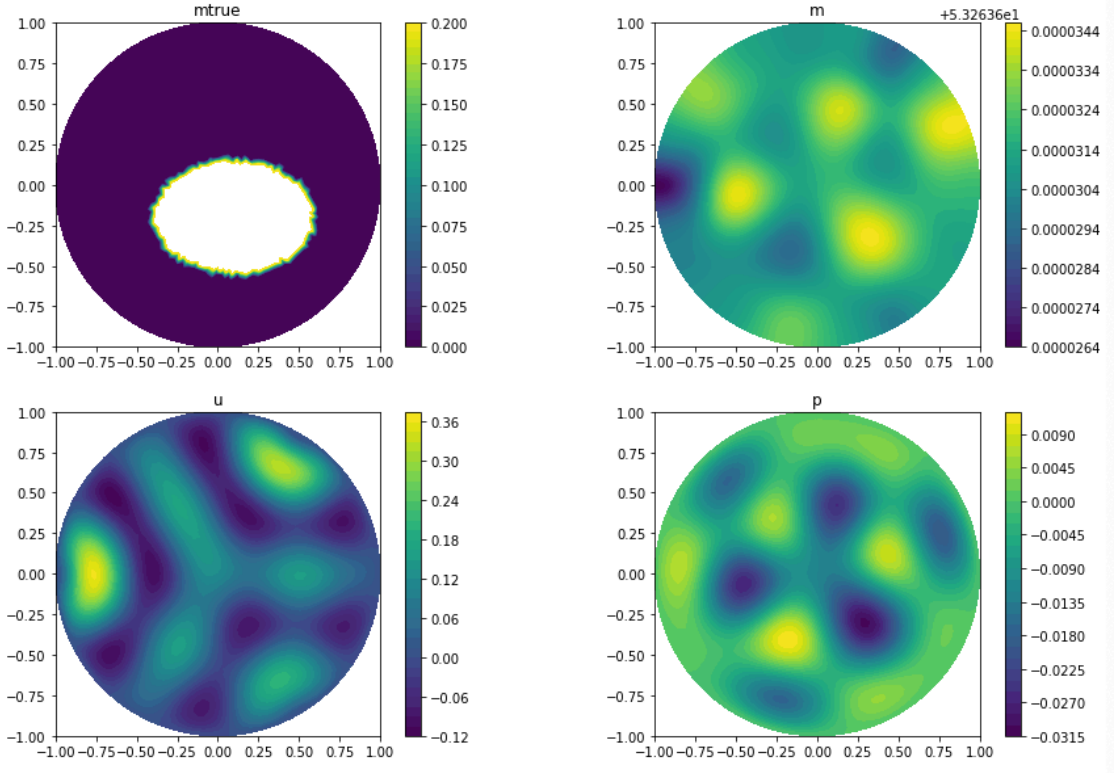
Question 3 :

1. Part - 1

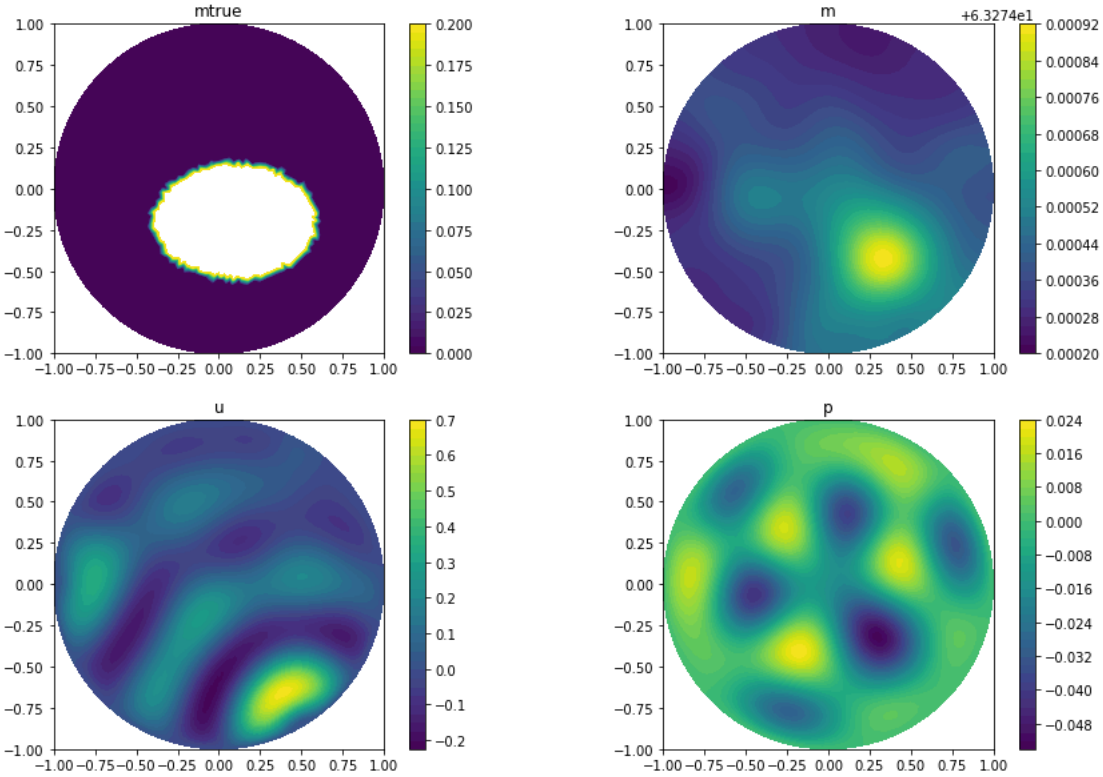
- Level 0:
  - $\beta = 10^{-3}$



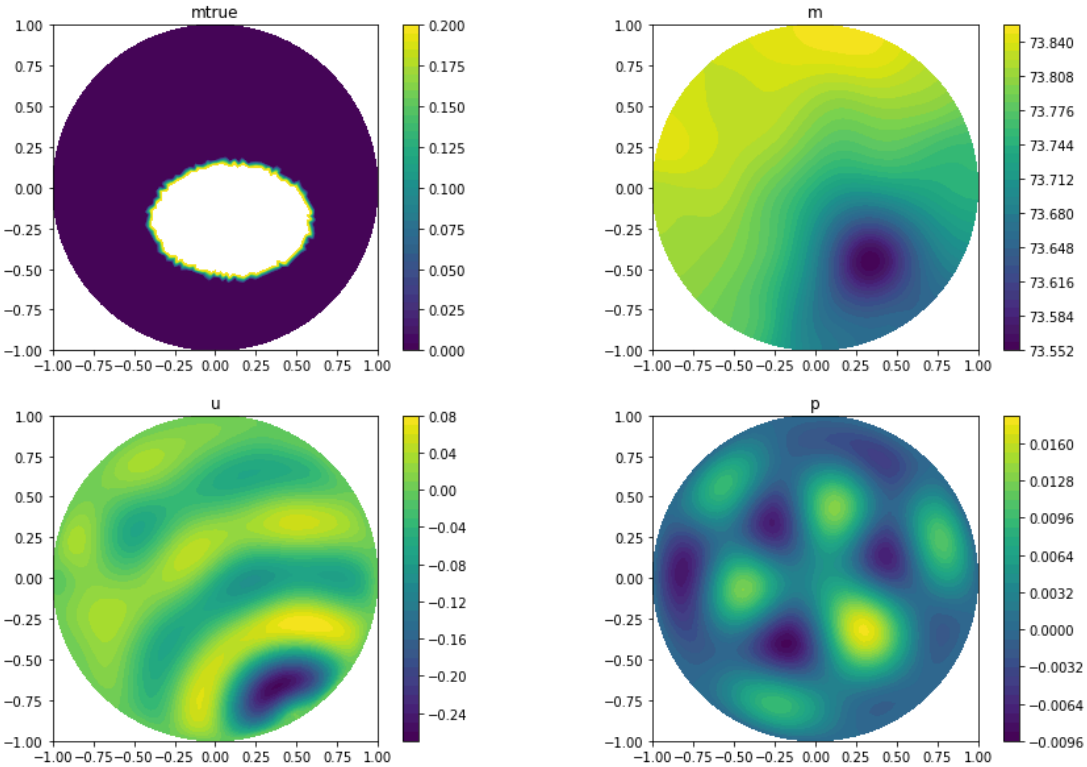
- $\beta = 10^{-4}$



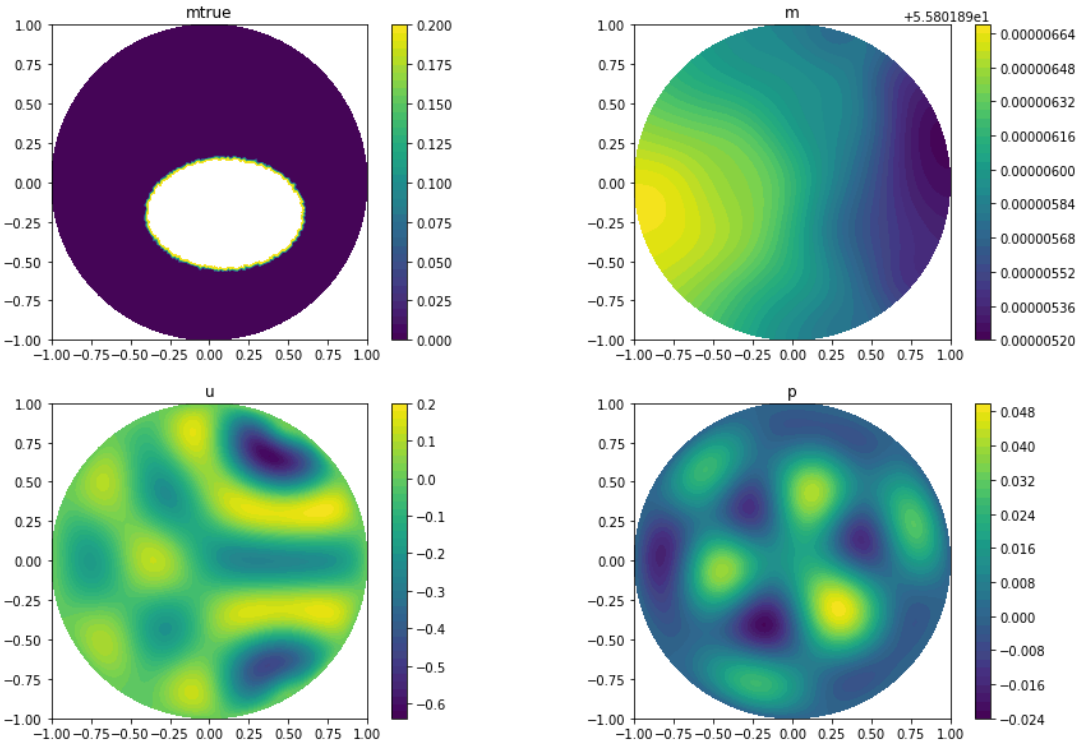
- $\beta = 10^{-6}$



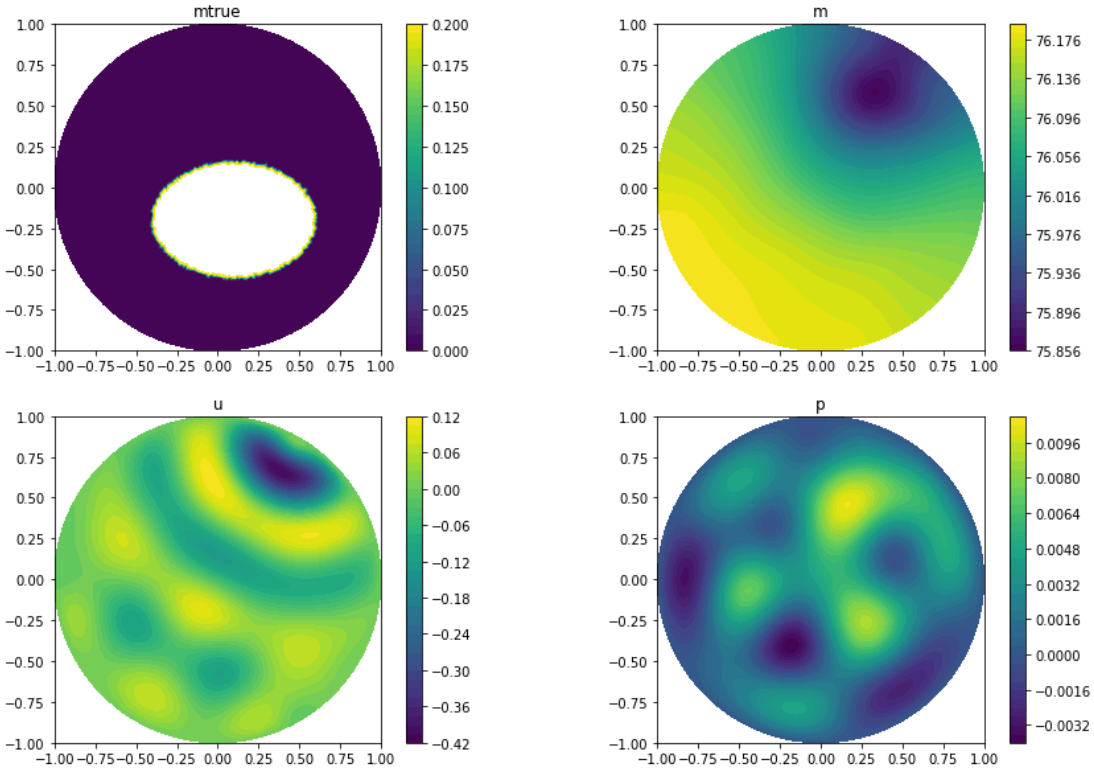
- $\beta = 10^{-8}$



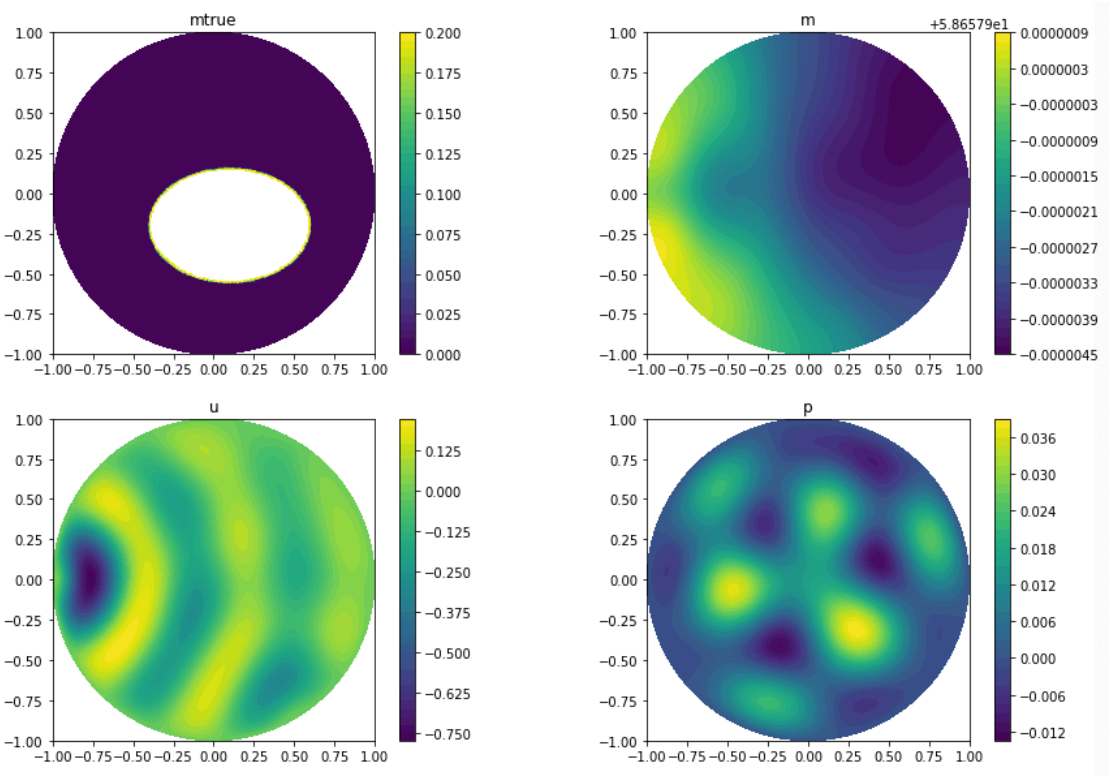
- Level 1:
  - $\beta = 10^{-3}$



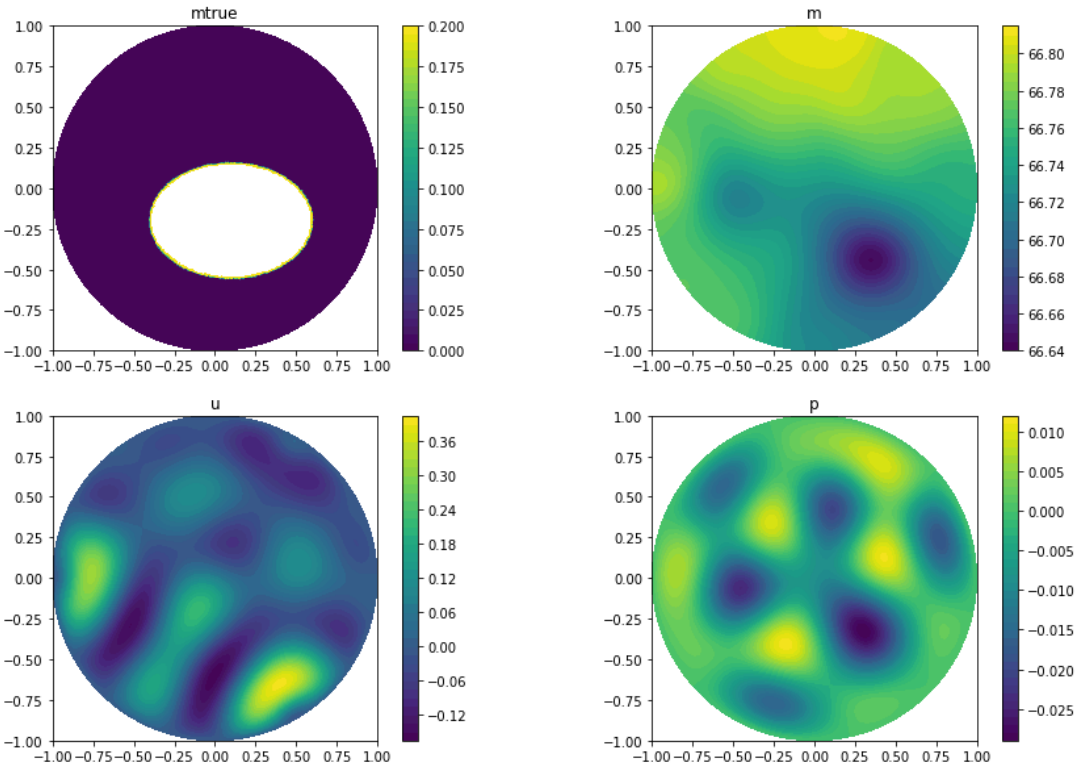
- $\beta = 10^{-8}$



- Level 2:
  - $\beta = 10^{-3}$

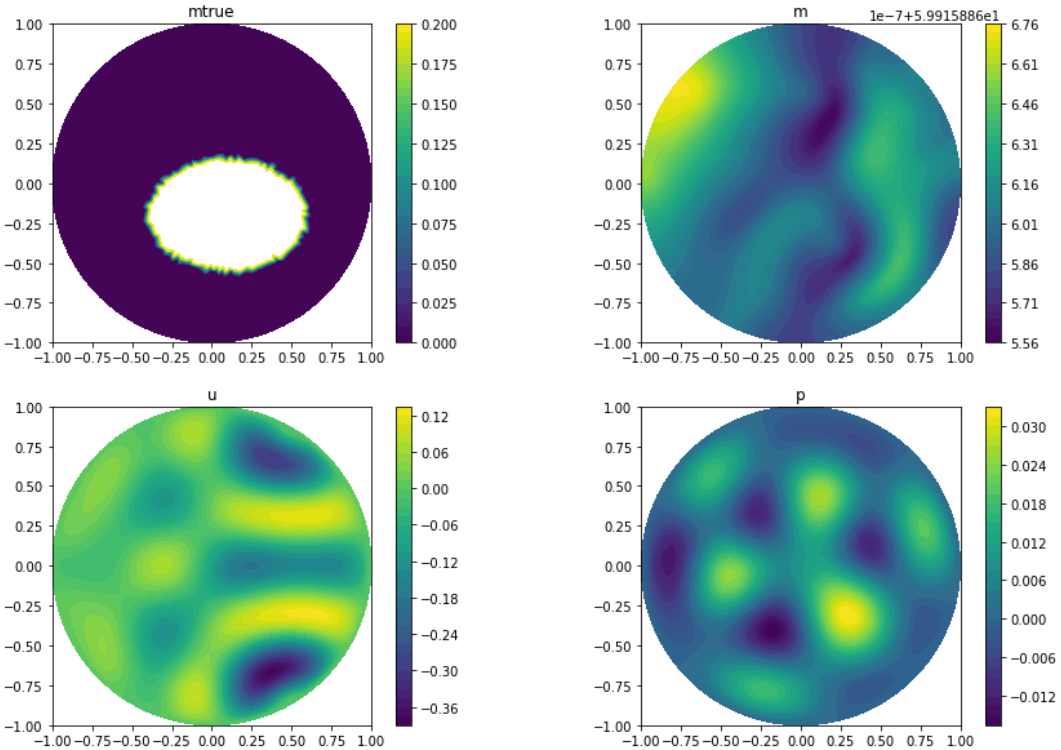


- $\beta = 10^{-8}$

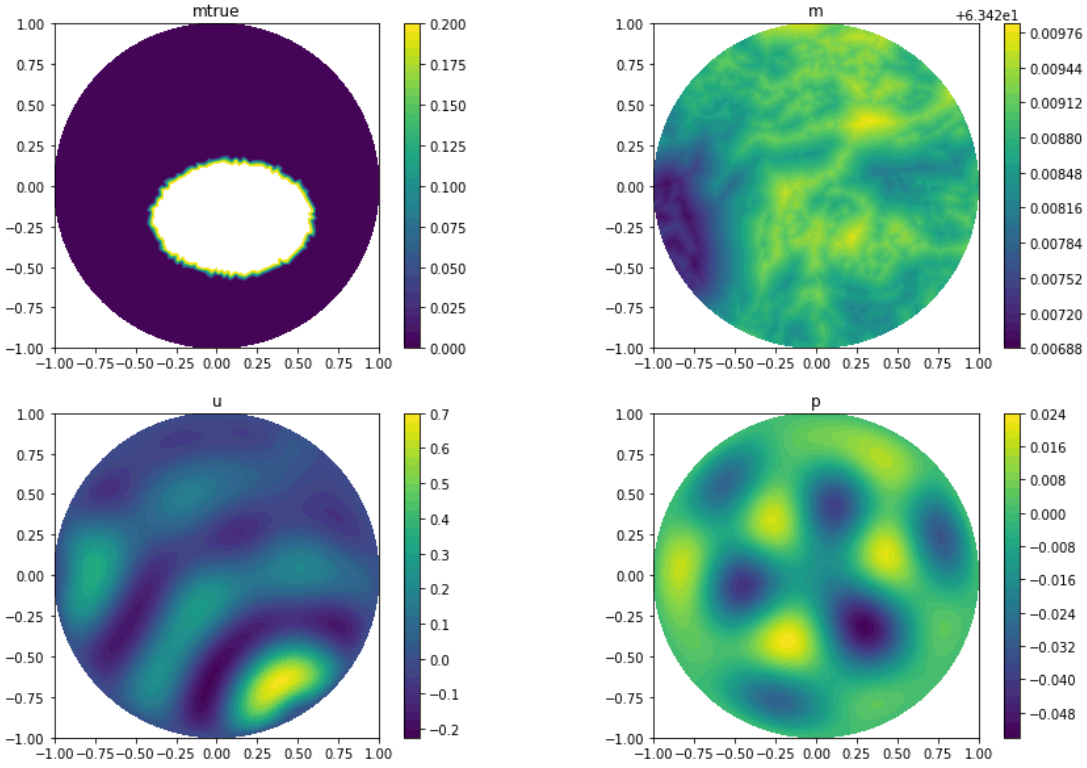


2. Part - 3

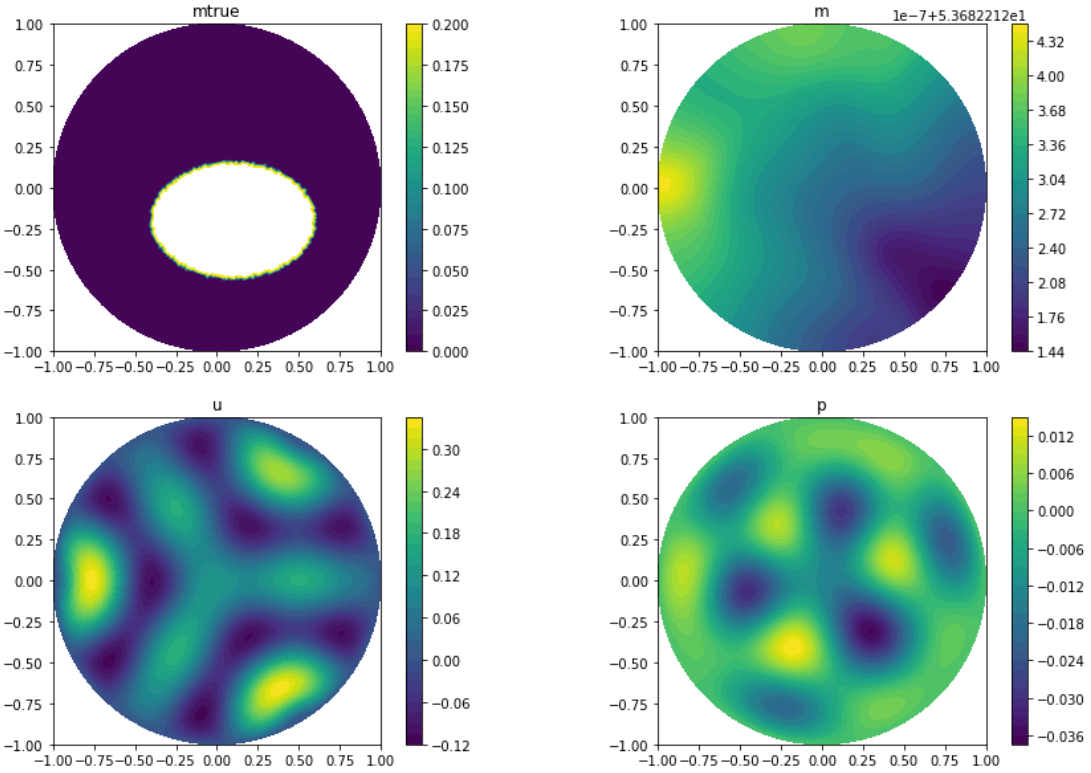
- Level 0:
  - $\beta = 10^{-3}$



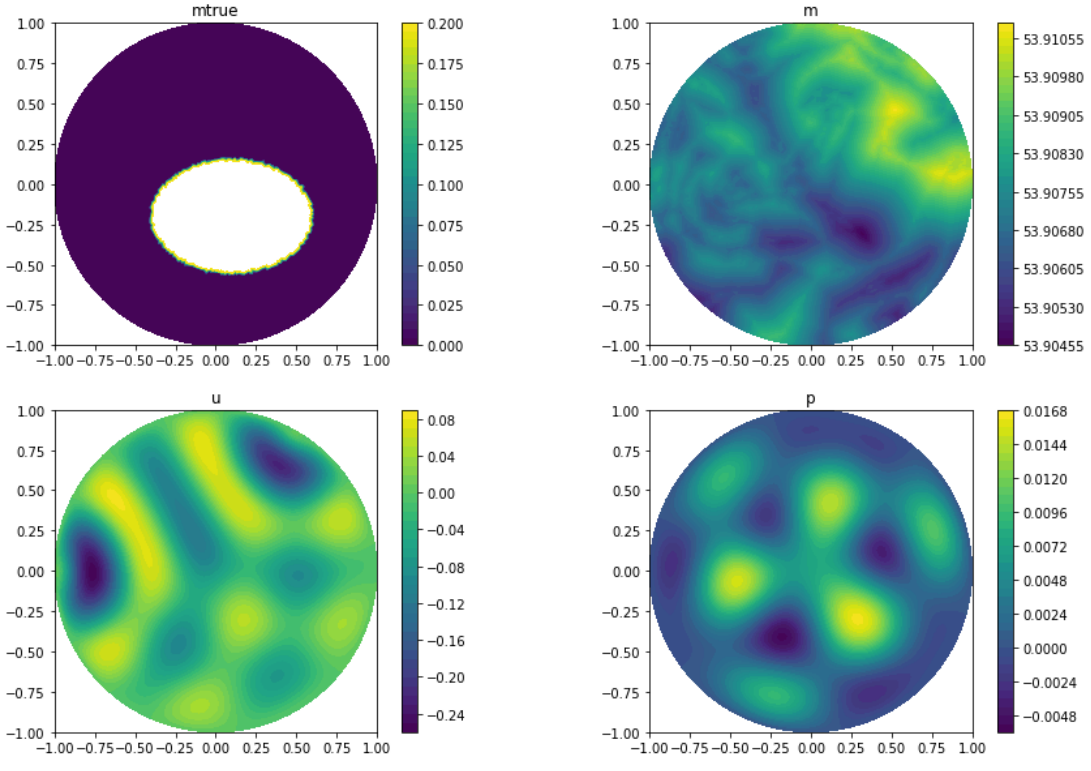
- $\beta = 10^{-8}$



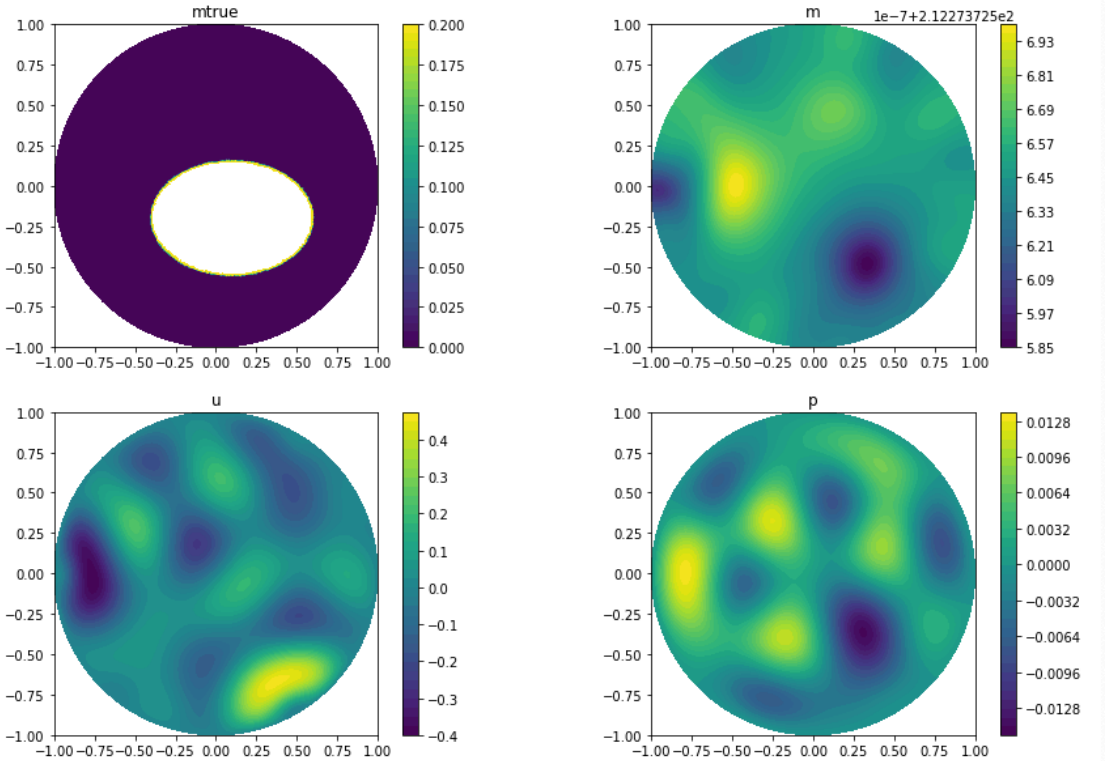
- Level 1:
  - $\beta = 10^{-3}$



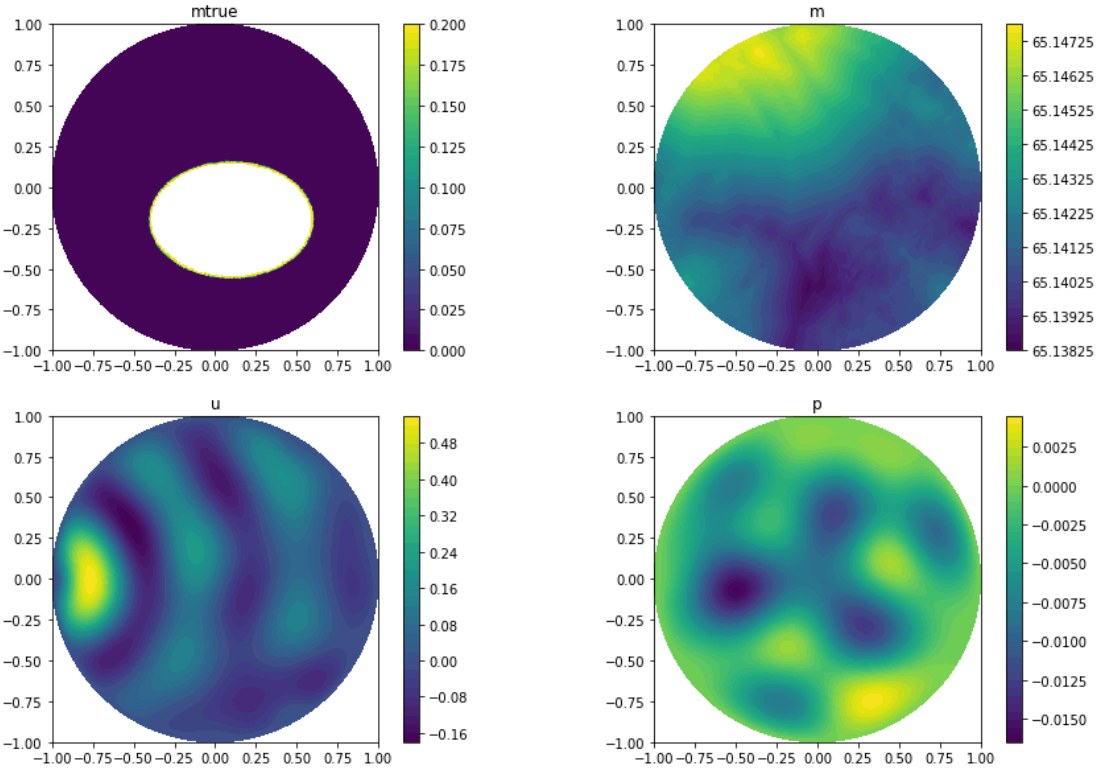
- $\beta = 10^{-8}$



- Level 2:
  - $\beta = 10^{-3}$



- $\beta = 10^{-8}$



In [ ]:
```
# Install the necessary packages
# try loading hiPPylib if it fails install it
try:
    import hiPPylib
    import FEniCS
    import matplotlib
    import dolfin
except ImportError:
    print(f"Issue with some library, installing the necessary packages.")
    """
    !pip install --upgrade --user matplotlib --yes
    !pip install --upgrade --user numpy --yes
    !pip install --upgrade --user pandas --yes
```

```
    !pip install --upgrade --user scikit-learn --yes
    !pip install --upgrade --user scipy --yes
    #!pip install --upgrade --user FEniCS --yes
    #!pip install --upgrade --user hiPPylib --yes
    #!pip install --upgrade --user dolfin --yes
    """
```

In [ ]:

```
""" Code for Q-2
Computational and Variational Methods for Inverse Problems (Spring 2024)
Assignment-04 -- Question-2

--->  Steepest Descent method for solving Helmholtz inverse problem with domain at unit Circle.

Author: Shreshth Saini (saini.2@utexas.edu)
Date: 29th April 2024
"""

# Question 2
#-----------------------------------------------------------------------------------------------

# Compute libraries
import dolfin as dl
import ufl
from hippylib import nb
import numpy as np
import mshr

# general libraries
import matplotlib.pyplot as plt
%matplotlib inline
import logging

# suppress warnings
import warnings
warnings.filterwarnings("ignore")
#-----------------------------------------------------------------------------------------------

# initialize the logger
logging.getLogger("FFC").setLevel(logging.ERROR)
logging.getLogger("UFL").setLevel(logging.ERROR)
dl.set_log_active(False)


np.random.seed(seed = 22)

class Helmholtz():
    def __init__(self,  starting_location, mesh_file:str, beta:float, noise_level:float, nsourc
        self.mesh_file = mesh_file
        self.beta = beta
        self.noise_level = noise_level
        self.regularization = regularization
        self.starting_point = starting_location
        self.print_name = print_name
        self.plot = plot_level
        self.k0 = k0
        self.k0squared = dl.Constant(k0*k0)
        self.nsources = nsources
        self.max_iter = max_iter
        self.alpha = alpha
        self.precon_solver = precon_solver

    # defining and solving - flow
    def solution_flow(self):
        # obtaining mesh
        self.get_mesh(self.mesh_file)

        # m and u
        self.Vm = dl.FunctionSpace(self.mesh, "Lagrange", 1)
        self.Vu = dl.FunctionSpace(self.mesh, "Lagrange", 2)

        # The true parameter
        self.mtrue_str = '.2*((x[0]-.1)*(x[0]-.1) + 2.*(x[1]+.2)*(x[1]+.2) < 0.25)'
        self.mtrue = dl.interpolate(dl.Expression(self.mtrue_str, degree=5), self.Vm)

        # if plot level is full
```

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

        # source initialization
        self.source_function(self.nsources, self.k0)

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

        # define trail and test functions
        self.u_trial, self.m_trial, self.p_trial = dl.TrialFunction(self.Vu), dl.TrialFunction(
        self.u_test, self.m_test, self.p_test = dl.TestFunction(self.Vu), dl.TestFunction(self.


        # initialize input functions
        u0 = dl.Constant(0.0) # difussion coefficient
        # already initialized the u_inc in source function
        # k0^2 is also defined


        # boundary conditions - define here
        self.bc_state = self.boundary_condition(u0)
        self.bc_adj = self.boundary_condition(dl.Constant(0.0))

        # weak form of the pde
        self.utrue = dl.Function(self.Vu)

        # defining the state equation with the true parameter
        phi_true = self.pde_varf(self.utrue, self.mtrue, self.u_test)
        #a_true = ufl.lhs(self.pde_varf(self.u_trial,self.mtrue,self.u_test))
        #L_true = ufl.rhs(self.pde_varf(self.u_trial,self.mtrue,self.u_test))

        # solving the forward problem;
        #A_true, b_true = dl.assemble_system(a_true, L_true)

        #dl.solve(A_true, self.utrue.vector(), b_true)
        dl.solve(phi_true==0, self.utrue, self.bc_state, solver_parameters={"newton_solver": {"


        # taken from Jupyter notebook provided
        Avarf = dl.inner(self.u_trial, self.u_test)*dl.dx
        self.A_true = dl.assemble(Avarf)

        # observed data
        self.d = dl.Function(self.Vu)
        self.d.assign(self.utrue)

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

        # apply noise to bc_adj
        self.bc_adj.apply(noise)

        # add noise to the state variable
        self.d.vector().axpy(1., noise)

        # plot
        if self.plot == "full":
            nb.multi1_plot([self.utrue, self.d], ["State solution with mtrue", "Synthetic obser
            plt.show()


        # weak form for settnig up the state equation # double check this line 79
        self.phi_state = self.pde_varf(self.u, self.m, self.p_test)
        #a_phi_state = ufl.lhs(self.pde_varf(self.u_trial,self.m,self.u_test))
        #L_phi_state = ufl.rhs(self.pde_varf(self.u_trial,self.m,self.u_test))
```

```
        #self.A_phi_state, self.b_phi_state = dl.assemble_system(a_phi_state, L_phi_state)

        """
        self.phi_state = ufl.inner(ufl.grad(self.u), ufl.grad(self.p_test))*ufl.dx \
                            -self.k0squared*(dl.Constant(1.) - ufl.tanh(self.m) )*self.u*self.p
                            - self.k0squared*ufl.tanh(self.m)*self.u_inc*self.p_test*ufl.dx
        """

        # now the weak form for the adjoint equation
        self.phi_adj = self.pde_adj(self.u, self.m, self.u_test)
        #a_phi_adj = ufl.lhs(self.pde_adj(self.p_trial,self.m,self.p_test))
        #L_phi_adj = ufl.rhs(self.pde_adj(self.p_trial,self.m,self.p_test))
        # solving the forward problem;
        #self.A_phi_adj, self.b_phi_adj = dl.assemble_system(a_phi_adj, L_phi_adj)

        """
        self.phi_adj = (self.u - self.d)*self.u_test*ufl.dx \
                        + ufl.inner(ufl.grad(self.p), ufl.grad(self.u_test))*ufl.dx \
                        - self.k0squared*(dl.Constant(1.) - ufl.tanh(self.m))*self.p*self.u_tes
        """

        # Now for the gradient (weak form)
        delta = 0.01
        # second term in the gradient
        self.CTvarf = -self.k0squared*(dl.Constant(1.) - ufl.tanh(self.m)**2)*(-self.u + self.u

        # the regulariation term (based on type of regularization)
        if self.regularization == "h1":
            self.gradRvarf = dl.Constant(self.beta)*(dl.inner(dl.grad(self.m), dl.grad(self.m_t

        elif self.regularization == "tv":
            self.gradRvarf = dl.Constant(self.beta)*dl.inner(dl.grad(self.m), dl.grad(self.m_te
                            /dl.sqrt(dl.inner(dl.grad(self.m), dl.grad(self.m)) + dl.Constant(d

        else:
            print(f"Regularization type not implemented.")
            return None

        # mass matrix assembly in parameter space
        Mvarf = dl.inner(self.m_trial, self.m_test)*dl.dx
        self.M = dl.assemble(Mvarf)


        #  matrix in parameter space
        Kvarf    = ufl.inner(self.m_trial, self.m_test) * ufl.dx + ufl.inner(ufl.grad(self.m_tr
        self.K = dl.assemble(Kvarf)

        ######## check finite diff grad
        #self.check_grad_error()

        # initial guess for the parameter
        if self.starting_point == 'low':
            print("Starting with m=0")
            m0 = dl.interpolate(dl.Constant(0.00), self.Vm)
            self.m.assign(m0)
        # if starting point is float then assign that value
        elif isinstance(self.starting_point, float)  or isinstance(self.starting_point, int):
            print("Starting m with given float: {}".format(self.starting_point))
            m0 = dl.interpolate(dl.Constant(self.starting_point), self.Vm)
            self.m.assign(m0)
        else:
            print("Starting with m = 8")
            m0 = dl.interpolate(dl.Constant(8.0), self.Vm)
            self.m.assign(m0)


        # solve the state equation
        dl.solve(self.phi_state == 0, self.u, self.bc_state, solver_parameters={"newton_solver"
        #dl.solve(self.A_phi_state, self.u.vector(), self.b_phi_state)

        # evaluate cost
        cost_old,_,_ = self.cost(self.u, self.d, self.m, self.beta)

        if self.plot == "full":
```

```
            plt.figure(figsize=(15,5))
            plt.subplot(121)
            dl.plot(self.m,title="m0", vmin=self.mtrue.vector().min(), vmax=self.mtrue.vector()
            plt.subplot(122)
            dl.plot(self.u, title="u(m0)")
            plt.show()


        # solving using the steepest descent method
        count, cost_new, misfit_new, reg_new = self.steepest_descent(cost_old, alpha = self.alp

        # plot the solution
        self.plot_solution(self.print_name)

        # return the solution
        Mstate = dl.assemble(self.u_trial*self.u_test*dl.dx)
        noise_norm2 = noise.inner(Mstate*noise)
        return self.Vm.dim(), count, noise_norm2, cost_new, misfit_new, reg_new



    # generate mesh or load from file
    def get_mesh(self, mesh_file=None):
        if mesh_file is None:
            self.mesh = self.generate_mesh()
        else:
            try:
                self.mesh = dl.Mesh(mesh_file)
            except:
                print(f"Mesh file not found. Trying in current directory... ")
                try:
                    self.mesh = dl.Mesh(mesh_file.split("/")[-1])
                except:
                    print(f"Mesh file not found. Generating mesh...")
                    self.mesh = self.generate_mesh()

    def generate_mesh(self, type="Circle"):
        # create mesh
        if type == "Circle":
            domain = mshr.Circle(dl.Point(0,0), 1)
            mesh = mshr.generate_mesh(domain, 50)
        else:
            print(f"Mesh type not implemented.")
            return None
        return mesh

    def source_function(self, nsources=3, k0=5.):
        # initialize input functions
        theta_view = np.linspace(-np.pi, np.pi, nsources, endpoint=False)
        radius = 1.05
        s0 = radius*np.cos(theta_view)
        s1 = radius*np.sin(theta_view)

        u_is = []
        # expression is predefined; does not need to be changed unless the source is changed
        for i in np.arange(nsources):
            distance_source = dl.Expression("std::sqrt((x[0]-s0)*(x[0]-s0) + (x[1]-s1)*(x[1]-s1
                                            s0 = s0[i], s1 = s1[i], degree=1)
            u_i_exp = dl.Expression("-std::cos(k0*distance_source)/(4.*DOLFIN_PI*distance_sourc
            u_is.append(dl.interpolate(u_i_exp, self.Vu) )

        if self.plot == "full":
            plt.figure(figsize=(15,3.3))
            nb.plot(u_is[0], subplot_loc=131, mytitle="First source")
            nb.plot(u_is[1], subplot_loc=132, mytitle="Second source")
            nb.plot(u_is[2], subplot_loc=133, mytitle="Third source")
            plt.show()


        # final source encoding; combining the sources
        encoding   = np.random.normal(size=nsources)
        # u_inc used in equation
        u_inc = dl.Function(self.Vu)

        for i, u_i in enumerate(u_is):
```

```
            u_inc.vector().axpy(encoding[i], u_i.vector())

        if self.plot == "full":
            plt.figure(figsize=(7.5,5))
            nb.plot(u_inc, mytitle="Super source")
            plt.show()

        self.u_inc = u_inc

    def pde_varf(self, u,m,p):
        return ufl.inner(ufl.grad(u), ufl.grad(p))*ufl.dx \
                -self.k0squared*(dl.Constant(1.) - ufl.tanh(m) )*u*p*ufl.dx \
                - self.k0squared*ufl.tanh(m)*self.u_inc*p*ufl.dx

    # based on this we make the adjoint equation
    def pde_adj(self, u,m,p):
            return (u - self.d)*p*ufl.dx \
                    + ufl.inner(ufl.grad(p), ufl.grad(self.p))*ufl.dx \
                    - self.k0squared*(dl.Constant(1.) - ufl.tanh(m))*p*self.p*ufl.dx

    # boundary condition
    def boundary_condition(self, u):
        def boundary(x, on_boundary):
            return on_boundary
        # defining the boundary condition
        bc = dl.DirichletBC(self.Vu, u, boundary)
        return bc

    # cost function
    def cost(self, u, d, m, beta):
        reg = 0.5 * beta * dl.assemble( (ufl.inner(dl.grad(m), dl.grad(m)))*ufl.dx )
        #reg = 0.5* beta * dl.assemble(dl.sqrt(dl.inner(dl.grad(m), dl.grad(m)))*dl.dx)
        misfit = 0.5 * dl.assemble( (u-d)**2*ufl.dx)
        return [reg + misfit, misfit, reg]

    # check grad error
    def check_grad_error(self):
        m0 = dl.interpolate(dl.Constant(0.0), self.Vm )
        n_eps = 32
        eps = np.power(2., -np.arange(n_eps))
        err_grad = np.zeros(n_eps)

        self.m.assign(m0)

        #Solve the fwd problem and evaluate the cost functional
        dl.solve(self.phi_state == 0, self.u, self.bc_state, solver_parameters={"newton_solver"

        c0, _, _ = self.cost(self.u, self.d, self.m, self.beta)


        # Solve the adjoint problem and evaluate the gradient
        dl.solve(self.phi_adj == 0, self.p, self.bc_adj, solver_parameters={"newton_solver": {"

        # evaluate the  gradient
        grad0 = dl.assemble(self.CTvarf+self.gradRvarf)

        # Define an arbitrary direction m_hat to perform the check
        mtilde = dl.Function(self.Vm).vector()
        mtilde.set_local(np.random.randn(self.Vm.dim()))
        mtilde.apply("")
        mtilde_grad0 = grad0.inner(mtilde)

        for i in range(n_eps):
            self.m.assign(m0)
            self.m.vector().axpy(eps[i], mtilde)

            dl.solve(self.phi_state == 0, self.u, self.bc_state, solver_parameters={"newton_sol

            cplus, _, _ = self.cost(self.u, self.d, self.m, self.beta)

            err_grad[i] = abs( (cplus - c0)/eps[i] - mtilde_grad0 )

        plt.figure()
        plt.loglog(eps, err_grad, "-ob", label="Error Grad")
        plt.loglog(eps, (.5*err_grad[0]/eps[0])*eps, "-.k", label="First Order")
```

```
        plt.title("Finite difference check of the first variation (gradient)")
        plt.xlabel("eps")
        plt.ylabel("Error grad")
        plt.legend(loc = "upper left")
        plt.show()




    # solve the system
    def steepest_descent(self, cost_old, tol=1e-4, backtrack_maxiter=20, c_armijo=1e-5, alpha =
        """
        tol         # Relative tolerance on the gradient norm
        maxiter     # Maximum number of iterations (we need a lot!)
        print_any   # We will only print progress on screen every few iterations
        plot_any    # We will plot the current solution every few iterations
        c_armijo    # The Armijo constant to ensure sufficient descent
        alpha       # The initial step-size
        """
        # initialize iter counters
        count = 0
        converged = False

        # initializations
        g = dl.Vector()
        # #Select whether to use the L^2 inner product (mass matrix M)
        # or to precondition steepest descent using the H^1 inner product (stiffness matrix K)
        if self.precon_solver:
            g = dl.Vector()
            self.K.init_vector(g,0)
            P = self.K
        else:
            self.M.init_vector(g,0)
            P = self.M

        m_prev = dl.Function(self.Vm)

        print( "Nit  cost          misfit        reg         ||grad||       alpha  N backtrack"
        print( "-------------------------------------------------------------------------------
        while count <  self.max_iter and not converged:
            print("Count", count)

            # solve the adoint problem
            dl.solve(self.phi_adj == 0, self.p, self.bc_adj,  solver_parameters={"newton_solver
            #dl.solve(self.A_phi_adj, self.p.vector(), self.b_phi_adj)

            # evaluate the  gradient
            MG = dl.assemble(self.CTvarf + self.gradRvarf)
            dl.solve(P, g, MG)

            # calculate the norm of the gradient
            grad_norm2 = g.inner(MG)
            gradnorm = np.sqrt(grad_norm2)

            if count == 0:
                gradnorm0 = gradnorm

            # linesearch
            it_backtrack = 0
            m_prev.assign(self.m)
            backtrack_converged = False
            for it_backtrack in range(backtrack_maxiter):

                self.m.vector().axpy(-alpha, g)

                # print("solving forward prob")
                #dl.solve(self.A_phi_state, self.u.vector(), self.b_phi_state)
                dl.solve(self.phi_state == 0, self.u, self.bc_state, solver_parameters={"newton_

                # evaluate cost
                [cost_new,misfit_new,reg_new] = self.cost(self.u, self.d, self.m, self.beta)

                #print(".....................Here.............")
                #print(alpha)
```

```
                #print(cost_new,cost_old)
                #print(cost_new,cost_old - alpha * c_armijo * grad_norm2)
                # check if Armijo conditions are satisfied
                if cost_new < cost_old - alpha * c_armijo * grad_norm2:
                    #print("..............>>>Finally here.....")
                    cost_old = cost_new
                    backtrack_converged = True
                    break
                else:
                    alpha *= 0.5
                    self.m.assign(m_prev)  # reset m

            if backtrack_converged == False:
                print( "Backtracking failed. A sufficient descent direction was not found" )
                converged = False
                break


            # printting the values
            sp = ""
            if (count % print_any)== 0 :
                print( "%3d %1s %8.5e %1s %8.5e %1s %8.5e %1s %8.5e %1s %8.5e %1s %3d" % \
                    (count, sp, cost_new, sp, misfit_new, sp, reg_new, sp, \
                    gradnorm, sp, alpha, sp, it_backtrack) )

            if (count % plot_any)== 0 :
                nb.multi1_plot([self.m,self.u,self.p], ["m","u","p"], same_colorbar=False)
                plt.show()

            # check for convergence
            if gradnorm < tol*gradnorm0 and count > 0:
                converged = True
                print ("Steepest descent converged in ",count,"  iterations")

            alpha *= 1.5    # Try to increase the step size for the next iteration
            count += 1

        if not converged:
            print ( "Steepest descent did not converge in ", self.max_iter, " iterations")

        return count, cost_new, misfit_new, reg_new

    # plot the solution
    def plot_solution(self, title="Not given"):
        plt.figure(figsize = (15,10))

        plt.subplot(221)
        pl = dl.plot(self.mtrue, title = "mtrue")
        plt.colorbar(pl)

        plt.subplot(222)
        pl = dl.plot(self.m, title = "m")
        plt.colorbar(pl)

        plt.subplot(223)
        pl = dl.plot(self.u, title = "u")
        plt.colorbar(pl)

        plt.subplot(224)
        pl = dl.plot(self.p, title = "p")
        plt.colorbar(pl)
        plt.savefig(title+".png")
        plt.show()


# main function

if __name__ == "__main__":
    # input parameters
    mesh_file = "circle.xml"
    beta = 1e-5
    noise_level = 0.01
    nsources = 3
    k0 = 5.0
    iters = 1200
```

```
    alpha = 1
    regul = "h1"
    start_point = 0.0 #4.05 # "low" or float # meaning at m = 0; else takes 8; or give float va
    graph_name = "Inversion results for case beta =1e-5, m(x)=0"
    precon = False

    # initialize the class
    helm = Helmholtz(start_point, mesh_file, beta, noise_level, nsources, k0, iters, alpha, reg
    # solve the problem
    helm.solution_flow()
```

In [ ]:

```
""" Code for Q-2.3
Computational and Variational Methods for Inverse Problems (Spring 2024)
Assignment-04 -- Question-2.3

--->  Steepest Descent method for solving Helmholtz inverse problem with domain at unit Circle.

Author: Shreshth Saini (saini.2@utexas.edu)
Date: 29th April 2024
"""

# Question 2.2
#-----------------------------------------------------------------------------------------------

# Compute libraries
import dolfin as dl
import ufl
from hippylib import nb
import numpy as np
import mshr

# general libraries
import matplotlib.pyplot as plt
%matplotlib inline
import logging

# suppress warnings
import warnings
warnings.filterwarnings("ignore")
#-----------------------------------------------------------------------------------------------

# initialize the logger
logging.getLogger("FFC").setLevel(logging.ERROR)
logging.getLogger("UFL").setLevel(logging.ERROR)
dl.set_log_active(False)


np.random.seed(seed = 22)

class Helmholtz():
    def __init__(self,  starting_location, mesh_file:str, beta:float, noise_level:float, nsourc
        self.mesh_file = mesh_file
        self.beta = beta
        self.noise_level = noise_level
        self.regularization = regularization
        self.starting_point = starting_location
        self.print_name = print_name
        self.plot = plot_level
        self.k0 = k0
        self.k0squared = dl.Constant(k0*k0)
        self.nsources = nsources
        self.max_iter = max_iter
        self.alpha = alpha
        self.precon_solver = precon_solver

    # defining and solving - flow
    def solution_flow(self):
        # obtaining mesh
        self.get_mesh(self.mesh_file)

        # m and u
        self.Vm = dl.FunctionSpace(self.mesh, "Lagrange", 1)
        self.Vu = dl.FunctionSpace(self.mesh, "Lagrange", 2)
```

```
        # The true parameter
        self.mtrue_str = '.2*((x[0]-.1)*(x[0]-.1) + 2.*(x[1]+.2)*(x[1]+.2) < 0.25)'
        self.mtrue = dl.interpolate(dl.Expression(self.mtrue_str, degree=5), self.Vm)

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

        # source initialization
        self.source_function(self.nsources, self.k0)

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

        # define trail and test functions
        self.u_trial, self.m_trial, self.p_trial = dl.TrialFunction(self.Vu), dl.TrialFunction(
        self.u_test, self.m_test, self.p_test = dl.TestFunction(self.Vu), dl.TestFunction(self.


        # initialize input functions
        u0 = dl.Constant(0.0) # difussion coefficient
        # already initialized the u_inc in source function
        # k0^2 is also defined


        # boundary conditions - define here
        self.bc_state = self.boundary_condition(u0)
        self.bc_adj = self.boundary_condition(dl.Constant(0.0))

        # weak form of the pde
        self.utrue = dl.Function(self.Vu)

        # defining the state equation with the true parameter
        for i in self.u_inc:
            phi_true = self.pde_varf(self.utrue, self.mtrue, self.u_test,i)
            #dl.solve(A_true, self.utrue.vector(), b_true)
            dl.solve(phi_true==0, self.utrue, self.bc_state, solver_parameters={"newton_solver"


        # taken from Jupyter notebook provided
        Avarf = dl.inner(self.u_trial, self.u_test)*dl.dx
        self.A_true = dl.assemble(Avarf)

        # observed data
        self.d = dl.Function(self.Vu)
        self.d.assign(self.utrue)

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

        # apply noise to bc_adj
        self.bc_adj.apply(noise)

        # add noise to the state variable
        self.d.vector().axpy(1., noise)

        # plot
        if self.plot == "full":
            nb.multi1_plot([self.utrue, self.d], ["State solution with mtrue", "Synthetic obser
            plt.show()


        # weak form for settnig up the state equation # double check this line 79
        self.phi_state = [self.pde_varf(self.u, self.m, self.p_test,i) for i in self.u_inc]

        """
```

```
        self.phi_state = ufl.inner(ufl.grad(self.u), ufl.grad(self.p_test))*ufl.dx \
                         -self.k0squared*(dl.Constant(1.) - ufl.tanh(self.m) )*self.u*self.p
                         - self.k0squared*ufl.tanh(self.m)*self.u_inc*self.p_test*ufl.dx
        """

        # now the weak form for the adjoint equation
        self.phi_adj = self.pde_adj(self.u, self.m, self.u_test)

        """
        self.phi_adj = (self.u - self.d)*self.u_test*ufl.dx \
                       + ufl.inner(ufl.grad(self.p), ufl.grad(self.u_test))*ufl.dx \
                       - self.k0squared*(dl.Constant(1.) - ufl.tanh(self.m))*self.p*self.u_tes
        """

        # Now for the gradient (weak form)
        delta = 0.01
        # second term in the gradient
        self.CTvarf = [-self.k0squared*(dl.Constant(1.) - ufl.tanh(self.m)**2)*(-self.u + u_inc

        # the regulariation term (based on type of regularization)
        if self.regularization == "h1":
            self.gradRvarf = dl.Constant(self.beta)*(dl.inner(dl.grad(self.m), dl.grad(self.m_t

        elif self.regularization == "tv":
            self.gradRvarf = dl.Constant(self.beta)*dl.inner(dl.grad(self.m), dl.grad(self.m_te
                             /dl.sqrt(dl.inner(dl.grad(self.m), dl.grad(self.m)) + dl.Constant(d

        else:
            print(f"Regularization type not implemented.")
            return None

        # mass matrix assembly in parameter space
        Mvarf = dl.inner(self.m_trial, self.m_test)*dl.dx
        self.M = dl.assemble(Mvarf)


        #  matrix in parameter space
        Kvarf    = ufl.inner(self.m_trial, self.m_test) * ufl.dx + ufl.inner(ufl.grad(self.m_tr
        self.K = dl.assemble(Kvarf)

        ######## check finite diff grad
        #self.check_grad_error()

        # initial guess for the parameter
        if self.starting_point == 'low':
            print("Starting with m=0")
            m0 = dl.interpolate(dl.Constant(0.00), self.Vm)
            self.m.assign(m0)
        # if starting point is float then assign that value
        elif isinstance(self.starting_point, float)  or isinstance(self.starting_point, int):
            print("Starting m with given float: {}".format(self.starting_point))
            m0 = dl.interpolate(dl.Constant(self.starting_point), self.Vm)
            self.m.assign(m0)
        else:
            print("Starting with m = 8")
            m0 = dl.interpolate(dl.Constant(8.0), self.Vm)
            self.m.assign(m0)


        # solve the state equation
        for i in range(len(self.u_inc)):
            dl.solve(self.phi_state[i] == 0, self.u, self.bc_state, solver_parameters={"newton_


        # evaluate cost
        cost_old,_,_ = self.cost(self.u, self.d, self.m, self.beta)

        if self.plot == "full":
            plt.figure(figsize=(15,5))
            plt.subplot(121)
            dl.plot(self.m,title="m0", vmin=self.mtrue.vector().min(), vmax=self.mtrue.vector()
            plt.subplot(122)
            dl.plot(self.u, title="u(m0)")
            plt.show()
```

```
        # solving using the steepest descent method
        count, cost_new, misfit_new, reg_new = self.steepest_descent(cost_old, alpha = self.alp

        # plot the solution
        self.plot_solution(self.print_name)

        # return the solution
        Mstate = dl.assemble(self.u_trial*self.u_test*dl.dx)
        noise_norm2 = noise.inner(Mstate*noise)
        return self.Vm.dim(), count, noise_norm2, cost_new, misfit_new, reg_new



    # generate mesh or load from file
    def get_mesh(self, mesh_file=None):
        if mesh_file is None:
            self.mesh = self.generate_mesh()
        else:
            try:
                self.mesh = dl.Mesh(mesh_file)
            except:
                print(f"Mesh file not found. Trying in current directory... ")
                try:
                    self.mesh = dl.Mesh(mesh_file.split("/")[-1])
                except:
                    print(f"Mesh file not found. Generating mesh...")
                    self.mesh = self.generate_mesh()

    def generate_mesh(self, type="Circle"):
        # create mesh
        if type == "Circle":
            domain = mshr.Circle(dl.Point(0,0), 1)
            mesh = mshr.generate_mesh(domain, 50)
        else:
            print(f"Mesh type not implemented.")
            return None
        return mesh

    def source_function(self, nsources=3, k0=5.):
        # initialize input functions
        theta_view = np.linspace(-np.pi, np.pi, nsources, endpoint=False)
        radius = 1.05
        s0 = radius*np.cos(theta_view)
        s1 = radius*np.sin(theta_view)

        u_is = []
        # expression is predefined; does not need to be changed unless the source is changed
        for i in np.arange(nsources):
            distance_source = dl.Expression("std::sqrt((x[0]-s0)*(x[0]-s0) + (x[1]-s1)*(x[1]-s1
                                            s0 = s0[i], s1 = s1[i], degree=1)
            u_i_exp = dl.Expression("-std::cos(k0*distance_source)/(4.*DOLFIN_PI*distance_sourc
            u_is.append(dl.interpolate(u_i_exp, self.Vu) )

        if self.plot == "full":
            plt.figure(figsize=(15,3.3))
            nb.plot(u_is[0], subplot_loc=131, mytitle="First source")
            nb.plot(u_is[1], subplot_loc=132, mytitle="Second source")
            nb.plot(u_is[2], subplot_loc=133, mytitle="Third source")
            plt.show()


        # final source encoding; combining the sources
        encoding   = np.random.normal(size=nsources)
        # u_inc used in equation
        u_inc = dl.Function(self.Vu)

        for i, u_i in enumerate(u_is):
            u_inc.vector().axpy(encoding[i], u_i.vector())

        if self.plot == "full":
            plt.figure(figsize=(7.5,5))
            nb.plot(u_inc, mytitle="Super source")
            plt.show()
```

```
        self.u_inc = u_is

    def pde_varf(self, u,m,p,u_inc):
        return ufl.inner(ufl.grad(u), ufl.grad(p))*ufl.dx \
                -self.k0squared*(dl.Constant(1.) - ufl.tanh(m) )*u*p*ufl.dx \
                - self.k0squared*ufl.tanh(m)*u_inc*p*ufl.dx

    # based on this we make the adjoint equation
    def pde_adj(self, u,m,p):
            return (u - self.d)*p*ufl.dx \
                    + ufl.inner(ufl.grad(p), ufl.grad(self.p))*ufl.dx \
                    - self.k0squared*(dl.Constant(1.) - ufl.tanh(m))*p*self.p*ufl.dx

    # boundary condition
    def boundary_condition(self, u):
        def boundary(x, on_boundary):
            return on_boundary
        # defining the boundary condition
        bc = dl.DirichletBC(self.Vu, u, boundary)
        return bc

    # cost function
    def cost(self, u, d, m, beta):
        reg = 0.5 * beta * dl.assemble( (ufl.inner(dl.grad(m), dl.grad(m)))*ufl.dx )
        #reg = 0.5* beta * dl.assemble(dl.sqrt(dl.inner(dl.grad(m), dl.grad(m)))*dl.dx)
        misfit = 0.5 * dl.assemble( (u-d)**2*ufl.dx)
        return [reg + misfit, misfit, reg]

    # check grad error
    def check_grad_error(self):
        m0 = dl.interpolate(dl.Constant(0.0), self.Vm )
        n_eps = 32
        eps = np.power(2., -np.arange(n_eps))
        err_grad = np.zeros(n_eps)

        self.m.assign(m0)

        #Solve the fwd problem and evaluate the cost functional
        dl.solve(self.phi_state == 0, self.u, self.bc_state, solver_parameters={"newton_solver"

        c0, _, _ = self.cost(self.u, self.d, self.m, self.beta)


        # Solve the adjoint problem and evaluate the gradient
        dl.solve(self.phi_adj == 0, self.p, self.bc_adj, solver_parameters={"newton_solver": {"

        # evaluate the  gradient
        grad0 = dl.assemble(self.CTvarf+self.gradRvarf)

        # Define an arbitrary direction m_hat to perform the check
        mtilde = dl.Function(self.Vm).vector()
        mtilde.set_local(np.random.randn(self.Vm.dim()))
        mtilde.apply("")
        mtilde_grad0 = grad0.inner(mtilde)

        for i in range(n_eps):
            self.m.assign(m0)
            self.m.vector().axpy(eps[i], mtilde)

            dl.solve(self.phi_state == 0, self.u, self.bc_state, solver_parameters={"newton_sol

            cplus, _, _ = self.cost(self.u, self.d, self.m, self.beta)

            err_grad[i] = abs( (cplus - c0)/eps[i] - mtilde_grad0 )

        plt.figure()
        plt.loglog(eps, err_grad, "-ob", label="Error Grad")
        plt.loglog(eps, (.5*err_grad[0]/eps[0])*eps, "-.k", label="First Order")
        plt.title("Finite difference check of the first variation (gradient)")
        plt.xlabel("eps")
        plt.ylabel("Error grad")
        plt.legend(loc = "upper left")
        plt.show()
```

```
# solve the system
def steepest_descent(self, cost_old, tol=1e-4, backtrack_maxiter=20, c_armijo=1e-5, alpha =
    """
    tol         # Relative tolerance on the gradient norm
    maxiter     # Maximum number of iterations (we need a lot!)
    print_any   # We will only print progress on screen every few iterations
    plot_any    # We will plot the current solution every few iterations
    c_armijo    # The Armijo constant to ensure sufficient descent
    alpha       # The initial step-size
    """
    # initialize iter counters
    count = 0
    converged = False

    # initializations
    g = dl.Vector()
    # #Select whether to use the L^2 inner product (mass matrix M)
    # or to precondition steepest descent using the H^1 inner product (stiffness matrix K)
    if self.precon_solver:
        g = dl.Vector()
        self.K.init_vector(g,0)
        P = self.K
    else:
        self.M.init_vector(g,0)
        P = self.M

    m_prev = dl.Function(self.Vm)

    print( "Nit  cost          misfit        reg         ||grad||       alpha  N backtrack"
    print( "-------------------------------------------------------------------------------
    while count <  self.max_iter and not converged:
        print("Count", count)

        # solve the adoint problem
        dl.solve(self.phi_adj == 0, self.p, self.bc_adj, solver_parameters={"newton_solver"

        # evaluate the  gradient
        for i in range(len(self.u_inc)):
            MG = dl.assemble(self.CTvarf[i] + self.gradRvarf)
            dl.solve(P, g, MG)

        # calculate the norm of the gradient
        grad_norm2 = g.inner(MG)
        gradnorm = np.sqrt(grad_norm2)

        if count == 0:
            gradnorm0 = gradnorm

        # linesearch
        it_backtrack = 0
        m_prev.assign(self.m)
        backtrack_converged = False
        for it_backtrack in range(backtrack_maxiter):

            self.m.vector().axpy(-alpha, g)

            # print("solving forward prob")
            for i in range(len(self.u_inc)):
                dl.solve(self.phi_state[i] == 0, self.u, self.bc_state, solver_parameters={

            # evaluate cost
            [cost_new,misfit_new,reg_new] = self.cost(self.u, self.d, self.m, self.beta)

            #print(".....................Here.............")
            #print(alpha)
            #print(cost_new,cost_old)
            #print(cost_new,cost_old - alpha * c_armijo * grad_norm2)
            # check if Armijo conditions are satisfied
            if cost_new < cost_old - alpha * c_armijo * grad_norm2:
                #print(".............>>>Finally here.....")
                cost_old = cost_new
                backtrack_converged = True
```

```
                    break
                else:
                    alpha *= 0.5
                    self.m.assign(m_prev)  # reset m

            if backtrack_converged == False:
                print( "Backtracking failed. A sufficient descent direction was not found" )
                converged = False
                break


            # printting the values
            sp = ""
            if (count % print_any)== 0 :
                print( "%3d %1s %8.5e %1s %8.5e %1s %8.5e %1s %8.5e %1s %8.5e %1s %3d" % \
                    (count, sp, cost_new, sp, misfit_new, sp, reg_new, sp, \
                    gradnorm, sp, alpha, sp, it_backtrack) )

            if (count % plot_any)== 0 :
                nb.multi1_plot([self.m,self.u,self.p], ["m","u","p"], same_colorbar=False)
                plt.show()

            # check for convergence
            if gradnorm < tol*gradnorm0 and count > 0:
                converged = True
                print ("Steepest descent converged in ",count,"  iterations")

            alpha *= 1.5    # Try to increase the step size for the next iteration
            count += 1

        if not converged:
            print ( "Steepest descent did not converge in ", self.max_iter, " iterations")

        return count, cost_new, misfit_new, reg_new

    # plot the solution
    def plot_solution(self, title="Not given"):
        plt.figure(figsize = (15,10))

        plt.subplot(221)
        pl = dl.plot(self.mtrue, title = "mtrue")
        plt.colorbar(pl)

        plt.subplot(222)
        pl = dl.plot(self.m, title = "m")
        plt.colorbar(pl)

        plt.subplot(223)
        pl = dl.plot(self.u, title = "u")
        plt.colorbar(pl)

        plt.subplot(224)
        pl = dl.plot(self.p, title = "p")
        plt.colorbar(pl)
        plt.savefig(title+".png")
        plt.show()


# main function

if __name__ == "__main__":
    # input parameters
    mesh_file = "circle.xml"
    beta = 1e-5
    noise_level = 0.01
    nsources = 3
    k0 = 5#10.0
    iters = 1200
    alpha = 1
    regul = "tv"
    start_point = 0.0 # "low" or float # meaning at m = 0; else takes 8; or give float value
    graph_name = "Inversion results for case beta =1e-5, m(x)=0"
    precon = True

    # initialize the class
```

```
    helm = Helmholtz(start_point, mesh_file, beta, noise_level, nsources, k0, iters, alpha, reg
    # solve the problem
    helm.solution_flow()
```

In [ ]:

```
""" Code for Q-3
Computational and Variational Methods for Inverse Problems (Spring 2024)
Assignment-04 -- Question-3

--->  Inexact Newton-CG method for solving Helmholtz inverse problem with domain at unit Circle

Author: Shreshth Saini (saini.2@utexas.edu)
Date: 29th April 2024
"""

# Question 3
#-----------------------------------------------------------------------------------------------

# Compute libraries
import dolfin as dl
import ufl
from hippylib import nb
import numpy as np
import mshr
from hippylib import *

# general libraries
import matplotlib.pyplot as plt
%matplotlib inline
import logging

# suppress warnings
import warnings
warnings.filterwarnings("ignore")
#-----------------------------------------------------------------------------------------------

# initialize the logger
logging.getLogger("FFC").setLevel(logging.ERROR)
logging.getLogger("UFL").setLevel(logging.ERROR)
dl.set_log_active(False)


np.random.seed(seed = 42)

# 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)

        try:
            dl.solve (self.A, self.du, rhs)
        except RuntimeError as e:
            print("Error in solving the incremental forward problem:", e)
            return

        #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)


class Helmholtz_InCG():
    def __init__(self,  starting_location, mesh_file:str, beta:float, noise_level:float, nsourc
        self.mesh_file = mesh_file
        self.beta = beta
        self.noise_level = noise_level
        self.regularization = regularization
        self.starting_point = starting_location
        self.print_name = print_name
        self.plot = plot_level
        self.k0 = k0
        self.k0squared = dl.Constant(k0*k0)
        self.nsources = nsources
        self.max_iter = max_iter
        self.alpha = alpha
        self.precon_solver = precon_solver
        self.refine_level = refine_level
```

```
        self.morozov = morozov

    # defining and solving - flow
    def solution_flow(self):
        # obtaining mesh
        self.get_mesh(self.mesh_file)

        # m and u
        self.Vm = dl.FunctionSpace(self.mesh, "Lagrange", 1)
        self.Vu = dl.FunctionSpace(self.mesh, "Lagrange", 2)

        # The true parameter
        self.mtrue_str = '.2*((x[0]-.1)*(x[0]-.1) + 2.*(x[1]+.2)*(x[1]+.2) < 0.25)'
        self.mtrue = dl.interpolate(dl.Expression(self.mtrue_str, degree=5), self.Vm)

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

        # source initialization
        self.source_function(self.nsources, self.k0)

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

        # define trail and test functions
        self.u_trial, self.m_trial, self.p_trial = dl.TrialFunction(self.Vu), dl.TrialFunction(
        self.u_test, self.m_test, self.p_test = dl.TestFunction(self.Vu), dl.TestFunction(self.


        # initialize input functions
        u0 = dl.Constant(0.0) # difussion coefficient
        # already initialized the u_inc in source function
        # k0^2 is also defined


        # boundary conditions - define here
        self.bc_state = self.boundary_condition(u0)
        self.bc_adj =self.boundary_condition(dl.Constant(0.0))

        # weak form of the pde
        self.utrue = dl.Function(self.Vu)

        # defining the state equation with the true parameter
        phi_true = self.pde_varf(self.utrue, self.mtrue, self.u_test)
        #a_true = ufl.lhs(self.pde_varf(self.u_trial,self.mtrue,self.u_test))
        #L_true = ufl.rhs(self.pde_varf(self.u_trial,self.mtrue,self.u_test))

        # solving the forward problem;
        #A_true, b_true = dl.assemble_system(a_true, L_true)

        #dl.solve(A_true, self.utrue.vector(), b_true)
        dl.solve(phi_true==0, self.utrue, self.bc_state, solver_parameters={"newton_solver": {"


        # taken from Jupyter notebook provided
        Avarf = dl.inner(self.u_trial, self.u_test)*dl.dx
        self.A_true = dl.assemble(Avarf)

        # observed data
        self.d = dl.Function(self.Vu)
        self.d.assign(self.utrue)

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

```
        # apply noise to bc_adj
        self.bc_adj.apply(self.noise)

        # add noise to the state variable
        self.d.vector().axpy(1., self.noise)

        # plot
        if self.plot == "full":
            nb.multi1_plot([self.utrue, self.d], ["State solution with mtrue", "Synthetic obser
            plt.show()


        # weak form for settnig up the state equation # double check this line 79
        self.phi_state = self.pde_varf(self.u, self.m, self.p_test)
        #a_phi_state = ufl.lhs(self.pde_varf(self.u_trial,self.m,self.u_test))
        #L_phi_state = ufl.rhs(self.pde_varf(self.u_trial,self.m,self.u_test))

        #self.A_phi_state, self.b_phi_state = dl.assemble_system(a_phi_state, L_phi_state)

        """
        self.phi_state = ufl.inner(ufl.grad(self.u), ufl.grad(self.p_test))*ufl.dx \
                            -self.k0squared*(dl.Constant(1.) - ufl.tanh(self.m) )*self.u*self.p
                            - self.k0squared*ufl.tanh(self.m)*self.u_inc*self.p_test*ufl.dx
        """

        # now the weak form for the adjoint equation
        self.phi_adj = self.pde_adj(self.u, self.m, self.u_test)
        #a_phi_adj = ufl.lhs(self.pde_adj(self.p_trial,self.m,self.p_test))
        #L_phi_adj = ufl.rhs(self.pde_adj(self.p_trial,self.m,self.p_test))
        # solving the forward problem;
        #self.A_phi_adj, self.b_phi_adj = dl.assemble_system(a_phi_adj, L_phi_adj)

        """
        self.phi_adj = (self.u - self.d)*self.u_test*ufl.dx \
                        + ufl.inner(ufl.grad(self.p), ufl.grad(self.u_test))*ufl.dx \
                        - self.k0squared*(dl.Constant(1.) - ufl.tanh(self.m))*self.p*self.u_tes
        """

        # Now for the gradient (weak form)
        delta = 0.001
        # second term in the gradient
        self.CTvarf = -self.k0squared*(dl.Constant(1.) - ufl.tanh(self.m)**2)*(-self.u + self.u

        # the regularization term (based on type of regularization)
        if self.regularization == "h1":
            self.gradRvarf = dl.Constant(self.beta)*(dl.inner(dl.grad(self.m), dl.grad(self.m_t

        elif self.regularization == "tv":
            self.gradRvarf = dl.Constant(self.beta)*dl.inner(dl.grad(self.m), dl.grad(self.m_te
                            /dl.sqrt(dl.inner(dl.grad(self.m), dl.grad(self.m)) + dl.Constant(d

        else:
            print(f"Regularization type not implemented.")
            return None

        # mass matrix assembly in parameter space
        Mvarf = dl.inner(self.m_trial, self.m_test)*dl.dx
        self.M = dl.assemble(Mvarf)


        #  matrix in parameter space
        Kvarf    = ufl.inner(self.m_trial, self.m_test) * ufl.dx + ufl.inner(ufl.grad(self.m_tr
        self.K = dl.assemble(Kvarf)

        ######## check finite diff grad
        #self.check_grad_error()

        # initial guess for the parameter
        if self.starting_point == 'low':
            print("Starting with m=0")
            m0 = dl.interpolate(dl.Constant(0.00), self.Vm)
            self.m.assign(m0)
        # if starting point is float then assign that value
        elif isinstance(self.starting_point, float)  or isinstance(self.starting_point, int):
            print("Starting m with given float: {}".format(self.starting_point))
```

```
            m0 = dl.interpolate(dl.Constant(self.starting_point), self.Vm)
            self.m.assign(m0)
        else:
            print("Starting with m = 8")
            m0 = dl.interpolate(dl.Constant(8.0), self.Vm)
            self.m.assign(m0)


        # solve the state equation
        dl.solve(self.phi_state == 0, self.u, self.bc_state, solver_parameters={"newton_solver"
        #dl.solve(self.A_phi_state, self.u.vector(), self.b_phi_state)

        # evaluate cost
        cost_old,_,_ = self.cost(self.u, self.d, self.m, self.beta)

        if self.plot == "full":
            plt.figure(figsize=(15,5))
            plt.subplot(121)
            dl.plot(self.m,title="m0", vmin=self.mtrue.vector().min(), vmax=self.mtrue.vector()
            plt.subplot(122)
            dl.plot(self.u, title="u(m0)")
            plt.show()

        #### For Hessian:
        ########################
        # Hessian Varf:
        # part of adjoint term
        W_varf   = dl.inner(self.u_trial, self.u_test) * dl.dx
        # part of m-derivative
        R_varf   = dl.Constant(self.beta) * dl.inner(dl.grad(self.m_trial), dl.grad(self.m_test
        # no m derivative term
        self.C_varf = dl.inner(self.k0squared*(dl.Constant(1.0) - ufl.tanh(self.m)**2)*(self.u
        self.Wum_varf = dl.inner(self.k0squared*(dl.Constant(1.0) - ufl.tanh(self.m)**2)*self.m
        self.Wmm_varf = dl.inner(-self.k0squared*(self.u - self.u_inc)* dl.Constant(2.0)*(ufl.t

        # hess pde eqs:
        self.a_state, self.a_adj = self.hess_pdes()

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


        # solving using the steepest descent method
        Vm,count,total_cg_iter, noise_norm2, cost_new, misfit_new, reg_new = self.InCG(cost_old

        # plot the solution
        self.plot_solution(self.print_name)

        # return the solution
        return self.Vm.dim(),count,total_cg_iter, noise_norm2, cost_new, misfit_new, reg_new



    # generate mesh or load from file
    def get_mesh(self, mesh_file=None):
        if mesh_file is None:
            self.mesh = self.generate_mesh()
        else:
            try:
                self.mesh = dl.Mesh(mesh_file)
            except:
                print(f"Mesh file not found. Trying in current directory... ")
                try:
                    self.mesh = dl.Mesh(mesh_file.split("/")[-1])
                except:
                    print(f"Mesh file not found. Generating mesh...")
                    self.mesh = self.generate_mesh()

        # refining the mesh
        nrefinements = self.refine_level
        for i in range(nrefinements):
            self.mesh = dl.refine(self.mesh)
            print("Refinement level {0}; Number of mesh elements: {1}".format(i+1, self.mesh.nu
```

```
    def generate_mesh(self, type="Circle"):
        # create mesh
        if type == "Circle":
            domain = mshr.Circle(dl.Point(0,0), 1)
            mesh = mshr.generate_mesh(domain, 50)
        else:
            print(f"Mesh type not implemented.")
            return None
        return mesh

    def source_function(self, nsources=3, k0=5.):
        # initialize input functions
        theta_view = np.linspace(-np.pi, np.pi, nsources, endpoint=False)
        radius = 1.05
        s0 = radius*np.cos(theta_view)
        s1 = radius*np.sin(theta_view)

        u_is = []
        # expression is predefined; does not need to be changed unless the source is changed
        for i in np.arange(nsources):
            distance_source = dl.Expression("std::sqrt((x[0]-s0)*(x[0]-s0) + (x[1]-s1)*(x[1]-s1
                                            s0 = s0[i], s1 = s1[i], degree=1)
            u_i_exp = dl.Expression("-std::cos(k0*distance_source)/(4.*DOLFIN_PI*distance_sourc
            u_is.append(dl.interpolate(u_i_exp, self.Vu) )

        if self.plot == "full":
            plt.figure(figsize=(15,3.3))
            nb.plot(u_is[0], subplot_loc=131, mytitle="First source")
            nb.plot(u_is[1], subplot_loc=132, mytitle="Second source")
            nb.plot(u_is[2], subplot_loc=133, mytitle="Third source")
            plt.show()


        # final source encoding; combining the sources
        encoding   = np.random.normal(size=nsources)
        # u_inc used in equation
        u_inc = dl.Function(self.Vu)

        for i, u_i in enumerate(u_is):
            u_inc.vector().axpy(encoding[i], u_i.vector())

        if self.plot == "full":
            plt.figure(figsize=(7.5,5))
            nb.plot(u_inc, mytitle="Super source")
            plt.show()

        self.u_inc = u_inc

    def pde_varf(self, u,m,p):
        return ufl.inner(ufl.grad(u), ufl.grad(p))*ufl.dx \
                -self.k0squared*(dl.Constant(1.) - ufl.tanh(m) )*u*p*ufl.dx \
                - self.k0squared*ufl.tanh(m)*self.u_inc*p*ufl.dx

    # based on this we make the adjoint equation
    def pde_adj(self, u,m,p):
            return (u - self.d)*p*ufl.dx \
                    + ufl.inner(ufl.grad(p), ufl.grad(self.p))*ufl.dx \
                    - self.k0squared*(dl.Constant(1.) - ufl.tanh(m))*p*self.p*ufl.dx

    def hess_pdes(self):
        state = ufl.inner(ufl.grad(self.u_trial), ufl.grad(self.p_test)) * ufl.dx + \
                    ufl.inner(-self.k0squared*(dl.Constant(1.0) - ufl.tanh(self.m))*self.u_tria
                        #self.k0squared*(self.u-self.u_inc)*(dl.Constant(1.0) - ufl.tanh(self.m

        adj = ufl.inner(ufl.grad(self.u_test), ufl.grad(self.p_trial)) * ufl.dx + \
                ufl.inner(-self.k0squared*(dl.Constant(1.0) - ufl.tanh(self.m))*self.u_test, se
                #self.k0squared*(dl.Constant(1.0) - ufl.tanh(self.m)**2)*self.u_test*self.m_tri

        return state, adj

    # boundary condition
    def boundary_condition(self, u):
```

```
        def boundary(x, on_boundary):
            return on_boundary
        # defining the boundary condition
        bc = dl.DirichletBC(self.Vu, u, boundary)
        return bc

    # cost function
    def cost(self, u, d, m, beta):
        reg = 0.5 * beta * dl.assemble( (ufl.inner(dl.grad(m), dl.grad(m)))*ufl.dx )
        #reg = 0.5* beta * dl.assemble(dl.sqrt(dl.inner(dl.grad(m), dl.grad(m)))*dl.dx)
        misfit = 0.5 * dl.assemble( (u-d)**2*ufl.dx)
        return [reg + misfit, misfit, reg]

    # check grad error
    def check_grad_error(self):
        m0 = dl.interpolate(dl.Constant(0.0), self.Vm )
        n_eps = 32
        eps = np.power(2., -np.arange(n_eps))
        err_grad = np.zeros(n_eps)

        self.m.assign(m0)

        #Solve the fwd problem and evaluate the cost functional
        dl.solve(self.phi_state == 0, self.u, self.bc_state, solver_parameters={"newton_solver"

        c0, _, _ = self.cost(self.u, self.d, self.m, self.beta)


        # Solve the adjoint problem and evaluate the gradient
        dl.solve(self.phi_adj == 0, self.p, self.bc_adj, solver_parameters={"newton_solver": {"

        # evaluate the  gradient
        grad0 = dl.assemble(self.CTvarf+self.gradRvarf)

        # Define an arbitrary direction m_hat to perform the check
        mtilde = dl.Function(self.Vm).vector()
        mtilde.set_local(np.random.randn(self.Vm.dim()))
        mtilde.apply("")
        mtilde_grad0 = grad0.inner(mtilde)

        for i in range(n_eps):
            self.m.assign(m0)
            self.m.vector().axpy(eps[i], mtilde)

            dl.solve(self.phi_state == 0, self.u, self.bc_state, solver_parameters={"newton_sol

            cplus, _, _ = self.cost(self.u, self.d, self.m, self.beta)

            err_grad[i] = abs( (cplus - c0)/eps[i] - mtilde_grad0 )

        plt.figure()
        plt.loglog(eps, err_grad, "-ob", label="Error Grad")
        plt.loglog(eps, (.5*err_grad[0]/eps[0])*eps, "-.k", label="First Order")
        plt.title("Finite difference check of the first variation (gradient)")
        plt.xlabel("eps")
        plt.ylabel("Error grad")
        plt.legend(loc = "upper left")
        plt.show()




    # solve the system using Inexact Newton CG:
    def InCG(self, cost_old, tol=1e-8, backtrack_maxiter=20, c_armijo=1e-4, alpha = 1, print_an
        """
        tol         # Relative tolerance on the gradient norm
        maxiter     # Maximum number of iterations (we need a lot!)
        print_any   # We will only print progress on screen every few iterations
        plot_any    # We will plot the current solution every few iterations
        c_armijo    # The Armijo constant to ensure sufficient descent
        alpha       # The initial step-size
        """
        # initialize iter counters
        count = 1
```

```
        total_cg_iter = 0
        converged = False

        # initializations
        g, m_delta = dl.Vector(), dl.Vector()
        # #Select whether to use the L^2 inner product (mass matrix M)
        # or to precondition steepest descent using the H^1 inner product (stiffness matrix K)
        if self.precon_solver:
            self.K.init_vector(g,0)
            P1 = self.K
        else:
            self.M.init_vector(g,0)
            P1 = self.M

        self.R.init_vector(m_delta,0)
        self.R.init_vector(g,0)

        m_prev = dl.Function(self.Vm)

        print( "Nit   CGit   cost          misfit        reg           sqrt(-G*D)    ||grad||
        print( "-----------------------------------------------------------------------------
        while count <  self.max_iter and not converged:
            #print("Count", count)

            # solve the adoint problem
            dl.solve(self.phi_adj == 0, self.p, self.bc_adj,  solver_parameters={"newton_solver

            # assemble the Hessian operator for adj
            adjoint_A,_ = dl.assemble_system(self.a_adj, dl.Constant(0.0)*self.p_test*dl.dx, se
            #adjoint_A,_ = dl.assemble_system(self.phi_adj, dl.Constant(0.)*self.p_test*dl.dx,
            # assemble the Hessian operator for state
            state_A,_ = dl.assemble_system(self.a_state, dl.Constant(0.0)*self.u_test*dl.dx, se
            #state_A,_ = dl.assemble_system(self.phi_state, dl.Constant(0.)*self.u_test*dl.dx,

            # evaluate the  gradient
            MG = dl.assemble(self.CTvarf + self.gradRvarf)
            dl.solve(P1, g, MG)

            # calculate the norm of the gradient
            grad_norm2 = g.inner(MG)
            gradnorm = np.sqrt(grad_norm2)

            # initialize the CG parameters
            if count ==1 :
                gradnorm_ini = gradnorm
            tolcg = min(0.5, np.sqrt(gradnorm/gradnorm_ini))

            # Hessian operators
            C = dl.assemble(self.C_varf)
            Wum = dl.assemble(self.Wum_varf)
            Wmm = dl.assemble(self.Wmm_varf)


            # define the Hessian apply operator (with preconditioner)
            Hess_Apply = HessianOperator(self.R, Wmm, C, state_A, adjoint_A, self.W, Wum, self.
            P = self.R + 0.1*self.beta * self.M
            Psolver = dl.PETScKrylovSolver("cg", amg_method()) ################Update this. amg
            Psolver.set_operator(P)

            solver = CGSolverSteihaug() ########### find this
            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
            ############## issueeee ***********
            solver.solve(m_delta, -MG)
            total_cg_iter += Hess_Apply.cgiter


            # linesearch
```

```
            it_backtrack = 0
            m_prev.assign(self.m)
            backtrack_converged = False
            for it_backtrack in range(backtrack_maxiter):

                self.m.vector().axpy(alpha, m_delta)

                # print("solving forward prob")
                #dl.solve(self.A_phi_state, self.u.vector(), self.b_phi_state)
                dl.solve(self.phi_state == 0, self.u, self.bc_state, solver_parameters={"newton

                # Hessian
                #state_A,_ = dl.assemble_system(self.a_state)

                # evaluate cost
                [cost_new,misfit_new,reg_new] = self.cost(self.u, self.d, self.m, self.beta)

                #print(".....................Here.............")
                #print(alpha)
                #print(cost_new,cost_old)
                #print(cost_new,cost_old - alpha * c_armijo * grad_norm2)
                # check if Armijo conditions are satisfied
                if cost_new < cost_old - alpha * c_armijo * MG.inner(m_delta):
                    #print(".............>>>Finally here.....")
                    cost_old = cost_new
                    backtrack_converged = True
                    break
                else:
                    alpha *= 0.5
                    self.m.assign(m_prev)  # reset m

            """
            if backtrack_converged == False:
                print( "Backtracking failed. A sufficient descent direction was not found" )
                converged = False
                break
            """
            graddir = np.sqrt(-MG.inner(m_delta))

            # printting the values
            sp = ""

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


            #nb.multi1_plot([self.m,self.u,self.p], ["m","u","p"], same_colorbar=False)
            #plt.show()

            # check for convergence
            if gradnorm < tol and count > 0:
                converged = True
                print( "Newton's method converged in ",count,"  iterations" )
                print( "Total number of CG iterations: ", total_cg_iter )

            alpha *= 1.5
            count += 1

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


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

        if not self.morozov:
            Hmisfit = HessianOperator(None, Wmm, C, state_A, adjoint_A, self.W, Wum, self.bc_ad
            k = 50
            p = 20

            Omega = MultiVector(m.vector(), k+p)
            parRandom.normal(1., Omega)
            lmbda, evecs = doublePassG(Hmisfit, P, Psolver, Omega, k)
            # check length and values of lmbda and evecs
```

```
            print(  "Eigenvalues of the Hessian operator")
            print( lmbda)
            print(evecs)

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


        return self.Vm.dim(),count,total_cg_iter, noise_norm2, cost_new, misfit_new, reg_new

    # plot the solution
    def plot_solution(self, title="Not given"):
        plt.figure(figsize = (15,10))

        plt.subplot(221)
        pl = dl.plot(self.mtrue, title = "mtrue")
        plt.colorbar(pl)

        plt.subplot(222)
        pl = dl.plot(self.m, title = "m")
        plt.colorbar(pl)

        plt.subplot(223)
        pl = dl.plot(self.u, title = "u")
        plt.colorbar(pl)

        plt.subplot(224)
        pl = dl.plot(self.p, title = "p")
        plt.colorbar(pl)
        plt.savefig(title+".png")
        plt.show()
```