Spring 2024: Compututational and Variational Methods for Inverse Problems

Assignment-03: Calculus of variations and image denoising with total variation regularization

Shreshth Saini (SS223464)
saini.2@utexas.edu
Due April 16, 2024

Problem-1: Image Denoising

Given:

Given a noisy image $d(x, y)$ defined on squared domain Ω, and denoised version of image $u(x, y)$, that is close to the noisy image in L2 sense. We need to minimize the following data misfit while removing the noise (note that noise has highly osciallatory nature):

$$\mathcal{F}_{LS}(u) := \frac{1}{2}\int_\Omega (u - d)^2 dx \tag{1.1}$$

In order to incorporate the denoising with above minimization, we add a regularization term to the above functional. The regularization term penalizes steeper gradients:

$$\mathcal{R}_{TN}(u) := \frac{\beta}{2}\int_\Omega \nabla u.\nabla u dx \tag{1.2}$$

where $\beta > 0$ is a regularization parameter that controls how strongly we impose the regularization. Generally, TN regularization introduces bluriness. A better regularization is Total Variation (TV) regularization, and to avoid non-differentiability at zero we add small parameter $\delta > 0$, which is defined as:

$$\mathcal{R}^\delta_{TV}(u) := \beta\int_\Omega (\nabla u.\nabla u + \delta)^{1/2} dx \tag{1.3}$$

We need to study the two denoising functionals $\mathcal{F}_{TN}(u)$ and $\mathcal{F}^\delta_{TV}(u)$ throught minimizing the following problems:

$$\min_{u\in\mathcal{U}} \mathcal{F}_{TN}(u) := \mathcal{F}_{LS}(u) + \mathcal{R}_{TN}(u) \tag{1.4}$$

$$\min_{u\in\mathcal{U}} \mathcal{F}^\delta_{TV}(u) := \mathcal{F}_{LS}(u) + \mathcal{R}^\delta_{TV}(u) \tag{1.5}$$

where $\mathcal{U}$ is the set of admissible functions, and boundary conditions are $\nabla u.\text{\boldn} = 0$ on four sides of the square domain. Image density does not change normal to the boundary of the image.

(a) First Order necessary condition for optimality:

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

For TN regularization:

variation of the functional $\mathcal{F}_{LS}(u)$ is given by:

$$\delta_u\mathcal{F}_{LS}(u)[\hat{u}] = \int_\Omega (u - d)\hat{u} dx \tag{1.6}$$

variation of the functional $\mathcal{R}_{TN}(u)$ is given by:

$$\delta_u\mathcal{R}_{TN}(u)[\hat{u}] = \int_\Omega \beta\nabla u.\nabla\hat{u} dx \tag{1.7}$$

Using calculus of variations, we can compute the variation of the functional $\mathcal{R}_{TV}(u)$ as $u - - > (u + \delta\hat{u})$:

$$\delta_u \mathcal{R}^{\delta}_{TV}(u)[\hat{u}] = \frac{d}{d\delta}[\int_{\Omega} \beta(\nabla(u+\delta\hat{u}).\nabla(u+\delta\hat{u})+\delta)^{1/2}dx]_{\delta=0} \tag{1.8}$$

$$= \int_{\Omega} \beta\frac{\nabla u.\nabla\hat{u}}{(\nabla u.\nabla u+\delta)^{1/2}}dx \tag{1.9}$$

using 1.6, 1.7 and 1.9, we can write the weak form of the functional $\mathcal{F}_{TN}(u)$ as:

$$\delta_u \mathcal{F}_{TN}(u)[\hat{u}] = \int_{\Omega}(u-d)\hat{u}dx + \int_{\Omega}\beta\nabla u.\nabla\hat{u}dx = 0\forall\hat{u}\in H^1(\Omega) \tag{1.10}$$

We can use by parts and gauss divergence theorem to write the strong form of the functional $\mathcal{F}_{TN}(u)$ as:

$$\delta_u \mathcal{F}_{TN}(u)[\hat{u}] = \int_{\Omega}(u-d)\hat{u}dx - \int_{\Omega}\beta\Delta u.\nabla\hat{u}dx + \int_{\partial\Omega}\beta\hat{u}\nabla u.\text{\boldn}ds = 0\forall\hat{u}\in H^1(\Omega) \tag{1.11}$$

here $\Delta = \nabla^2$, and the strong form :

$$\beta\Delta u = (u-d), in\Omega \tag{1.12}$$

$$\nabla u.\text{\boldn} = 0, on\partial\Omega \tag{1.13}$$

For TV regularization:

using 1.6 and 1.9, we can write the weak form of the functional $\mathcal{F}_{TV}(u)$ as:

$$\delta_u \mathcal{F}^{\delta}_{TV}(u)[\hat{u}] = \int_{\Omega}(u-d)\hat{u}dx + \int_{\Omega}\beta\frac{\nabla u.\nabla\hat{u}}{(\nabla u.\nabla u+\delta)^{1/2}}dx = 0\forall\hat{u}\in H^1(\Omega) \tag{1.14}$$

again, by using integration by parts and gauss divergence theorem, we can write the strong form of the functional $\mathcal{F}_{TV}(u)$ as:

$$\delta_u \mathcal{F}^{\delta}_{TV}(u)[\hat{u}] = \int_{\Omega}(u-d)\hat{u}dx - \int_{\Omega}\beta\nabla.(\frac{\nabla u}{(\nabla u.\nabla u+\delta)^{1/2}})\nabla\hat{u}dx + \int_{\partial\Omega}\beta\hat{u}\frac{\nabla u.\text{\boldn}}{\nabla u.\nabla u+\delta}ds = 0\forall\hat{u}\in$$

thus the strong form is:

$$\beta\nabla.(\frac{\nabla u}{(\nabla u.\nabla u+\delta)^{1/2}}) = (u-d), in\Omega \tag{1.16}$$

$$\nabla u.\text{\boldn} = 0, on\partial\Omega \tag{1.17}$$

(b) For $\nabla u = 0$, $\mathcal{R}_{TV}$ is not differentiable but $\mathcal{R}^{\delta}_{TV}$ is:

From 1.3:

$$\delta_u \mathcal{R}_{TV}(u,\hat{u}) = \beta\int_{\Omega}\frac{1}{2}(\nabla u.\nabla\hat{u})^(-1/2)2(\nabla u.\hat{u})dx$$

$$= \beta\int_{\Omega}\frac{\nabla u.\nabla\hat{u}}{(\nabla u.\nabla u)^{1/2}}dx \tag{1.18}$$

we can observe that for $\nabla u = 0$ denominator in 1.19 becomes zero, and thus the functional $\mathcal{R}_{TV}$ is not differentiable.

Whereas in case of δ

$$\delta_u \mathcal{R}^{\delta}_{TV}(u,\hat{u}) = \beta\int_{\Omega}\frac{\nabla u.\nabla\hat{u}}{(\nabla u.\nabla u+\delta)^{1/2}}dx \tag{1.19}$$

Here, the denominator is always positive, and thus the functional $\mathcal{R}^{\delta}_{TV}$ is differentiable.

(c) Infinite Dimensional Newton step:

Second variations of the different components by using a perturbation $\tilde{u} \in H^1(\Omega)$. Using calculus of variation on (1.6) and (1.7):

$$\delta^2 \mathcal{F}_{LS}(u, \hat{u}, \tilde{u}) = \int_\Omega \tilde{u}\hat{u} dx \tag{1.20}$$

$$\delta^2 \mathcal{R}_{TN}(u, \hat{u}, \tilde{u}) = \beta \int_\Omega \nabla\tilde{u} \cdot \nabla\hat{u}\, dx \tag{1.21}$$

Second variation of R^δ_{TV} using the calculus of variation:

$$\delta^2 \mathcal{R}^\delta_{TV}(u, \hat{u}, \tilde{u}) = \int_\Omega \frac{\beta}{(\nabla u \cdot \nabla u + \delta)^{1/2}} \left(\mathcal{I} - \frac{\nabla u \otimes \nabla u}{\nabla u \cdot \nabla u + \delta}\right) (\nabla\hat{u} \cdot \nabla\tilde{u})\, dx \tag{1.22}$$

The Newton step $\tilde{u}$ is given by

$$\delta^2 \mathcal{F}_u(u, \hat{u}, \tilde{u}) = -\delta_u \mathcal{F}(u, \hat{u}, \tilde{u}), \quad \forall \tilde{u} \in H^1(\Omega) \tag{1.23}$$

Now, let's apply it for the different regularizations:

TN Regularization:

Plugging (1.20), (1.21), (1.11) and (1.5) in (1.23) gives the weak form

$$\int_\Omega \tilde{u}\hat{u}\, dx + \beta \int_\Omega \nabla\hat{u} \cdot \nabla\tilde{u}\, dx = -\int_\Omega (u-d)\hat{u}\, dx - \beta \int_\Omega \nabla u \cdot \nabla\hat{u}\, dx = 0, \quad \forall \hat{u}, \tilde{u} \in H^1(\Omega) \quad (1.24)$$

Now, again using the integration by parts,

$$\int_\Omega \tilde{u}\hat{u}\, dx - \beta \int_\Omega \Delta\hat{u}\tilde{u}\, dx + \beta \int_{\partial\Omega} \hat{u}\nabla\tilde{u} \cdot n\, ds = -\int_\Omega (u-d)\hat{u}\, dx + \beta \int_\Omega \Delta u\hat{u}\, dx - \beta \int_{\partial\Omega} \hat{u}\nabla u \cdot n\, ds$$

strong form:

$$\tilde{u} - \beta\Delta\tilde{u} = -(u-d) + \beta\Delta u \quad \text{in} \quad \Omega \tag{1.26}$$

$$\beta\nabla\tilde{u} \cdot n = -\beta\nabla u \cdot n \quad \text{on} \quad \partial\Omega \tag{1.27}$$

TV Regularization:

The weak form

$$\int_\Omega \hat{u}\tilde{u}\, dx + \int_\Omega \frac{\beta}{(\nabla u \cdot \nabla u + \delta)^{1/2}} \left(I - \frac{\nabla u \otimes \nabla u}{\nabla u \cdot \nabla u + \delta}\right) \nabla\tilde{u}\nabla\hat{u}\, dx = -\int_\Omega (u-d)\hat{u}\, dx - \beta \int_\Omega \frac{\nabla}{(\nabla u \cdot}$$

Integration by parts leads to gives

$$\int_\Omega \hat{u}\tilde{u}\, dx - \int_\Omega \nabla \cdot \left(\frac{\beta}{(\nabla u \cdot \nabla u + \delta)^{1/2}} \left(I - \frac{\nabla u \otimes \nabla u}{\nabla u \cdot \nabla u + \delta}\right) \nabla\tilde{u}\right) \hat{u}\, dx$$

$$+ \int_{\partial\Omega} \hat{u} \frac{\beta}{(\nabla u \cdot \nabla u + \delta)^{1/2}} \left(I - \frac{\nabla u \otimes \nabla u}{\nabla u \cdot \nabla u + \delta}\right) \nabla\tilde{u} \cdot \backslash\text{bold} nds$$

$$-\int_\Omega \frac{\beta\Delta u}{(\nabla u \cdot \nabla u + \delta)^{1/2}} \hat{u} + \int_{\partial\Omega} \hat{u} \frac{\beta\nabla u \cdot n}{(\nabla u \cdot \nabla u + \delta)^{-1/2}}\, ds = -\int_\Omega (u-d)\hat{u}\, dx \tag{1.29}$$

$\tilde{u}$ leads to the strong form :

$$\tilde{u} - \nabla \cdot \left(\frac{\beta}{(\nabla u \cdot \nabla u + \delta)^{1/2}} \left(I - \frac{\nabla u \otimes \nabla u}{\nabla u \cdot \nabla u + \delta}\right) \nabla\tilde{u}\right) = -(u-d) + \nabla \cdot \left(\frac{\beta\nabla u}{(\nabla u \cdot \nabla u + \delta)^{1/2}}\right) \quad \text{i}$$

$$\frac{\beta\nabla\tilde{u} \cdot n}{(\nabla u \cdot \nabla u + \delta)^{1/2}} \cdot (\mathcal{I} - \frac{\nabla u \otimes \nabla u}{\nabla u . \nabla u + \delta}) = -\frac{\beta\nabla u \cdot n}{(\nabla u \cdot \nabla u + \delta)^{1/2}} \quad \text{on} \quad \partial\Omega \tag{1.31}$$

Here, the anisotropic tensor $A(u)$ that plays the role of the diffusion coefficient which is defined as

$$A(u) = \frac{\beta}{(\nabla u \cdot \nabla u + \delta)^{1/2}} \left(I - \frac{\nabla u \otimes \nabla u}{\nabla u \cdot \nabla u + \delta} \right) \tag{1.32}$$

(d) Eige values and Vectors of A(u):

The tensor $A(u) \in \mathbb{R}^{2\times 2}$ has two eigenvalues λ_1 and λ_2 and corresponding eigenvector v_1 and v_2.

$v_1 = \mu \nabla u = \left(\mu \frac{\partial u}{\partial x}, \mu \frac{\partial u}{\partial y} \right)$.

using (1.32),

$$\begin{aligned} Av_1 &= \frac{\beta}{(\nabla u \cdot \nabla u + \delta)^{1/2}} \left(I - \frac{\nabla u \otimes \nabla u}{\nabla u \cdot \nabla u + \delta} \right) \mu \nabla u \\ &= \frac{\beta}{(\nabla u \cdot \nabla u + \delta)^{1/2}} \left(1 - \frac{\nabla u \cdot \nabla u}{\nabla u \cdot \nabla u + \delta} \right) \mu \nabla u \end{aligned} \tag{1.33}$$

First eigenvalue is

$$\lambda_1 = \frac{\beta}{(\nabla u \cdot \nabla u + \delta)^{1/2}} \left(1 - \frac{\nabla u \cdot \nabla u}{\nabla u \cdot \nabla u + \delta} \right) \tag{1.34}$$

Let the second eigenvectors to be orthogonal to the previous eigenvectors as A is symmetric. So, let $v_2 = \gamma (\nabla u)^\perp$ and again applying it to (1.32),

$$Av_2 = \frac{\beta}{(\nabla u \cdot \nabla u + \delta)^{1/2}} \left(I - \frac{\nabla u \otimes \nabla u}{\nabla u \cdot \nabla u + \delta} \right) \gamma (\nabla u)^\perp \tag{1.35}$$

$$= \frac{\beta}{(\nabla u \cdot \nabla u + \delta)^{1/2}} \left(\gamma (\nabla u)^\perp - \frac{\nabla u \cdot (\nabla u)^\perp}{\nabla u \cdot \nabla u + \delta} \gamma (\nabla u) \right) \tag{1.36}$$

$$= \frac{\beta}{(\nabla u \cdot \nabla u + \delta)^{1/2}} \gamma (\nabla u)^\perp \tag{1.37}$$

$$\lambda_2 = \frac{\beta}{(\nabla u \cdot \nabla u + \delta)^{1/2}} \tag{1.38}$$

Finally:

$$\lambda_1 = \frac{\beta}{(\nabla u \cdot \nabla u + \delta)^{1/2}} \left(1 - \frac{\nabla u \cdot \nabla u}{\nabla u \cdot \nabla u + \delta} \right), \quad v_1 = \nabla u \tag{1.39}$$

$$\lambda_2 = \frac{\beta}{(\nabla u \cdot \nabla u + \delta)^{1/2}}, \quad v_2 = (\nabla u)^\perp \tag{1.40}$$

For a very small δ, $\lambda_1 \approx 0$ whereas $\lambda_2 \approx \frac{\beta}{\nabla u \cdot \nabla u}$. Therefore, the diffusion is much smaller in the direction of the gradient when compared to its perpendicular direction. It can be observed from the strong form that the regularization is a diffusion equation where the diffusion is related to the laplacian Δ. This operator gives rise to a de-noising effect, which reduce the oscillations in the solution.

The difference between TV and TN regularization lies in the isotropic diffusion tensor. From the eigen-decomposition, we found that the eigenvalue parallel to the gradient ∇u is much smaller than the in the direction perpendicular to it $(\nabla u)^\perp$. This leads to the Newton steps that encourage diffusion and subsequently, smoothness. Moreover, it discourages diffusion and preserves sharp edges in the direction of the gradients. Thus, F_{TV}^δ is effective at preserving the sharpness of the images. The TN regularization yields only an isotropic diffusion tensor, which results diffusion in all the directions. Therefore, TN regularization also blurs the image while de-noising.

(e) Edge preserving for δ :

Using (1.15) and (1.22), the strong forms of Hessian $H_{TV}^\delta(u)$ and gradient $\mathcal{G}_{TV}^\delta(u)$:

$$\mathcal{H}^{\delta}_{TV}(u)\tilde{u} = -\nabla \cdot \left(\frac{\beta}{(\nabla u \cdot \nabla u + \delta)^{1/2}} \left(I - \frac{\nabla u \otimes \nabla u}{\nabla u \cdot \nabla u + \delta} \right) \tilde{u} \right) \tag{1.41}$$

$$\mathcal{G}^{\delta}_{TV}(u) = -\nabla \cdot \left(\frac{\beta \nabla u}{(\nabla u \cdot \nabla u + \delta)^{1/2}} \right) \tag{1.42}$$

For δ, we can observe that $\frac{\beta}{(\nabla u \cdot \nabla u + \delta)^{1/2}} \approx \frac{\beta}{\sqrt{\delta}}$ and $\left(I - \frac{\nabla u \otimes \nabla u}{\nabla u \cdot \nabla u + \delta} \right) \approx I$. Therefore, $H^{\delta}_{TV}(u)\tilde{u} \approx -\nabla \cdot \left(\frac{\beta}{\sqrt{\delta}} \tilde{u} \right)$ and $G^{\delta}_{TV}(u) \approx -\nabla \cdot \left(\frac{\alpha}{\sqrt{\delta}} \nabla u \right)$. For Tikhonov, $\mathcal{H}_{TN}(u)\tilde{u} \approx -\beta \Delta \tilde{u}$ and $G_{TN}(u) = -\beta \Delta u$.

So, both Hessians and Gradients are comparable. As a result, (R_{TN}) behaves like (R^{\delta}_{TV}).

For $\delta = 0, \lambda_1 = 0$ from (1.39) and therefore the isotropic diffusion tensor A is singular. This renders the diffusion problem from the Hessian of R^{δ}_{TV} ill-posed.

(f) Optimize-then-discretize (OTD) and Discretize-then-optimize (DTO) approaches :

OTD (Galerkin method):

Taking p to be the Newton step in infinite dimensions, we solve the system

$$\delta^2_u \mathcal{F}^{\delta}_{TV}(u, p, \hat{u}) = -\delta_u \mathcal{F}^{\delta}_{TV}(u, \hat{u}) \quad \forall \hat{u} \in H^1(\Omega)$$

We take $u_h = \sum_i u_i \phi_i$, $p_h = \sum_k p_k \phi_k$, and consider the basis function individually $\hat{u} = \phi_j$. Substituting this into the weak form of the Newton step (1.27), we arrive at

$$\int_{\Omega} \phi_j \left(\sum_k p_k \phi_k \right) + \frac{\beta}{\left(\left| \sum_i u_i \nabla \phi_i \right|^2 + \delta \right)^{1/2}} \left(I - \frac{\left(\sum_i u_i \nabla \phi_i\right) \otimes \left(\sum_i u_i \nabla \phi_i\right)}{\left| \sum_i u_i \nabla \phi_i \right|^2 + \delta} \right) \left(\sum_k p_k \nabla \phi_k \right) \cdot \nabla \phi_j \, dx$$

$$= -\int_{\Omega} \left(\sum_i u_i \phi_i - d \right) \phi_j + \frac{\beta}{\left(\left| \sum_i u_i \nabla \phi_i \right|^2 + \delta \right)^{1/2}} \left(\sum_i u_i \nabla \phi_i \cdot \nabla \phi_j \right) dx \tag{1.43}$$

Updating gives us:

$$\sum_k p_k \int_{\Omega} \phi_j \phi_k + \frac{\beta}{(\sum_i u_i \nabla \phi_i)^2 + \delta)^{1/2}} \left(I - \frac{\left(\sum_i u_i \nabla \phi_i\right) \otimes \left(\sum_i u_i \nabla \phi_i\right)}{(\sum_i u_i \nabla \phi_i)^2 + \delta} \right) \nabla \phi_j \cdot \nabla \phi_k \, dx$$

$$= -\int_{\Omega} \left(\sum_i u_i \phi_i - d \right) \phi_j + \beta \frac{\left(\sum_i u_i \nabla \phi_i\right) \cdot \nabla \phi_j}{(\sum_i u_i \nabla \phi_i)^2 + \delta)^{1/2}} \, dx \tag{1.44}$$

DTO (Ritz method):

$$\mathcal{F}^{\delta}_{TV}(u) = \mathcal{F}_{LS} + \mathcal{R}^{\delta}_{TV} = \frac{1}{2} \int_{\Omega} (u - d)^2 dx + \int_{\Omega} \beta (\nabla u \cdot \nabla u + \delta)^{1/2} dx \tag{1.45}$$

we take $u \approx u_h = \sum_i u_i \phi_i$. Substituting into the energy functional:

$$\mathcal{G}^{\delta}_{TV}(u) = \frac{1}{2} \int_{\Omega} (\sum_i u_i . \phi_i - d)^2 dx + \beta \left(\left(\sum_i u_i \nabla \phi_i \right) \cdot \left(\sum_i u_i \nabla \phi_i \right) + \delta) \right)^{1/2} dx \tag{1.46}$$

Now, we take the gradient for the j-th component:

$$g_j = \frac{\partial \mathcal{G}^{\delta}_{TV}(u)}{\partial u_j} = \int_{\Omega} \left(\sum_i u_i \phi_i - d \right) \phi_j + \beta \frac{\left(\sum_i u_i \nabla \phi_i\right) \cdot \nabla \phi_j}{\left(\sum_i u_i \nabla \phi_i\right)^2 + \delta)^{1/2}} \, dx$$

Correspondinly we can get the hessian:

$$H_{jk} = \frac{\partial g_j}{\partial u_k}$$

$$= \int_\Omega \phi_j\phi_k + \frac{\beta}{\left(|\sum_i u_i\nabla\phi_i\right)^2 + \delta)|^{1/2}}\left(\mathcal{I} - \frac{(\sum_i u_i\nabla\phi_i)\otimes(\sum_i u_i\nabla\phi_i)}{\sum_i u_i\nabla\phi_i + \delta}\right)\nabla\phi_j\cdot\nabla\phi_k\,dx$$

The Newton step for p is given by $\sum_k H_{jk}p_k = -g_j$:

$$\sum_k p_k\int_\Omega \phi_j\phi_k + \frac{\beta}{(\sum_i u_i\nabla\phi_i)^2 + \delta)^{1/2}}\left(I - \frac{(\sum_i u_i\nabla\phi_i)\otimes(\sum_i u_i\nabla\phi_i)}{(\sum_i u_i\nabla\phi_i)^2 + \delta}\right)\nabla\phi_j\cdot\nabla\phi_k\,dx \quad (1.47)$$

$$= -\int_\Omega\left(\sum_i u_i\phi_i - d\right)\phi_j + \beta\frac{(\sum_i u_i\nabla\phi_i)\cdot\nabla\phi_j}{(\sum_i u_i\nabla\phi_i)^2 + \delta)^{1/2}}\,dx \quad (1.48)$$

From 1.48 and 1.45 we can conclude that OTD and DTO approaches are equivalent.

Problem-2: Accessing and Setting Up JupyterHub Server

LINK TO CVIPS SERVER

- Example implementations and supporting files for this assignment can be found in the folder
  - cvips_labs/Assignments/Assignment3
  - cvips_labs/Assignments/Assignment4

Problem-3: 2D Anisotropic Poisson Problem

Given:

Given a 2D anisotropic Poisson problem in domain Ω has strong form as:

$$-\nabla\cdot(A\nabla u) = f \quad \text{in} \quad \Omega \quad (3.1)$$

$$u = u_0 \quad \text{on} \quad \text{\Tau} \quad (3.2)$$

where $A \in R^{2x2}$ is a symmetric positive definite matrix, conductivity matrix. $f(x)$ is a given distributed source and $u_0(x)$ is the source on the boundary $\text{\Tau}$

(a) Weak form of the problem:

The weak form of the problem is obtained by multiplying the strong form by a test function $v \in H^1(\Omega)$ and integrating over the domain Ω:

$$-\int_\Omega \nabla\cdot(A\nabla u)v\,dx = \int_\Omega fv\,dx \quad (3.3)$$

Using the divergence theorem, we can write the left-hand side as:

$$-\int_\Omega \nabla\cdot(A\nabla u)v\,dx = \int_{\text{\Tau}} A\nabla u\cdot\nabla v\,dx - \int_{\partial\Omega} A\nabla u\cdot nv\,ds \quad (3.4)$$

where n is the outward unit normal to the boundary $\partial\text{\Tau}$. Using the boundary condition (3.2), the boundary term vanishes. Therefore, the weak form of the problem is:

$$\int_\Omega A\nabla u\cdot\nabla v\,dx = \int_\Omega fv\,dx \quad \forall v \in H^1(\Omega) \quad (3.5)$$

Energy Functionals, we be obtained by replacing v with u in the weak form:

$$min_{u\in H^1(\Omega),u|\text{\Tau}=u_0}\mathcal{F}(u) = \frac{1}{2}\int_\Omega A\nabla u\cdot\nabla u\,dx - \int_\Omega fu\,dx \quad (3.6)$$

(b) Using FEniCS to Solve Anisotropic Poisson Problem in 2D.

Given:

The domain $\Omega = [0,1]\times[0,1]$, is the circle with radius 1 around origin. The source term as:

$$f(x) = exp(-100((x)^2 + (y)^2)) \quad (3.7)$$

$$u_0 = 0 \tag{3.8}$$

The conductivity matrix A is given as:

$$A_1 = \begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix} \tag{3.9}$$

and

$$A_2 = \begin{bmatrix} 1 & -5 \\ -5 & 100 \end{bmatrix} \tag{3.10}$$

NOTE: We need to compare the results for A_1 and A_2.

Check code and results below (with analysis/discussion)->

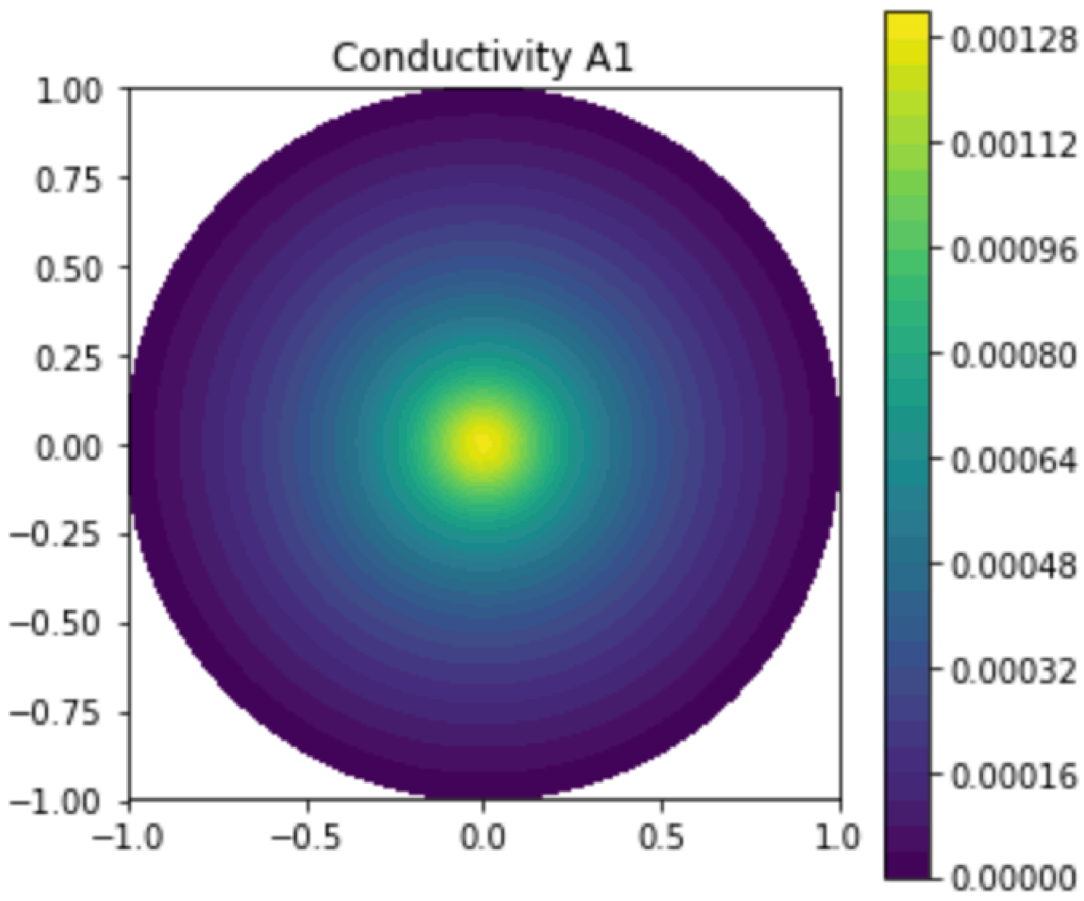
In [ ]:
```
# Install the necessary packages
# try loading hiPPylib if it fails install it
try:
    import hiPPylib
    import FEniCS
    import matplotlib
    import dolfin
except ImportError:
    print(f"Issue with some library, installing the necessary packages.")
    """
    !pip install --upgrade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

```
Issue with some library, installing the necessary packages.
```

In [ ]:
```
"""
Computational and Variational Methods for Inverse Problems (Spring 2024)
Assignment-03 -- Question-3 (b)

---> 2D Anisotropic Poisson Problem with domain at unit Circle.

Author: Shreshth Saini (saini.2@utexas.edu)
Date: 10th April 2024
"""

# Question 3(B)
#-----------------------------------------------------------------------------------------------

# Compute libraries
import dolfin as dl
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)


class AnisotropicPoisson2D():
    def __init__(self, mesh_file:str, func_form: str, conduct_tensor:list, space_degree=2, func
        self.mesh_file = mesh_file
        self.space_degree = space_degree
        self.conduct_tensor = conduct_tensor
        self.func_form = func_form
        self.func_degree = func_degree

    # defining and solving - flow
    def solution_flow(self):
        # obtaining mesh
        self.get_mesh(self.mesh_file)
        # getting function space
        self.function_space(self.space_degree)
        # functional equation
        self.define_functions(self.func_form, self.func_degree)
        # extracting conduct tensors
        for i in range(len(self.conduct_tensor)):
            self.conduct_tensor[i] = dl.Constant(self.conduct_tensor[i])

        # stiffness form
        self.stiffness_form = []
        for i in range(len(self.conduct_tensor)):
            self.stiffness_form.append(self.generate_forms(self.conduct_tensor[i]))

        # rhs form
        self.rhs_form = self.f*self.v_test*dl.dx

        # boundary condition
        self.bc = self.boundary_condition(dl.Constant(0.0), "on_boundary")

        # solving the system of equations
        self.U = []
        for i in range(len(self.stiffness_form)):
            self.U.append(self.solve_system(self.stiffness_form[i], self.rhs_form, self.bc))

        # make the plots
        for i in range(len(self.U)):
            self.plot_solution(self.U[i], f"Conductivity A{i+1}")

    # 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

    # defining the finite element space
    def function_space(self, degree=2):
        # defining the function space
        self.V = dl.FunctionSpace(self.mesh, "CG", degree)
        # defining the test and trial functions
        self.u = dl.TrialFunction(self.V)
        # defining the test function
        self.v_test = dl.TestFunction(self.V)

    # defining the functons
    def define_functions(self, func_form, degree=4):
        # defining the functions
        self.f = dl.Expression(func_form , degree=degree)

    # generating the stiffness
    def generate_forms(self, conduct_tensor):
        # defining the bilinear and linear forms
        return dl.inner(dl.dot(conduct_tensor, dl.grad(self.u)), dl.grad(self.v_test))*dl.dx

    # boundary condition
    def boundary_condition(self, u_D, boundary):
        # defining the boundary condition
        bc = dl.DirichletBC(self.V, u_D, boundary)
        return bc

    # solve the system
    def solve_system(self,stiffness_form, rhs, bc):
        # solving the system
        # solution init
        U = dl.Function(self.V)
        dl.solve(stiffness_form == rhs, U, bc)
        return U

    # plot the solution
    def plot_solution(self, U, title="Conductivity not specified"):
        # plot the solution
        plt.figure(figsize=(5,5))
        p = dl.plot(U)
        plt.title(title)
        plt.colorbar(p)
        # save the image
        plt.savefig(title+".png")
        plt.show()


# main function

if __name__ == "__main__":
    # input parameters
    mesh_file = "P3/circle.xml"
    func_form = "exp(-100*(pow(x[0],2)+pow(x[1],2)))"
    conduct_tensor = [((10.0, 0.0),(0.0, 10.0)), ((1.0 ,-5.0),(-5.0,100.0))]
    space_degree = 2
    func_degree = 4

    # solving the problem
    anisotropic_poisson = AnisotropicPoisson2D(mesh_file, func_form, conduct_tensor, space_degr
    anisotropic_poisson.solution_flow()
```

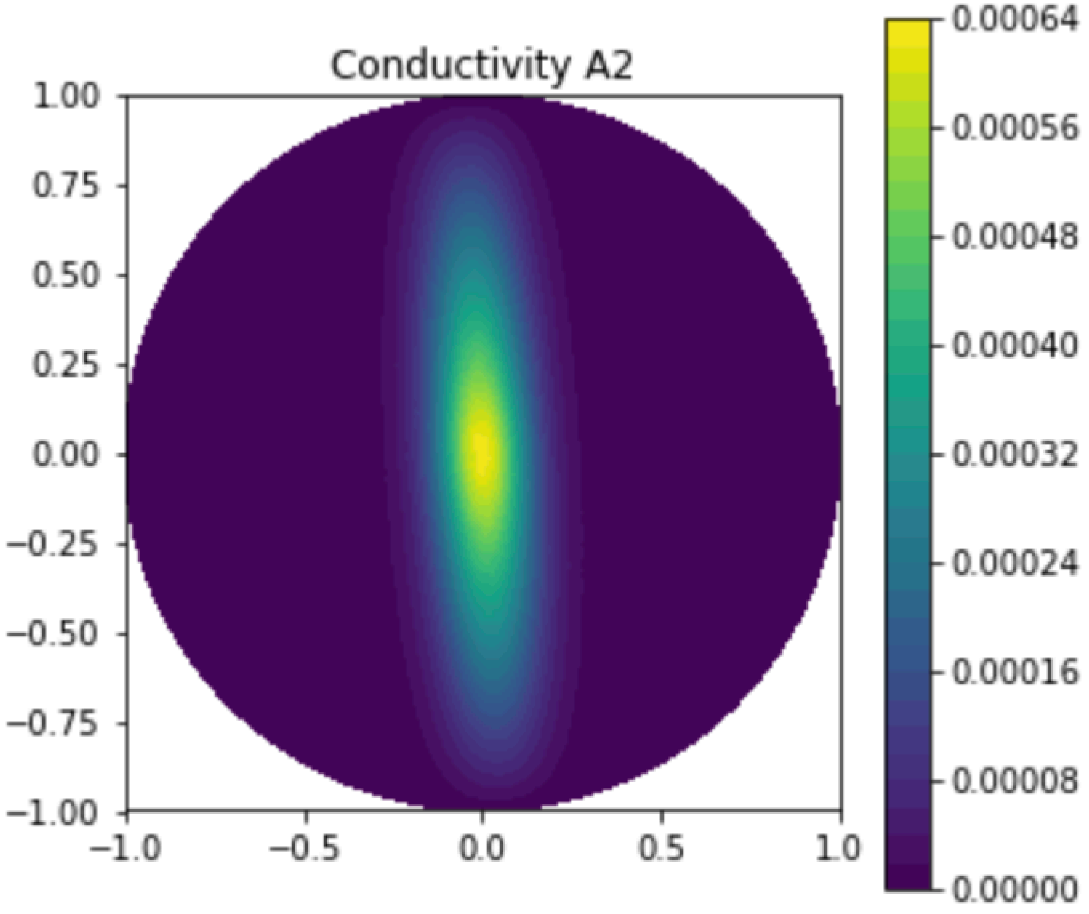
Analysis of results

figures:


Conductivity A1
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0
0.00128
0.00112
0.00096
0.00080
0.00064
0.00048
0.00032
0.00016
0.00000

```
<Figure size 432x288 with 0 Axes>
```

1.



2.
```
<Figure size 432x288 with 0 Axes>
```

The conductivity matrix A_1 is isotropic and has the same conductivity in both x and y directions(radially symmetric). The conductivity matrix A_2 is anisotropic and has 100 times more conductivity in the y direction than x. The ellipse is not entirely vertical due to off-diagonal terms.

Problem-4: Solving Image Denoising using TN and TV:

Given:

Given a 2D image.dat file.

(a) Image Denoising using TN regularization:

NOTE: we can use linear solver since gradient is linear in u. And use $\beta > 0$ giving a reasonable reconstruction(avoids over smoothing).

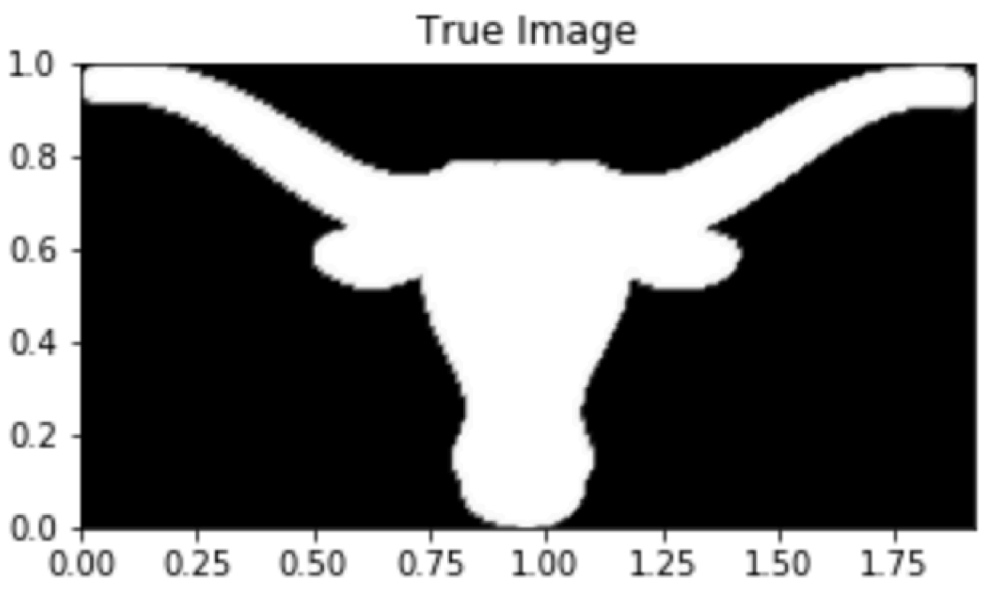
Check the code and results below (with discussion)->

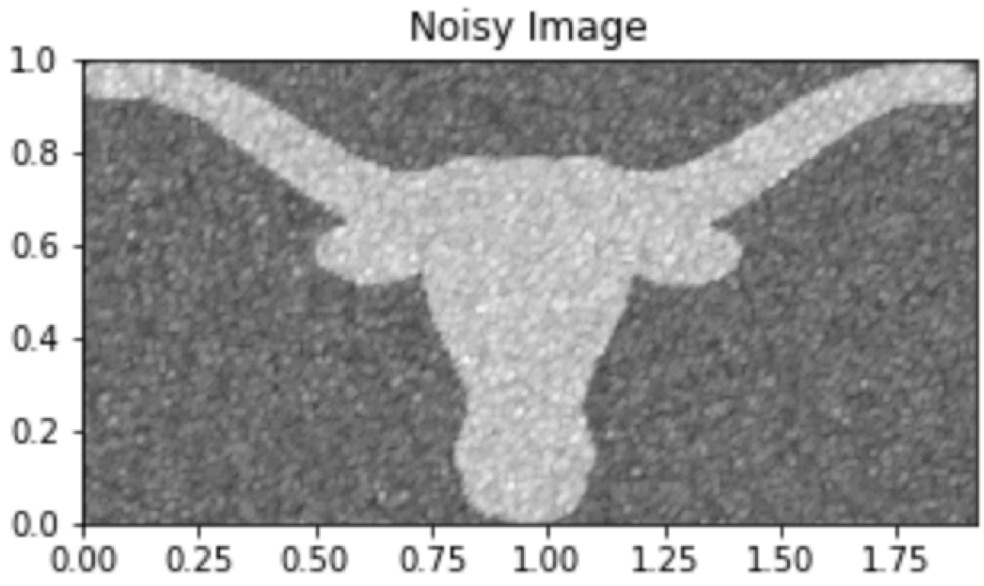
In [ ]:
```
"""
Computational and Variational Methods for Inverse Problems (Spring 2024)
Assignment-03 -- Question-4 (a)

---> 2D Image Denoising with Tikhonov regularization.

Author: Shreshth Saini (saini.2@utexas.edu)
Date: 10th April 2024
"""

# Question 4(a)
#----------------------------------------------------------------------------------------------

# Compute libraries
```

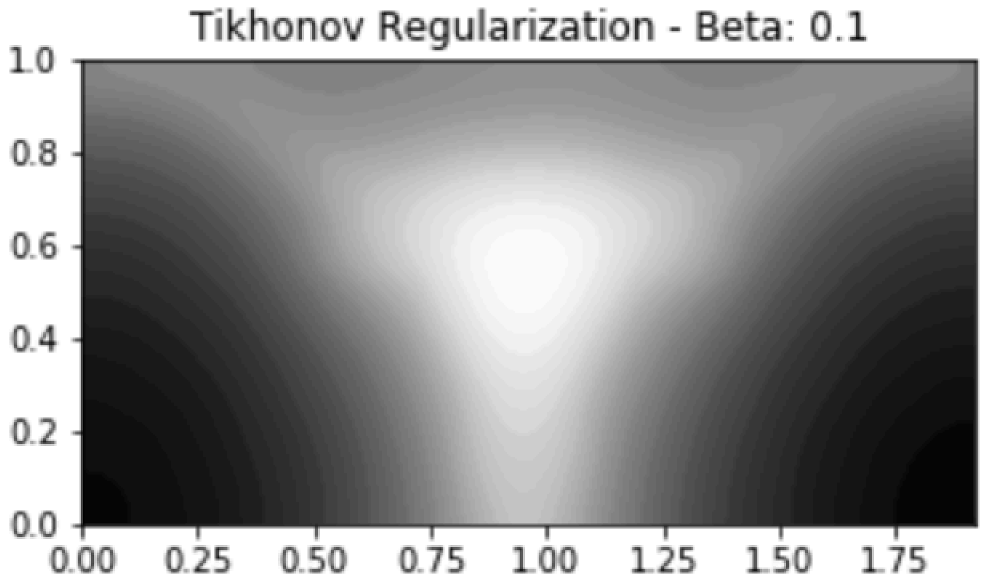
```
import dolfin as dl
import numpy as np
import mshr
import hippylib as hp
import ufl

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

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

# custom libraries
try:
    from P4.unconstrainedMinimization import InexactNewtonCG
except ImportError:
    print(f"Custom library not found. Trying to load with current directory...")
    try:
        from unconstrainedMinimization import InexactNewtonCG
    except:
        print(f"Custom library not found. Please check the path.")
        exit()

#-----------------------------------------------------------------------------------------------

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


class image_denoise():
    def __init__(self, image_path, noise_std_dev=0.3, delimiter=',', **kwargs):
        self.noise_std_dev = noise_std_dev
        self.image_path = image_path
        self.delimiter = delimiter

    # solving with TN
    def TN_solver(self, noisy, V, delta=0.01, solver=None, beta=1e-3, max_iter=100):
        # NOTE: we are only using linear since our gradients take linear form
        # defining the functionals
        beta = dl.Constant(beta)
        # defining the functional
        u_trial = dl.TrialFunction(V)
        u_test = dl.TestFunction(V)
        var_F_tn = (u_trial - noisy)*u_test*dl.dx + beta*dl.inner(dl.grad(u_trial), dl.grad(u_t

        # solving using dl.solve
        u_tn = dl.Function(V)
        # solving
        dl.solve(dl.lhs(var_F_tn) == dl.rhs(var_F_tn), u_tn)

        return u_tn

    # load the image
    def load_image(self):
        if self.image_path==None:
            print(f"Image path not provided.")
            return None
        else:
            try:
                image = np.transpose(np.loadtxt(self.image_path, delimiter=self.delimiter))
            except:
                print(f"Error loading image. Trying in current directory...")
                try:
                    image = np.transpose(np.loadtxt(self.image_path.split("/")[-1], delimiter=s
                except:
                    print(f"Error loading image. Please check the path.")
                    return None
        return image
    # generate noise
    def generate_noise(self, image):
        noise = self.noise_std_dev*np.random.randn(image.shape[0], image.shape[1])
        return noise

    # generate mesh
    def generate_mesh(self, nx=200, ny=100):
        # generate mesh
        mesh = dl.RectangleMesh(dl.Point(0,0), dl.Point(self.Lx, self.Ly), nx, ny)
        return mesh

    # image structure
    def image_structure(self, h, image, noise, **kwargs):
        true_image = hp.NumpyScalarExpression2D()
        true_image.setData(image, h,h)
        noise_image = hp.NumpyScalarExpression2D()
        noise_image.setData(image + noise, h,h)
        return true_image, noise_image

    # scale the data/image into mesh
    def interpolate_image(self, image, V):
        return dl.interpolate(image, V)

    # plot the solution
    def plot_images(self, img, title="Image not specified"):
        # getting range
        vmin = np.min(img.vector().get_local())
        vmax = np.max(img.vector().get_local())

        # plot the solution
        plt.figure(figsize=(5,5))
        dl.plot(img, title=title, vmin=vmin, vmax=vmax, cmap="gray")
        plt.savefig(title+".png")
        plt.show()

    # solution flow
    def solution_flow(self, space_deg = 1, **kwargs):
        # loading image
        self.image = self.load_image()
        # defining noise
        self.noise = self.generate_noise(self.image)
        # defining the domain
        self.Lx = float(self.image.shape[0])/float(self.image.shape[1])
        self.Ly = 1.0
        self.h = self.Ly/float(self.image.shape[1])

        # generate mesh
        self.mesh = self.generate_mesh()
        # function space
        self.V = dl.FunctionSpace(self.mesh, "Lagrange", space_deg)

        # defining the true and noise image
        self.true_image, self.noise_image = self.image_structure(self.h, self.image, self.noise

        # Interpolation of a given function into a given finite element space.
        self.u_true = self.interpolate_image(self.true_image, self.V)
        self.u_noise = self.interpolate_image(self.noise_image, self.V)

        # show true and noisy image
        self.plot_images(self.u_true, "True Image")
        self.plot_images(self.u_noise, "Noisy Image")

        # solving the problem
        if "solver" in kwargs:
            if kwargs["solver"] == "TN":
                betas = np.logspace(-1,-6,6)
                delta = None
                solver = None
                image_plot_title = "Tikhonov Regularization - Beta: "
                l_curve_title = "TN_L-Curve"

            else:
                print(f"Solver not implemented.")
                exit()
            residual_tn_norm = []
            u_tn_norm = []
            u_tn_list = []
            for beta in betas:
                if kwargs["solver"] ==  "TN":
                    u_tn = self.TN_solver(self.u_noise, self.V, delta= delta, solver=solver, be

                # error calculation
                residual_tn_norm.append(np.linalg.norm(u_tn.vector()[:] - self.u_true.vector()[
                u_tn_norm.append(np.linalg.norm(u_tn.vector()[:]))
                u_tn_list.append(u_tn.copy())

                # plot the denoise version with beta
                self.plot_images(u_tn, image_plot_title+str(beta))

            # plot the l-curve
            plt.figure(figsize=(5,5))
            plt.loglog(residual_tn_norm, u_tn_norm, 'k--')
            for i , txt in enumerate(betas):
                plt.loglog(residual_tn_norm[i], u_tn_norm[i], 'o')
                plt.annotate(txt, (residual_tn_norm[i], u_tn_norm[i]))
            plt.xlabel(r"Residual norm $|| u - u_{noisy}||_2$")
            plt.ylabel(r"Solution norm $||u||_2$")
            plt.title(l_curve_title)
            plt.grid(True, which="both")

            plt.savefig(l_curve_title+".png")




# main function

if __name__ == "__main__":
    # input parameters
    np.random.seed(1)
    solver = "TN"
    img_path = "P4/image.dat"
    noise_level = 0.3

    # solving the problem
    image_denoise = image_denoise(img_path, noise_level)
    image_denoise.solution_flow(solver=solver)
```

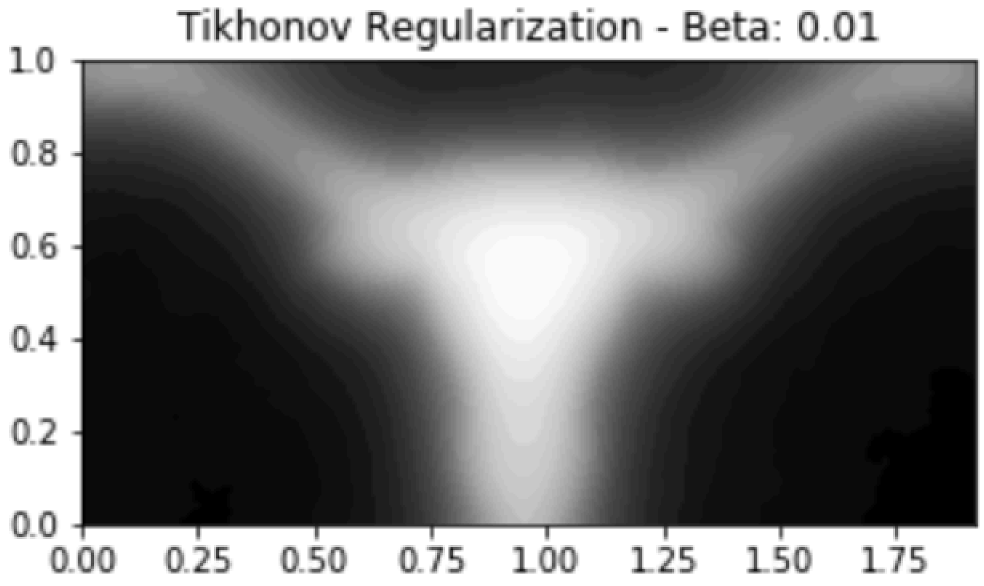
Discussion and Results

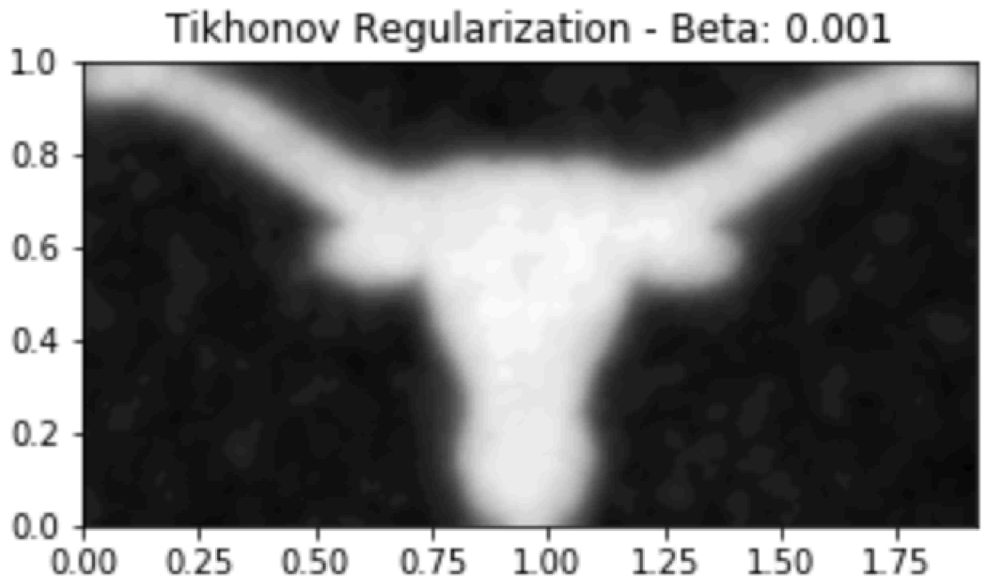
figures:

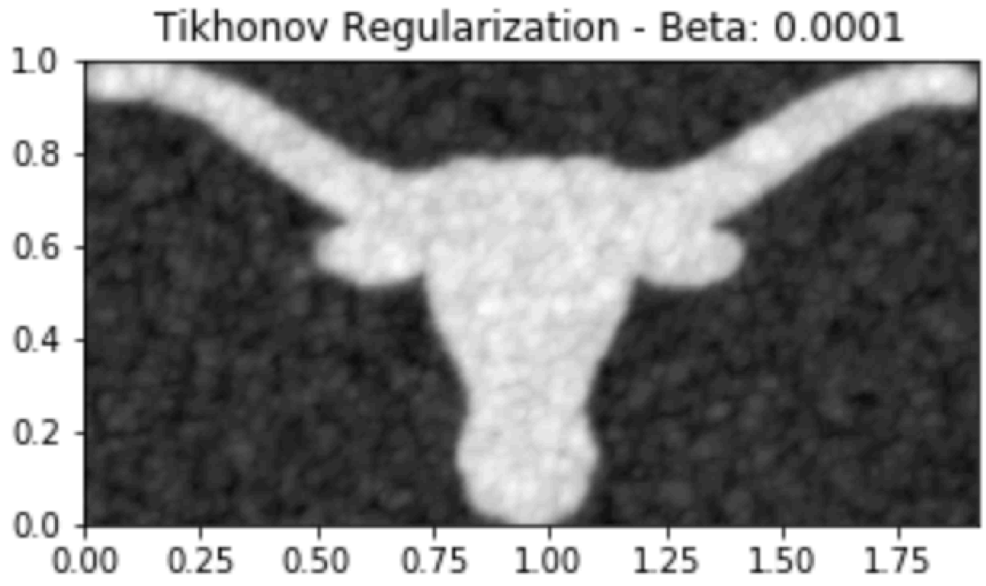


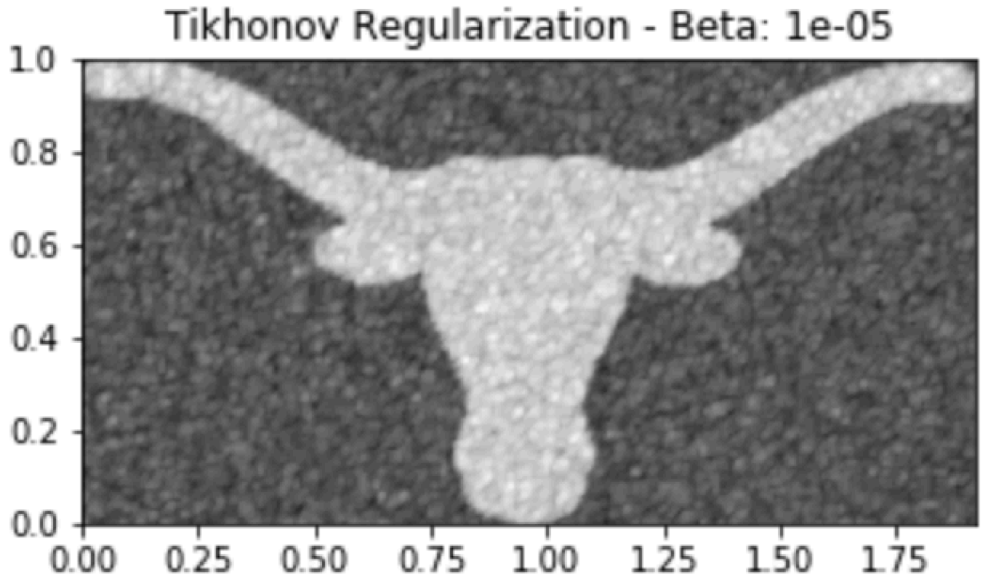

Noisy Image
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75

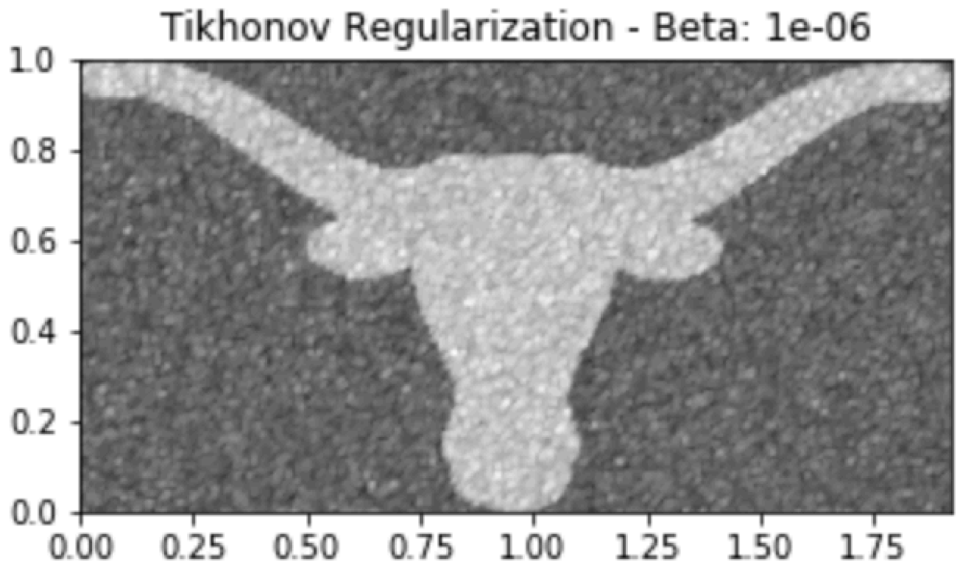

Tikhonov Regularization - Beta: 0.1
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75


Tikhonov Regularization - Beta: 0.01
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75

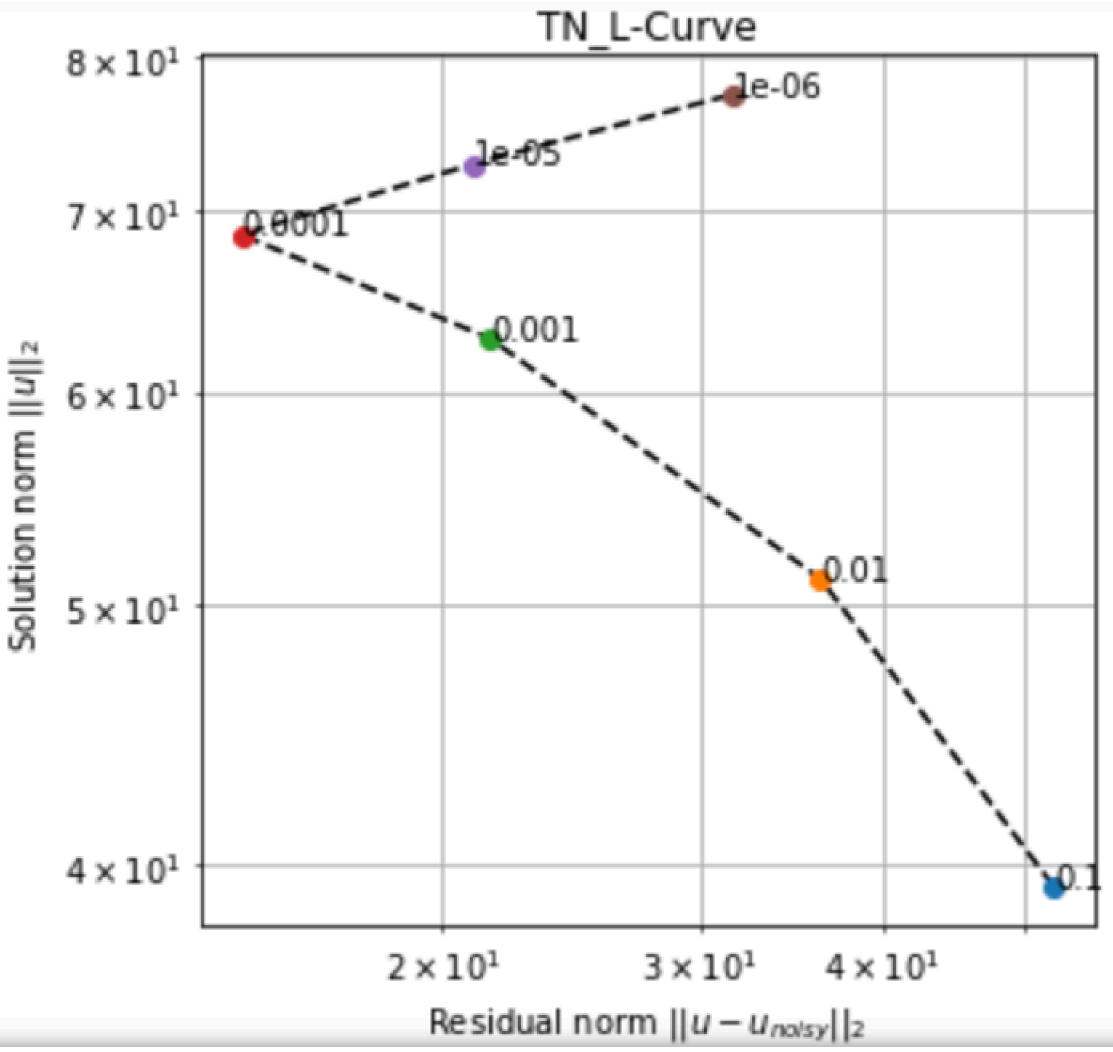

Tikhonov Regularization - Beta: 0.001
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75


Tikhonov Regularization - Beta: 0.0001
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75


Tikhonov Regularization - Beta: 1e-05
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75



L-Curve:



Optimal β is close to 10^{-3}, because it gives a reasonable reconstruction without over-smoothing, and L-curve verifies that.

(b) Image Denoising using TV regularization:

NOTE: gradient is non-linear in u, we use InexactNewtonCG non-linear solver.

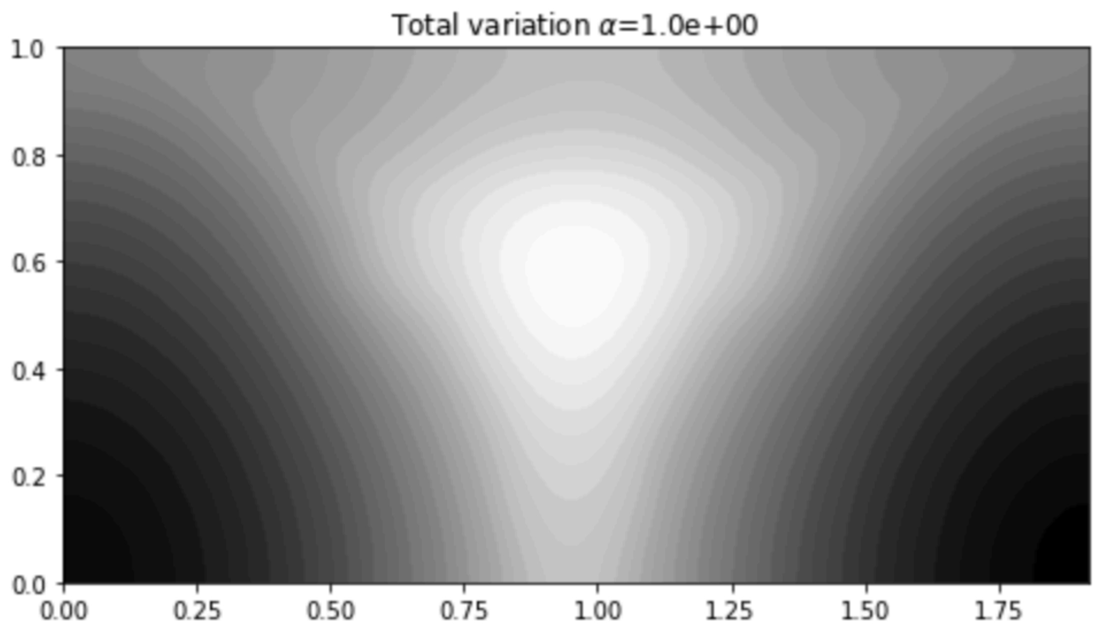
Check the code and results below (with discussion)->

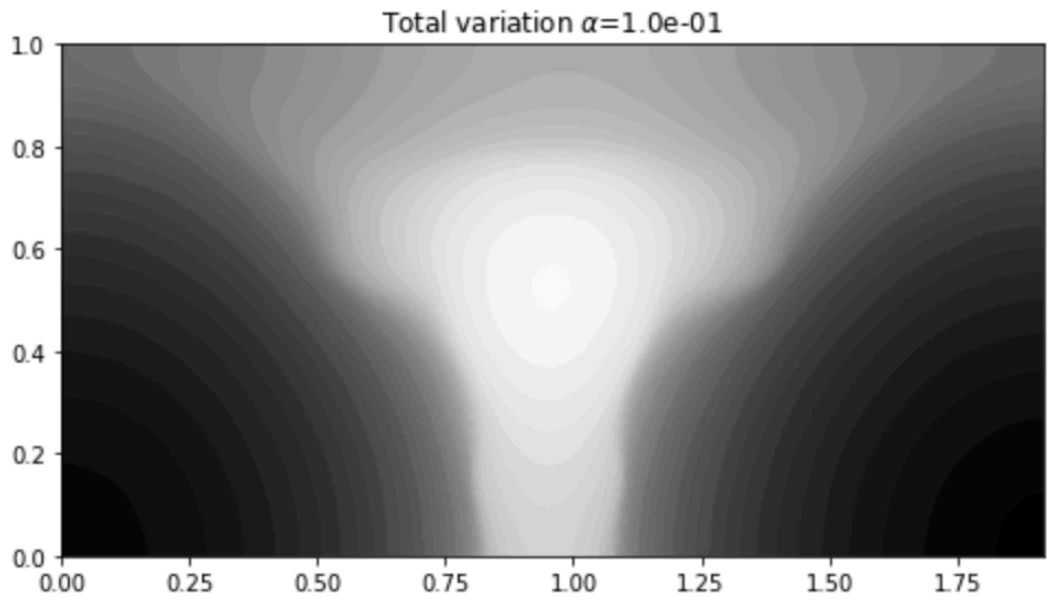
```
In [ ]: # we use same code base as part - and add the new solver in it.

        """
        Computational and Variational Methods for Inverse Problems (Spring 2024)
        Assignment-03 -- Question-4 (b)

        ---> 2D Image Denoising with Total Variation.

        Author: Shreshth Saini (saini.2@utexas.edu)
        Date: 10th April 2024
        """

        # Question 4(b)
        #------------------------------------------------------------------------------------------------

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

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

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

        # custom libraries
        try:
            from P4.unconstrainedMinimization import InexactNewtonCG
        except ImportError:
            print(f"Custom library not found. Trying to load with current directory...")
            try:
                from unconstrainedMinimization import InexactNewtonCG
            except:
                print(f"Custom library not found. Please check the path.")
                exit()

        #------------------------------------------------------------------------------------------------

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


        class image_denoise():
            def __init__(self, image_path, noise_std_dev=0.3, delimiter=',', **kwargs):
                self.noise_std_dev = noise_std_dev
                self.image_path = image_path
                self.delimiter = delimiter

            # load the image
            def load_image(self):
                if self.image_path==None:
                    print(f"Image path not provided.")
                    return None
                else:
                    try:
                        image = np.transpose(np.loadtxt(self.image_path, delimiter=self.delimiter))
                    except:
                        print(f"Error loading image. Trying in current directory...")
                        try:
                            image = np.transpose(np.loadtxt(self.image_path.split("/")[-1], delimiter=s
                        except:
                            print(f"Error loading image. Please check the path.")
                            return None
                return image
```

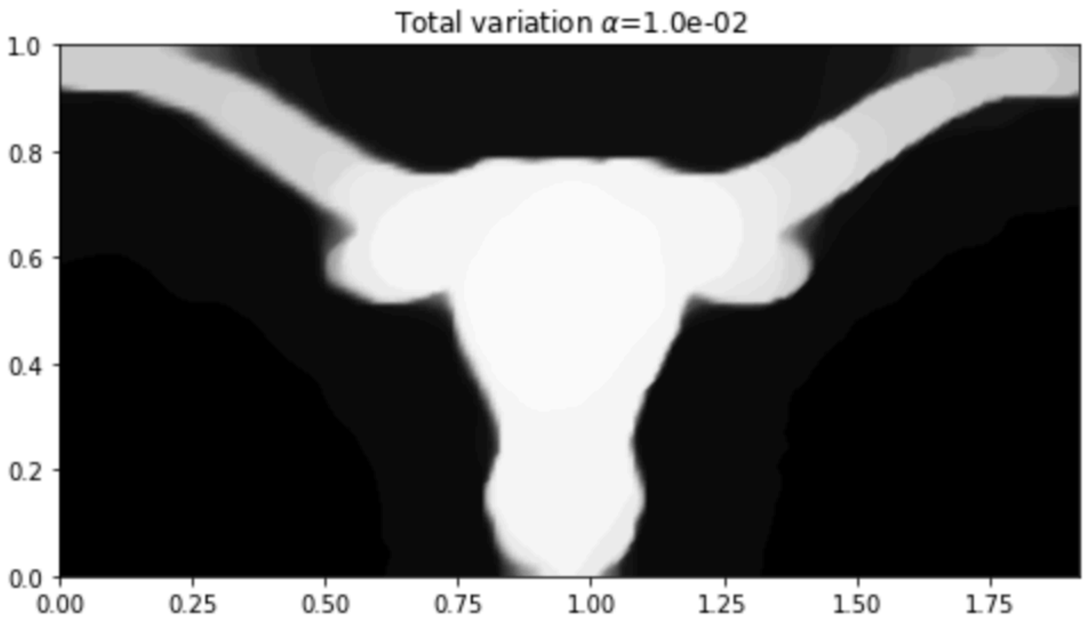
```
    # generate noise
    def generate_noise(self, image):
        noise = self.noise_std_dev*np.random.randn(image.shape[0], image.shape[1])
        return noise

    # generate mesh
    def generate_mesh(self, nx=200, ny=100):
        # generate mesh
        mesh = dl.RectangleMesh(dl.Point(0,0), dl.Point(self.Lx, self.Ly), nx, ny)
        return mesh

    # image structure
    def image_structure(self, h, image, noise, **kwargs):
        true_image = hp.NumpyScalarExpression2D()
        true_image.setData(image, h,h)
        noise_image = hp.NumpyScalarExpression2D()
        noise_image.setData(image + noise, h,h)
        return true_image, noise_image

    # scale the data/image into mesh
    def interpolate_image(self, image, V):
        return dl.interpolate(image, V)

    # solution flow
    def solution_flow(self, space_deg = 1, **kwargs):
        # loading image
        self.image = self.load_image()
        # defining noise
        self.noise = self.generate_noise(self.image)
        # defining the domain
        self.Lx = float(self.image.shape[0])/float(self.image.shape[1])
        self.Ly = 1.0
        self.h = self.Ly/float(self.image.shape[1])

        # generate mesh
        self.mesh = self.generate_mesh()
        # function space
        self.V = dl.FunctionSpace(self.mesh, "Lagrange", space_deg)

        # defining the true and noise image
        self.true_image, self.noise_image = self.image_structure(self.h, self.image, self.noise

        # Interpolation of a given function into a given finite element space.
        self.u_true = self.interpolate_image(self.true_image, self.V)
        self.u_noise = self.interpolate_image(self.noise_image, self.V)

        # show true and noisy image
        self.plot_images(self.u_true, "True Image")
        self.plot_images(self.u_noise, "Noisy Image")

        # solving the problem

        # Define solver
        solver = InexactNewtonCG()
        solver.parameters["rel_tolerance"] = 1e-6
        solver.parameters["abs_tolerance"] = 1e-9
        solver.parameters["gdu_tolerance"] = 1e-18
        solver.parameters["max_iter"] = 5000
        solver.parameters["c_armijo"] = 1e-5
        solver.parameters["print_level"] = 1
        solver.parameters["max_backtracking_iter"] = 10

        # function
        def TV_functions(beta,delta,u_func):
            beta = dl.Constant(beta) #Regularization parameters
            delta = dl.Constant(delta) #Added term in the regularization
            F_tv  = dl.Constant(0.5) * (u_func - self.u_noise)**2 * dl.dx + beta * dl.sqrt(dl.i
            return F_tv

        # TN regularization
        betas = np.logspace(0,-4,5)
        delta  = 0.01
        res_tv_norm = []
        u_tv_norm   = []
        u_tv_list   = []

        for beta in betas:
            u_tv = dl.Function(self.V)
            F_tv = TV_functions(beta,delta,u_tv)
            solver.solve(F_tv, u_tv)

            # Misfit
            res_tv_norm.append(np.linalg.norm(u_tv.vector()[:] - self.u_noise.vector()[:]))
            u_tv_norm.append(np.linalg.norm(u_tv.vector()[:]))
            u_tv_list.append(u_tv.copy())

            #plotting the image
            plt.figure(figsize=(8,8))
            dl.plot(u_tv, title="Total variation $\\alpha$=%.1e" %(beta))
            plt.savefig("Q4_TN_%.1e_.pdf" %(beta))

        # plot l-curve
        fig,ax = plt.subplots(figsize=(10,8))
        plt.loglog(res_tv_norm, u_tv_norm,'k--')
        for i, txt in enumerate(betas):
            ax.loglog(res_tv_norm[i], u_tv_norm[i],'o')
            ax.annotate(txt, (res_tv_norm[i], u_tv_norm[i]))

        plt.grid(True, which="both")
        plt.xlabel(r'$||u - u_{Noisy}||_2$')
        plt.ylabel(r'$||u||_2$')
        plt.title("Total Variation's L-curve")
        plt.savefig("Q4_TV_lcurve.pdf")


        # for delta observations:
        # TN regularization
        beta = 1e-2
        deltas= np.logspace(1,-4,6)
        num_iters=[]

        for delta in deltas:
            u_tv = dl.Function(self.V)
            F_tv = TV_functions(beta,delta,u_tv)
            solver.solve(F_tv, u_tv)
            num_iters.append(solver.it)
        # plot iteration dependence on the delta
        fig,ax = plt.subplots(figsize=(10,8))
        ax.semilogx(deltas, num_iters,'ro')
        ax.semilogx(deltas, num_iters,'k--')
        plt.grid(True, which="both")
        plt.xlabel(r'$\delta$')
        plt.ylabel(r'Newton iterations')
        plt.title("Convergence dependence on $\delta$")
        plt.savefig("Q4_TV_Newton.pdf")


# main function

if __name__ == "__main__":
    # input parameters
    np.random.seed(1)
    solver = "TN"
    img_path = "P4/image.dat"
    noise_level = 0.3

    # solving the problem
    image_denoise = image_denoise(img_path, noise_level)
    image_denoise.solution_flow()
```

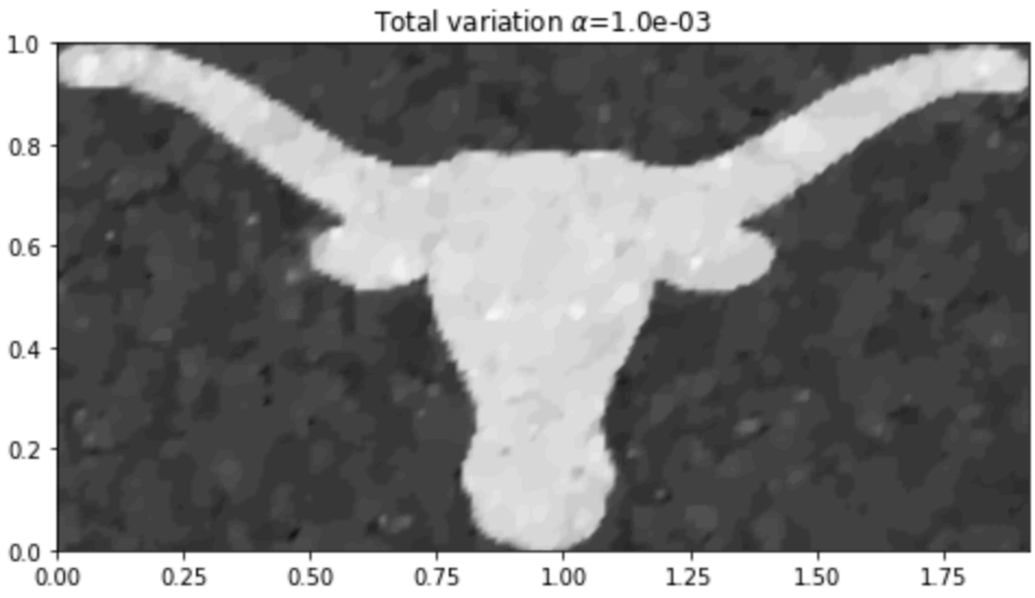
Discussion:

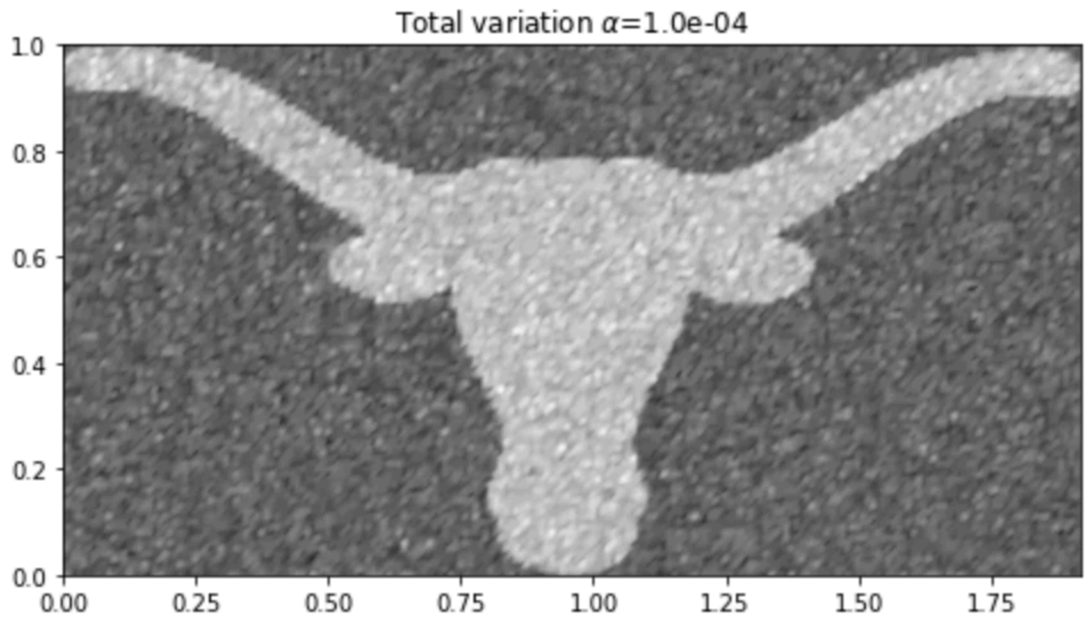
Figures:


Total variation α=1.0e+00
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75

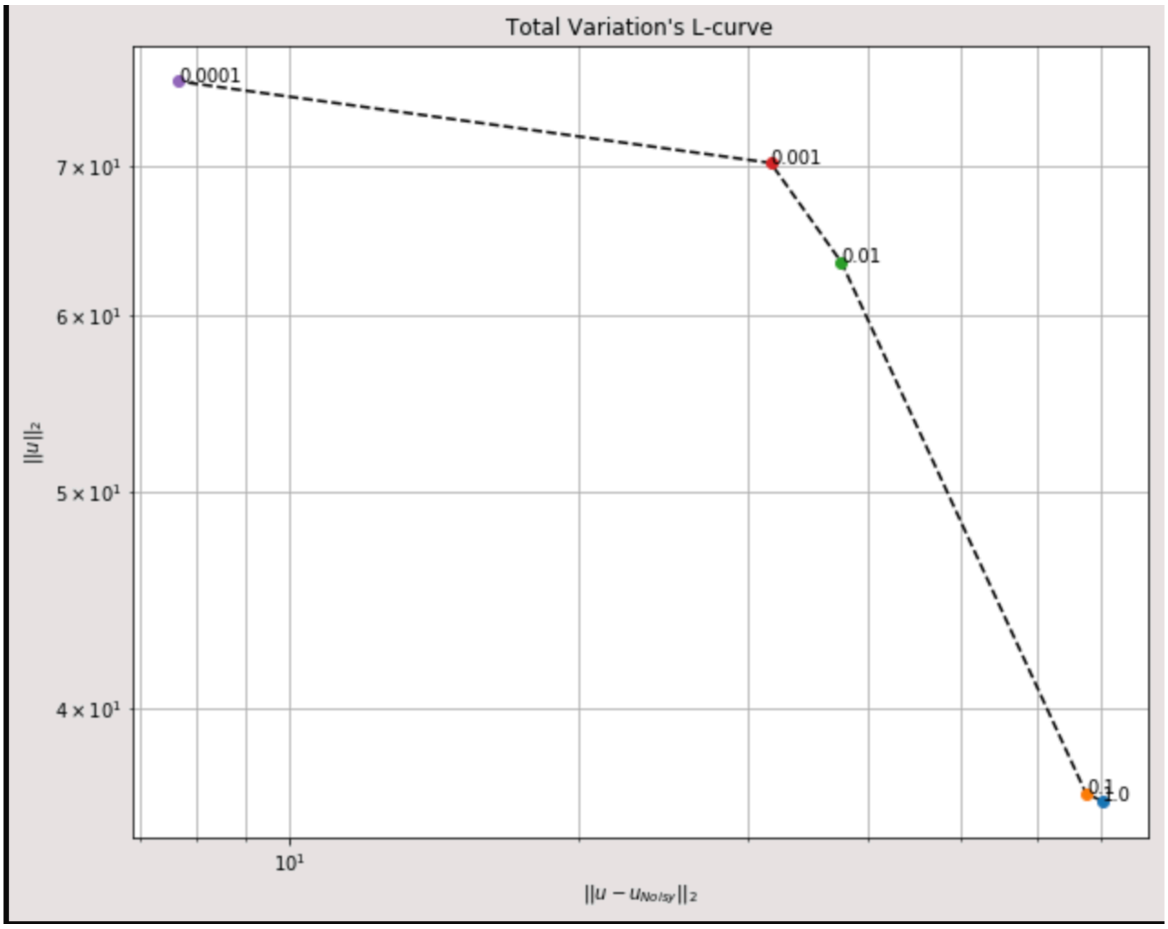

Total variation α=1.0e-01
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75


Total variation α=1.0e-02
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75


Total variation α=1.0e-03
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75



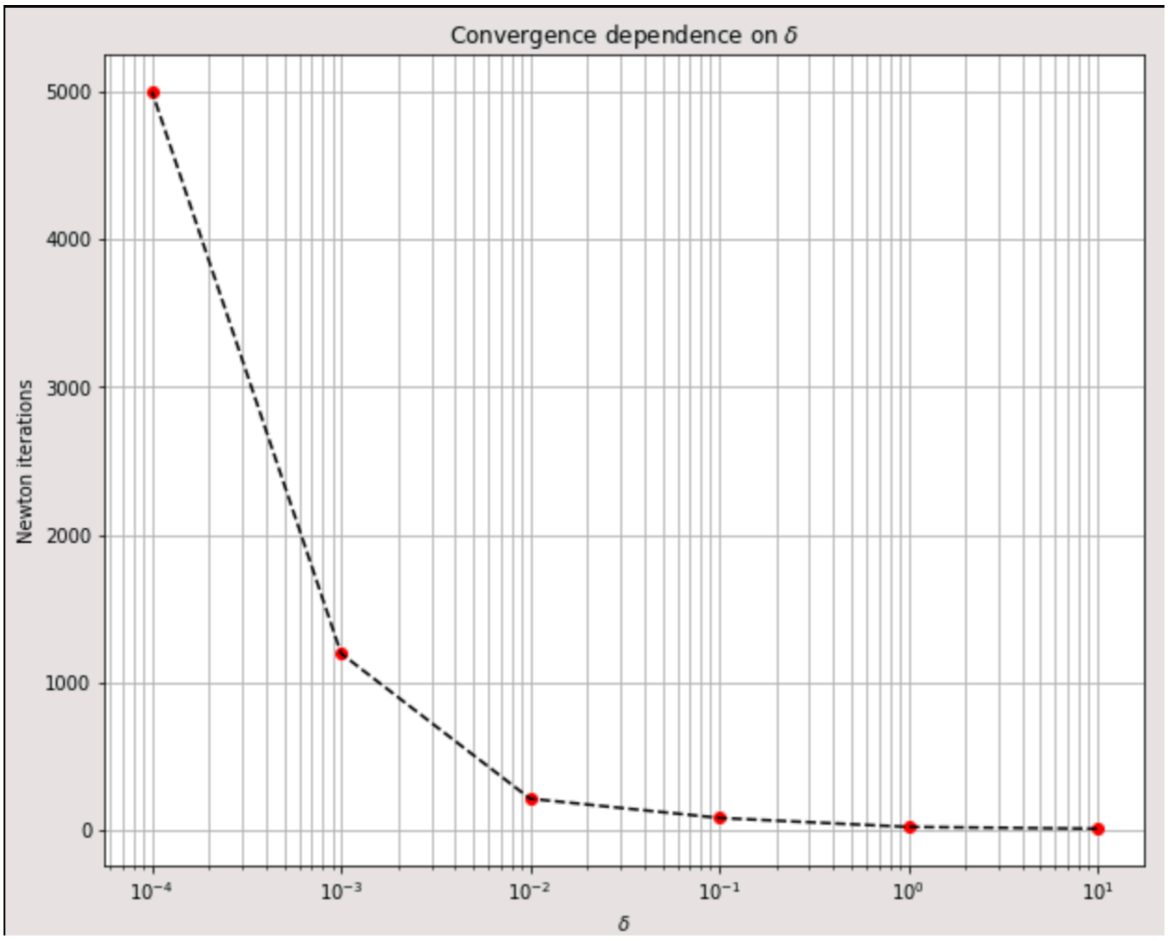
L-Curve:



We can observe that optimal β is 10^{-2}

δ effect

Figures:



Increasing the δ causes a sharp drop in the Newton steps for convergence, plateauing at almost a constant value. This happens as Hessian becomes more ill-conditioned when the δ is reduced. Also, the size of the valley of optimal convergence also reduces with decreasing δ. So, it takes more steps to find that valley.

(c) Comparing denoised images from TN and TV:

From the optimal results for TV and TN regularizations, it is evident that the TV regularization preserves the edges but TN regularization smears sharp edges. This occurs since TN solves the diffusion problem isotropically, i.e., both in the direction of fastest gradient and its perpedicular direction.

However, TV solves the diffusion problem in the direction perpendicular to the steepest gradient. This leads to the Newton steps that encourage diffusion and subsequently, smoothness. Moreover, it discourages diffusion and preserves sharp edges in the direction of the gradients.