Optimize-then-discretize versus discretize-then-optimize for an inverse problem governed by the time-dependent advection-diffusion PDE

Math 292 Special topics on Inverse Problems project

Irabiel Romero

Applied Mathematics Department
UC Merced
January 17, 2022

Introduction

When solving inverse problems, there are two methods one can do. They can either optimize-then-discretize (OTD) or discritize-then-optimize (DTO). These two methods usually commute. In the paper [2], it was shown that OTD and DTO maybe not commute. In this paper, we will come to the same conclusion, but for the case of inverse problem where we discritize the time-dependent PDE with backwards Euler.

This paper will serve two purposes. First, it will show an example of how to solve an inverse problem with a time-dependent advection-diffusion PDE. Second, show two ways one can conclude that the OTD and DTO do not commute.

Problem description

In what follows, let $\Omega \subset \mathbf{R}^n$ be open and bounded and consider measurements on a part $\Gamma_m \subset \partial\Omega$ of the boundary over the time horizon $[T_1, T]$, with $0 < T_1 < T$. The inverse problem, from the technical report [1], is formulated as follows:

$$\min_m J(m) := \frac{1}{2}\int_{T_1}^{T}\int_{\Gamma_m}(u-u_d)^2 dxdt + \frac{\gamma_1}{2}\int_\Omega m^2 dx + \frac{\gamma_2}{2}\int_\Omega |\nabla m|^2 dx, \tag{1}$$

where u is the solution of

$$u_t - \kappa\Delta u + \mathbf{v}\cdot\nabla u = 0 \quad \text{in } \Omega\times[0,T], \tag{2}$$
$$u(x,0) = u_0(x) \quad \text{in } \Omega, \tag{3}$$
$$\kappa\nabla u\cdot\mathbf{n} = 0 \quad \text{on } \partial\Omega\times[0,T]. \tag{4}$$

The velocity field, $\mathbf{v}$, is computed by solving the steady Navier Stokes equation with the side walls driving the flow,

$$-\frac{1}{Re}\Delta\mathbf{v} + \nabla q + \mathbf{v}\cdot\nabla\mathbf{v} = 0 \quad \text{in } \Omega, \tag{5}$$
$$\nabla\cdot\mathbf{v} = 0 \quad \text{in } \Omega, \tag{6}$$
$$\mathbf{v} = \mathrm{g} \quad \text{in } \partial\Omega. \tag{7}$$

First, we will begin with OTD, then we will do DTO. For both methods we will derive their gradient and Hessian-apply Newton system to see how both methods differ. We will start by doing OTD. For simplicity let $u_0(x) = m(x)$.

Optimize-then-discritize

To do this approach, we need the weak form of the forward problem, the Lagrangian of equation(1), derive the adjoint, and the gradient. Then we can find the adjoint equation,the gradient, state incremental equation, adjoint incremental equation. Before we define the Lagrangian, we fine to find the weak form the the forward problem.

1. Weak form of the state equation (2): Scale equation (2) by $p \in \mathcal{V}$.

$$\int_0^T\int_\Omega u_t p\ dxdt - \kappa\Delta up\ dxdt + \mathbf{v}\cdot\nabla up\ dxdt = 0$$

where $\mathcal{V} := \{v \in H^1(\Omega),\ \text{for each } t \in (0,T)\}$ and $\mathcal{M} := H^1(\Omega)$. By integration by part

$$\int_0^T\int_\Omega \kappa\Delta up\ dxdt = \int_0^T\int_{\partial\Omega}\kappa\nabla u\cdot\mathbf{n}p\ dSdt - \int_0^T\int_\Omega \kappa\nabla u\cdot\nabla p\ dxdt$$

Recall, the boundary condition $\kappa\nabla u\cdot\mathbf{n} = 0$. Thus,

$$\int_0^T\int_\Omega \kappa\Delta up\ dxdt = -\int_0^T\int_\Omega \kappa\nabla u\cdot\nabla p\ dxdt$$

Thus the weak form of u: Find $u \in \mathcal{V}$ such that

$$\int_0^T\int_\Omega u_t p\ dxdt + \kappa\nabla u\cdot\nabla p\ dxdt + \mathbf{v}\cdot\nabla up\ dxdt, \quad \forall p \in V.$$

2. The Lagrangian function: We can now define the Lagrangian function

$$\mathcal{L}(u,m,p,p_0) = J(m) + \int_0^T \int_\Omega u_t p \; dxdt + \int_0^T \int_\Omega \kappa \nabla u \cdot \nabla p \; dxdt$$
$$+ \int_0^T \int_\Omega \mathbf{v} \cdot \nabla u p \; dxdt + \int_\Omega (u(x,0) - m(x)) p_0 \; dxdt.$$

To begin, we will take the first variation of $\mathcal{L}$ with respect to $p, u \in \mathcal{V}$ and $p_0, m \in \mathcal{M}$.

3. State boundary condition: The first vartiation in the $\tilde{p_0}$ direction

$$\mathcal{L}_{p_0}(u,m,p,p_0)(\tilde{p_0}) = \int_\Omega (u(x,0) - m(x))\tilde{p_0} \; dxdt = 0, \quad \forall \tilde{p_0} \in \mathcal{M}$$

Recall that the variation vanishes at the stationary point for arbitrary $\tilde{p}_0$. Thus,

$$u(x,0) - m(x) = 0 \implies u(x,0) = m(x), \quad \forall x \in \Omega$$

4. Total state equation: Then we will take the first variations in the $\tilde{p}$ direction

$$\mathcal{L}_p(u,m,p,p_0)(\tilde{p}) = \int_0^T \int_\Omega u_t \tilde{p} \; dxdt + \int_0^T \int_\Omega \kappa \nabla u \cdot \nabla \tilde{p} \; dxdt + \int_0^T \int_\Omega \mathbf{v} \cdot \nabla u \tilde{p} \; dxdt$$
$$= \int_0^T \int_\Omega (u_t - \kappa \Delta u + \mathbf{v} \cdot \nabla u)\tilde{p} \; dxdt + \int_0^T \int_{\partial\Omega} \kappa \nabla u \cdot \mathbf{n} \tilde{p} \; dSdt = 0, \quad \forall \tilde{p} \in \mathcal{V}$$

This vanishes for arbitrary $\tilde{p}$. Thus, combining the information from both variations, we recover the strong form of the forward problem:

$$\begin{aligned} u_t - \kappa \Delta u + \mathbf{v} \cdot \nabla u &= 0 && \text{in } \Omega \times [0,T], \\ u(x,0) &= m(x) && \text{in } \Omega, \\ \kappa \nabla u \cdot \mathbf{n} &= 0 && \text{on } \partial\Omega \times [0,T]. \end{aligned}$$

5. The abjoint equation: All that is left is to take the first variation of $\mathcal{L}$ w.r.t u in the direction $\tilde{u}$.

$$\mathcal{L}_u(u,m,p,p_0)(\tilde{u}) = \int_{T_1}^T \int_{\Gamma_m} (u - u_d)\tilde{u} \; dxdt + \int_0^T \int_\Omega (\tilde{u}_t + \mathbf{v} \cdot \tilde{u}) p \; dxdt$$
$$+ \int_0^T \int_\Omega \kappa \nabla \tilde{u} \cdot \nabla p \; dxdt + \int_\Omega \tilde{u}(x,0) p_0(x) \; dx = 0, \quad \forall p_0 \in \mathcal{M}$$

To remove all the derivatives from $\tilde{u}$ we need to do integration by parts in time $\tilde{u}_t p$ and space $(\mathbf{v} \cdot \tilde{u})p = (\mathbf{v}p) \cdot \tilde{u}$ and $\kappa \nabla \tilde{u} \cdot \nabla p$ results in

$$\mathcal{L}_u(u,m,p,p_0)(\tilde{u}) = \int_{T_1}^T \int_{\Gamma_m} (u - u_d)\tilde{u} \; dxdt + \int_0^T \int_\Omega (-p_t - \nabla \cdot (\mathbf{v}p) - \kappa \Delta p)\tilde{u} \; dxdt$$
$$+ \int_\Omega \tilde{u}(x,T)p(x,T) + (p_0(x) - p(x,0))\tilde{u}(0) \; dx + \int_0^T \int_{\partial\Omega} (\mathbf{v}p + \kappa \nabla p) \cdot \mathbf{n} \; dxdt = 0, \quad \forall \tilde{u} \in \mathcal{M}$$

This vanishes for arbitrary $\tilde{u}$, we obtain

$$p_0(x) = p(x,0), \quad p(x,T) = 0, \quad \forall x \in \Omega$$

as well as the following strong form of the adjoint

$$\begin{aligned} -p_t - \nabla \cdot (\mathbf{v}p) - \kappa \Delta p &= 0 && \text{in } \Omega \times [0,T], \\ p(x,T) &= 0 && \text{in } \Omega, \\ (\mathbf{v}p + \kappa \nabla p) &= -(u - u_d) && \text{on } \Gamma_m \times [T_1,T], \\ (\mathbf{v}p + \kappa \nabla p) &= 0 && \text{on } \partial\Omega / \Gamma_m \times [T_1,T]. \end{aligned}$$

6. The Gradient: The system only gives p at time $t = T$ rather than at $t = 0$. With the final variation of the Lagrangian with respect to m in the direction of $\tilde{m}$ is

$$\mathcal{L}_m(u,m,p,p_0)(\tilde{m}) = \int_\Omega \gamma_1 m \tilde{m} + \gamma_2 \nabla m \cdot \nabla \tilde{m} - p_0 \tilde{m} \; dx = 0$$

We can define the strong form of gradient as follow after integration by parts,

$$G(m) = \begin{cases} \gamma_1 m - \gamma_2 \Delta m - p_0, & \text{in } \Omega \\ \gamma_2 \nabla m \cdot \mathbf{n}, & \text{on } \partial\Omega \end{cases}$$

Hessian-apply:

Next, we have to construct the Hessian-apply

1. The Hessian-apply:

$$\begin{aligned}\mathcal{L}^H(u,m,p,p_0,\hat{u},\hat{m},\hat{p},\hat{p}_0) = &\int_{T_1}^T\int_{\Gamma_m}(u-u_d)\hat{u}\;dxdt + \int_0^T\int_\Omega(\hat{u}_t+\mathbf{v}\cdot\hat{u})p\;dxdt + \int_0^T\int_\Omega \kappa\nabla\hat{u}\cdot\nabla p\;dxdt\\ &+\int_\Omega \gamma_1 m\hat{m}+\gamma_2\nabla m\cdot\nabla\hat{m}-p_0\hat{m}\;dx + \int_0^T\int_\Omega u_t\hat{p}\;dxdt + \int_0^T\int_\Omega\kappa\nabla u\cdot\nabla\hat{p}\;dxdt\\ &+\int_0^T\int_\Omega \mathbf{v}\cdot\nabla u\hat{p}\;dxdt + \int_\Omega(u(x,0)-m(x))\hat{p}_0\;dx + \int_\Omega\hat{u}(x,0)p_0(x)\;dx.\end{aligned}$$

2. Incremental (second-order) adjoint:

$$\begin{aligned}\mathcal{L}^H_u(u,m,p,p_0,\hat{u},\hat{m},\hat{p},\hat{p}_0)(\tilde{u}) = &\int_{T_1}^T\int_{\Gamma_m}\tilde{u}\hat{u}\;dxdt + \int_0^T\int_\Omega\tilde{u}_t\hat{p}\;dxdt + \int_0^T\int_\Omega\kappa\nabla\tilde{u}\cdot\nabla\hat{p}\;dxdt\\ &+\int_0^T\int_\Omega\mathbf{v}\cdot\nabla\tilde{u}\hat{p}\;dxdt + \int_\Omega\tilde{u}(x,0)\hat{p}_0(x)\;dxdt = 0,\quad\forall\tilde{u}\in\mathcal{V}.\end{aligned}$$

3. Incremental state:

$$\mathcal{L}^H_p(u,m,p,p_0,\hat{u},\hat{m},\hat{p},\hat{p}_0)(\tilde{p}) = \int_0^T\int_\Omega(\hat{u}_t+\mathbf{v}\cdot\nabla\hat{u})\tilde{p}\;dxdt + \int_0^T\int_\Omega\kappa\nabla\hat{u}\cdot\nabla\tilde{p}\;dxdt = 0,\quad\forall\tilde{p}\in\mathcal{V}.$$

4. Incremental state of $\hat{u}$ initial condition:

$$\mathcal{L}^H_{p_0}(u,m,p,p_0,\hat{u},\hat{m},\hat{p},\hat{p}_0)(\tilde{p}_0) = \int_\Omega(\hat{u}(x,0)-\hat{m}(x))\tilde{p}_0(x)\;dx = 0,\quad\forall\tilde{p}_0\in\mathcal{M}.$$

5. Hessian apply:

$$\mathcal{L}^H_m(u,m,p,p_0,\hat{u},\hat{m},\hat{p},\hat{p}_0)(\tilde{m}) = \int_\Omega\gamma_1\tilde{m}\hat{m}+\gamma_2\nabla\tilde{m}\cdot\nabla\hat{m}\;dx - \int_\Omega\tilde{m}(x)\hat{p}_0(x)\;dx = 0,\quad\forall\tilde{m}\in\mathcal{M}.$$

Discretize

Let,

$$\begin{aligned}&\hat{u}_h(x,t)\approx\sum_{j=1}^{n_s}\hat{u}_j(t)\phi_j(x),\quad \hat{p}_h(x,t)\approx\sum_{j=1}^{n_s}\hat{p}_j(t)\phi_j(x),\quad \hat{m}_h(x)\approx\sum_{j=1}^{n_s}\hat{m}_j\psi_j(x),\quad \hat{p_0}_h(x)\approx\sum_{j=1}^{n_s}\hat{p_0}_j\psi_j(x)\\ &\tilde{u_h}(x,t)\approx\sum_{j=1}^{n_s}\tilde{u}_j(t)\phi_j(x),\quad \tilde{p}_h(x,t)\approx\sum_{j=1}^{n_s}\tilde{p}_j(t)\psi_j(x),\quad \tilde{m}_h(x)\approx\sum_{j=1}^{n_s}\tilde{m}_j\psi_j(x).\end{aligned}$$

1. The incremental state of $\hat{u}$ initial condition: for all $\tilde{p}_0$

$$\int_\Omega(\hat{u}(x,0)-\hat{m}(x))\tilde{p}_0\;dx = 0 \quad\Longrightarrow\quad \boldsymbol{U}(0) = \underline{m}$$

   where

$$\boldsymbol{U}(t) = \begin{bmatrix}\hat{u}_1(t)\\ \hat{u}_2(t)\\ \vdots\\ \hat{u}_{n_s}(t)\end{bmatrix},\quad \underline{m} = \begin{bmatrix}\hat{m}_1\\ \hat{m}_2\\ \vdots\\ \hat{m}_{n_s}\end{bmatrix}\in\mathcal{R}^{n_s}$$

2. Discrete state increment: for all $\tilde{p}$, solve for $\hat{u}$

$$\mathcal{L}^H_p(u,m,p,p_0,\hat{u},\hat{m},\hat{p},\hat{p}_0)(\tilde{p}) = \int_0^T\int_\Omega(\hat{u}_t+\mathbf{v}\cdot\nabla\hat{u})\tilde{p}\;dxdt + \int_0^T\int_\Omega\kappa\nabla\hat{u}\cdot\nabla\tilde{p}\;dxdt = 0$$

   We will break this up into two parts,

(a)

$$\int_0^T \int_\Omega \hat{u}_t \tilde{p} \; dxdt = \sum_{i=1}^{n_s} \sum_{j=1}^{n_s} \int_0^T \dot{\hat{u}}_i(t) \tilde{p}(t) \; dt \int_\Omega \phi_i(x) \phi_j(x) \; dx$$
$$= \int_0^T \tilde{P}^T(t) M \dot{\boldsymbol{U}}(t) \; dt$$

(b)

$$\int_0^T \int_\Omega \mathbf{v} \cdot \nabla \hat{u} \tilde{p} + \kappa \nabla \hat{u} \cdot \nabla \tilde{p} \; dxdt = \sum_{i=1}^{n_s} \sum_{j=1}^{n_s} \int_0^T \hat{u}_i(t) \tilde{p}(t) \; dt \int_\Omega (\kappa \nabla \phi_i(x) \cdot \nabla \phi_j(x) \; + \mathbf{v} \cdot \nabla \phi_i(x) \phi_j(x) \;) dx$$
$$= \int_0^T \tilde{P}^T(t) N \boldsymbol{U}(t) \; dt$$

All together

$$\int_0^T \tilde{P}^T (M \dot{\boldsymbol{U}}(t) + N \boldsymbol{U}(t)) \; dt = 0, \quad \forall \tilde{P}$$

Thus,

$$M \dot{\boldsymbol{U}}(t) + N \boldsymbol{U}(t) = 0$$

where,

$$M, N \in \mathcal{R}^{n_s \times n_s}$$

- Recall: Backwards Euler (unconditionally stable for any size $\Delta t > 0$)

$$u_t = f(t, u)$$
$$u(x_i; t_{n+1}) - u(x_i; t_n) \approx \Delta t f(t_{n+1}, u(x_i; t_{n+1}))$$

By Backwards Euler and simplification,

$$M^T \boldsymbol{U}^{n+1} + \Delta t N \boldsymbol{U}^{n+1} = M \boldsymbol{U}^n$$

Let,

$$A = M + \Delta t N$$

Hence,

$$A \boldsymbol{U}^{n+1} = M \boldsymbol{U}^n, \quad 0 < n < n_t \tag{8}$$

With equation (8) and the initial conditions $\boldsymbol{U}^0 = u(0)$ we can get the following system,

$$S \boldsymbol{U} = F$$

where

$$S = \begin{bmatrix} M & & & \\ -M & A & & \\ & \ddots & \ddots & \\ & & -M & A \end{bmatrix} \in \mathcal{R}^{n_s(n_t+1) \times n_s(n_t+1)}, \quad F = \begin{bmatrix} M\underline{m} \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \boldsymbol{U} = \begin{bmatrix} \boldsymbol{U}^0 \\ \boldsymbol{U}^1 \\ \vdots \\ \boldsymbol{U}^{n_t} \end{bmatrix} \in \mathcal{R}^{n_s(n_t+1)}$$

and $\boldsymbol{U}^i = \boldsymbol{U}(i\Delta t)$

3. Discrete Adjoint increment: for all $\tilde{u}$, solve for $\hat{p}$

$$\mathcal{L}_u^H(u, m, p, \hat{u}, \hat{m}, \hat{p}, \hat{p}_0)(\tilde{u}) = \int_{T_1} \int_{\Gamma_m} \tilde{u} \hat{u} \; dxdt + \int_0^T \int_\Omega \tilde{u}_t \hat{p} \; dxdt + \int_0^T \int_\Omega \kappa \nabla \tilde{u} \cdot \nabla \hat{p} \; dxdt$$
$$+ \int_0^T \int_\Omega \mathbf{v} \cdot \nabla \tilde{u} \hat{p} \; dxdt + \int_\Omega \tilde{u}(x, 0) \hat{p}_0(x) \; dxdt = 0,$$

Before we start discritzing, we need to move the time derivative to our unknown $\hat{p}(x,t)$,

$$\int_0^T \int_\Omega \tilde{u}_t(x,t)\hat{p}(x,t)\ dxdt = \int_\Omega \tilde{u}(x,T)\hat{p}(x,t) - \tilde{u}(x,0)\hat{p}(x,0)\ dx - \int_0^T \int_\Omega \tilde{u}(x,t)\hat{p}_t(x,t)\ dxdt$$

$$\begin{aligned}\mathcal{L}_u^H(u,m,p,\hat{u},\hat{m},\hat{p},\hat{p}_0)(\tilde{u}) &= \int_{T_1}^T \int_{\Gamma_m} \tilde{u}\hat{u}\ dxdt + \int_\Omega \tilde{u}(x,T)\hat{p}(x,t)\ dx - \int_0^T \int_\Omega \tilde{u}(x,t)\hat{p}_t(x,t)\ dxdt \\ &+ \int_0^T \int_\Omega \kappa\nabla\tilde{u}\cdot\nabla\hat{p}\ dxdt + \int_0^T \int_\Omega \mathbf{v}\cdot\nabla\tilde{u}\hat{p}\ dxdt + \int_\Omega \tilde{u}(x,0)(\hat{p}_0(x) - \hat{p}(x,0))\ dx \\ &= 0\end{aligned}$$

At the stationary point the variation is zero in the domain. Thus we get the following conditions

$$\hat{p}_0(x) = \hat{p}(x,0), \quad \hat{p}(x,T) = 0, \quad \forall x \in \Omega$$

Thus,

$$\begin{aligned}\mathcal{L}_u^H(u,m,p,\hat{u},\hat{m},\hat{p},\hat{p}_0)(\tilde{u}) &= \int_{T_1}^T \int_{\Gamma_m} \tilde{u}\hat{u}\ dxdt - \int_0^T \int_\Omega \tilde{u}(x,t)\hat{p}_t(x,t)\ dxdt \\ &+ \int_0^T \int_\Omega \mathbf{v}\cdot\nabla\tilde{u}\hat{p}\ dxdt + \int_0^T \int_\Omega \kappa\nabla\tilde{u}\cdot\nabla\hat{p}\ dxdt \\ &= 0\end{aligned}$$

Now, we will discritize the Lagrangian in pieces,

$$\begin{aligned}\int_{T_1}^T \int_{\Gamma_m} \tilde{u}\hat{u}\ dxdt &= \sum_{i=1}^{n_s}\sum_{j=1}^{n_s} \int_{T_1}^T \tilde{u}_i(t)\hat{u}(t)\ dxdt \int_{\Gamma_m} \phi_i(x)\phi_j(x)\ dx \\ &= \int_{T_1}^T \tilde{U}^T(t) M^\Gamma \boldsymbol{U}(t)\ dt\end{aligned}$$

$$\begin{aligned}\int_0^T \int_\Omega \tilde{u}(t)\hat{p}_t(t)\ dxdt &= \sum_{i=1}^{n_s}\sum_{j=1}^{n_s} \int_0^T \tilde{u}_i(t)\dot{\hat{p}}(t)\ dt \int_\Omega \phi_i(x)\phi_j(x)\ dx \\ &= \int_0^T \tilde{U}^T(t) M^T \dot{\boldsymbol{P}}(t)\ dt\end{aligned}$$

$$\begin{aligned}\int_0^T \int_\Omega \kappa\nabla\tilde{u}\cdot\nabla\hat{p} + \mathbf{v}\cdot\nabla\tilde{u}\hat{p}\ dxdt &= \sum_{i=1}^{n_s}\sum_{j=1}^{n_s} \int_0^T \tilde{u}_i(t)\hat{p}(t)\ dt \int_\Omega (\kappa\nabla\phi_i(x)\cdot\nabla\phi_j(x)\ + \mathbf{v}\cdot\nabla\phi_i(x)\phi_j(x)\ )dx \\ &= \int_0^T \tilde{U}^T(t) N^T \boldsymbol{P}(t)\ dt\end{aligned}$$

All together,

$$\int_{T_1}^T \tilde{U}^T(t)(M^\Gamma \boldsymbol{U}(t) - M^T\dot{\boldsymbol{U}}(t) + N^T\boldsymbol{U}(t))\ dt = 0, \quad \forall \tilde{U}(t) \in \mathcal{R}^{n_s}$$

Thus,

$$M^\Gamma \boldsymbol{U}(t) - M^T\dot{\boldsymbol{P}}(t) + N^T\boldsymbol{P}(t) = 0 \tag{9}$$

where,

$$\boldsymbol{P}(t) = \begin{bmatrix} \hat{p}_1(t) \\ \hat{p}_2(t) \\ \vdots \\ \hat{p}_{n_s}(t) \end{bmatrix} \in \mathcal{R}^{n_s}$$

and $\boldsymbol{P}^i = \boldsymbol{P}(i\Delta T)$. By Backwards Euler and simplifications,

$$\begin{aligned} M^\Gamma \boldsymbol{U}^{n+1} - M^T \frac{\boldsymbol{P}^{n+1} - \boldsymbol{P}^n}{\Delta t} + N^T \boldsymbol{P}^{n+1} &= 0 \\ \Delta t M^\Gamma \boldsymbol{U}^{n+1} - M^T \boldsymbol{P}^{n+1} + M^T \boldsymbol{P}^n + \Delta t N^T \boldsymbol{P}^{n+1} &= 0 \\ \implies \hat{A}\boldsymbol{P}^{n+1} &= -\Delta t M^\Gamma \boldsymbol{U}^n - M^T \boldsymbol{P}^n, \quad 0 < n < n_t \end{aligned}$$

Where,

$$\hat{A} = -M^T + \Delta t N^T,$$

The adjoint equation can also be written as the matrix system

$$\hat{S}\boldsymbol{P} = -\hat{Q}\boldsymbol{U} + \hat{F}$$

where

$$\hat{Q} = \begin{bmatrix} 0 & & & & & & \\ & \ddots & & & & & \\ & & 0 & & & & \\ & & & \Delta t M^\Gamma & & & \\ & & & & \ddots & & \\ & & & & & \Delta t M^\Gamma & \\ & & & & & & 0 \end{bmatrix} \in \mathcal{R}^{n_s(n_t+1)\times n_s(n_t+1)}$$

$$\hat{S} = \begin{bmatrix} M & & & & \\ M^T & \hat{A} & & & \\ & \ddots & \ddots & & \\ & & M^T & \hat{A} & \\ & & & & \hat{A} \end{bmatrix} \in \mathcal{R}^{n_s(n_t+1)\times n_s(n_t+1)}, \quad \hat{F} = \begin{bmatrix} M\boldsymbol{P}(0) \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \boldsymbol{P} = \begin{bmatrix} \boldsymbol{P}^0 \\ \boldsymbol{P}^1 \\ \vdots \\ \boldsymbol{P}^{n_t} \end{bmatrix} \in \mathcal{R}^{n_s(n_t+1)}$$

Note, $\hat{Q} = 0$ for times $t \leq T_1$ and at $t = T$.

4. Discritize adjoint equation: for all $\tilde{u}$, solve for p

$$\int_{T_1}^T \int_{\Gamma_m} (u - u_d)\tilde{u}\ dxdt + \int_0^T \int_\Omega (\tilde{u}_t + \mathbf{v}\cdot\tilde{u})p\ dxdt + \int_0^T \int_\Omega \kappa \nabla\tilde{u}\cdot\nabla p\ dxdt + \int_\Omega \tilde{u}(x,0)p_0(x)\ dx = 0$$

by similar argument of the incremental adjoint equation, we get the following,

$$\hat{S}P = -\hat{Q}(U - U_d) + F^*$$

where,

$$P(t) = \begin{bmatrix} p_1(t) \\ p_2(t) \\ \vdots \\ p_{n_s}(t) \end{bmatrix} \in \mathcal{R}^{n_s}, \quad P = \begin{bmatrix} P^0 \\ P^1 \\ \vdots \\ P^{n_t} \end{bmatrix}, \quad F^* = \begin{bmatrix} MP(0) \\ 0 \\ \vdots \\ 0 \end{bmatrix} \in \mathcal{R}^{n_s(n_t+1)}$$

and $P^i = P(i\Delta t)$.

5. All that is left is the hessian for all $\tilde{m}$ solve for $\hat{m}$.

$$\int_\Omega \gamma_1 \tilde{m}\hat{m} + \gamma_2 \nabla\tilde{m}\cdot\nabla\hat{m}\ dx - \int_\Omega \tilde{m}\hat{p}_0\ dx = 0$$

As before, we will break up the discration into part,

(a)

$$\begin{aligned}\int_\Omega \gamma_1 \tilde{m}\hat{m}\ dx &= \gamma_1 \sum_{j=1}^{n_s}\sum_{i=1}^{n_s} \tilde{m}_j \hat{m}_j \int_\Omega \psi_j(x)\psi_i(x) \\ &= \gamma_1 \underline{\tilde{m}} M^m \underline{\hat{m}}\end{aligned}$$

(b)

$$\begin{aligned}\int_\Omega \gamma_2 \nabla\tilde{m}\cdot\nabla\hat{m}\ dx &= \gamma_2 \sum_{j=1}^{n_s}\sum_{i=1}^{n_s} \tilde{m}_j \hat{m}_j \int_\Omega \nabla\psi_j(x)\cdot\nabla\psi_i(x) \\ &= \gamma_2 \underline{\tilde{m}} R^m \underline{\hat{m}}\end{aligned}$$

(c)

$$\begin{aligned}\int_\Omega \tilde{m}\hat{p}_0\ dx &= \sum_{j=1}^{n_s}\sum_{i=1}^{n_s} \tilde{m}_j \hat{p_0}_j \int_\Omega \psi_j(x)\cdot\psi_i(x) \\ &= \underline{\tilde{m}} M^m \underline{\hat{p}_0}\end{aligned}$$

where,

$$\underline{\tilde{m}} = \begin{bmatrix}\tilde{m}_1 \\ \tilde{m}_2 \\ \vdots \\ \tilde{m}_{n_s}\end{bmatrix}, \quad \underline{\hat{m}} = \begin{bmatrix}\hat{m}_1 \\ \hat{m}_2 \\ \vdots \\ \hat{m}_{n_s}\end{bmatrix}, \quad \underline{\hat{p}_0} = \begin{bmatrix}(\hat{p}_0)_1 \\ (\hat{p}_0)_2 \\ \vdots \\ (\hat{p}_0)_{n_s}\end{bmatrix} \in \mathcal{R}^{n_s}$$

All together,

$$\underline{\tilde{m}}(\gamma_1 M^m \underline{\hat{m}} + \gamma_2 R^m \underline{\hat{m}} - M^m \underline{\hat{p}_0}) = 0 \quad \Longrightarrow \quad \gamma_1 M^m \underline{\hat{m}} + \gamma_2 R^m \underline{\hat{m}} - M^m \underline{\hat{p}_0} = H\underline{\hat{m}}$$

6. Discrete gradient for all $\tilde{m}$ solve for m.

$$\mathcal{L}_m(u,m,p,p_0)(\tilde{m}) = \int_\Omega \gamma_1 m\tilde{m} + \gamma_2 \nabla m\cdot\nabla\tilde{m} - p_0\tilde{m}\ dx = 0$$

By similar arguments as the hessian we get,

$$\gamma_1 M^m \underline{m} + \gamma_2 R^m \underline{m} - M^m \underline{p_0} = G(\underline{m})$$

Discrete systems

Now we can put all the equations together. We will start with the incremental state equation,

- n = 0; $\quad A\boldsymbol{U}^1 = M\boldsymbol{U}^0$
- n = 1; $\quad A\boldsymbol{U}^2 = M\boldsymbol{U}^1$

  $\vdots$
- $n = N_t - 1$; $\quad A\boldsymbol{U}^{N_t} = M\boldsymbol{U}^{N_t-1}$

We can also define it as a large matrix system

$$S\boldsymbol{U} = F$$

Where,

$$S = \begin{bmatrix} M & & & \\ -M & A & & \\ & \ddots & \ddots & \\ & & -M & A \end{bmatrix} \in \mathcal{R}^{n_s(n_t+1)\times n_s(n_t+1)}, \quad F = \begin{bmatrix} M\boldsymbol{U}(0) \\ 0 \\ \vdots \\ 0 \end{bmatrix} \in \mathcal{R}^{n_s(n_t+1)}$$

Thus, the discrite incremental adjoint can be defined by the following

$$\begin{aligned} \hat{A}\boldsymbol{P}^{i+1} &= -Q\boldsymbol{U}^i + M^T\boldsymbol{P}^i \quad \text{for } i = n_t - 1, ..., 1 \\ \boldsymbol{P}^{n_t} &= 0 \end{aligned}$$

Which we saw it can be written as,

$$\hat{S}\boldsymbol{P} = -\hat{Q}\boldsymbol{U} + \hat{F}$$

Lastly, the Hessian is defined by

$$\gamma_1 M^m \underline{\hat{m}} + \gamma_2 R^m \underline{\hat{m}} - M^m \underline{\hat{p}_0} = H\underline{\hat{m}}$$

Newton system

Thus, the Newton system can be defined by,

$$H\underline{\hat{m}} = -\mathcal{L}_m(U(0), 0, P) \iff G(m) = 0$$

Hence, out newton system is defined by,

$$\gamma_1 M^m \underline{\hat{m}} + \gamma_2 R^m \underline{\hat{m}} - M^m \underline{\hat{p}}_0 = M^m \underline{p}_0$$

where $\underline{\hat{p}}_0$ solves,

$$\begin{aligned} S^T\boldsymbol{P} &= -\hat{Q}\boldsymbol{U} + \hat{F} \\ S\boldsymbol{U} &= F \end{aligned}$$

and $\underline{p}_0$ solves,

$$\begin{aligned} \hat{S}P &= \hat{Q}U_d + F^* \\ U &= 0 \text{ since } m = 0 \end{aligned}$$

Discretize then optimize

To do this approach, we will find following; FEM discretization of the state equation, Discretize the cost function, find Lagrangian and Hessian applied. Lastly, find the adjoint equation,the gradient, state incremental equation, adjoint incremental equation

FEM discretization

Discretize equation (2) by scaling by $\tilde{u} \in \mathcal{V}$ and integrating over the spacial temporal domain

$$\int_0^T \int_\Omega u_t\tilde{u} - k\Delta u\tilde{u} + \mathbf{v} \cdot u\tilde{u} \; dxdt = 0$$

Let,

$$u_h(x,t) \approx \sum_{j=1}^{n_s} u_j(t)\phi_j(x), \qquad \tilde{u}(x,t) = \tilde{u}_i(t)\phi_i(x)$$

1. Time derivative term

$$\int_0^T \int_\Omega u_t\tilde{u} \; dxdt = \int_0^T \sum_{j=1}^{n_s} \dot{u}_j(t)\tilde{u}_i(t) \int_\Omega \phi_i(x)\phi_j(x)dt = \int_0^T \tilde{U}^T(t)M\dot{U}(t) \; dt$$

2. Advection diffusion term

$$\int_0^T \int_\Omega -k\Delta u\tilde{u} + \mathbf{v} \cdot \nabla u\tilde{u} \; dxdt = \int_0^T \sum_{i=1}^{n_s}\sum_{j=1}^{n_s} u_j(t)\tilde{u}_i(t) \int_\Omega (-k\Delta\phi_j(x)\phi_i(x) + \mathbf{v} \cdot \nabla\phi_j(x)\phi_i(x)) = \int_0^T \tilde{U}^T(t)NU(t)$$

All together,

$$\int_0^T \tilde{U}(t)^T(M\dot{U}(t) + NU(t)) = 0, \quad \forall \tilde{U}(t) \quad \implies M\dot{U}(t) + NU(t) = 0$$

By implicit Backwards Euler,

$$M\frac{U^{n+1} - U^n}{\Delta t} + NU^{n+1} = 0 \quad \implies MU^{n+1} + \Delta t NU^{n+1} = MU^n$$

Let,

$$A = M + \Delta t N$$

Note: $u(x,0) = m(x) \implies U^0 = m \in \mathcal{R}^{n_s}$ Thus we get the following,

- n = 0; $AU^1 = MU^0$
- n = 1; $AU^2 = MU^1$
  $\vdots$
- $n = N_t - 1$; $AU^{N_t} = MU^{N_t - 1}$

Which can also be defined as a matrix system,

$$\begin{bmatrix} M & & & \\ -M & A & & \\ & \ddots & \ddots & \\ & & -M & A \end{bmatrix} \begin{bmatrix} U^0 \\ U^1 \\ \vdots \\ U^{N_t} \end{bmatrix} = \begin{bmatrix} Mm \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Let

$$S = \begin{bmatrix} M & & & \\ -M & A & & \\ & \ddots & \ddots & \\ & & -M & A \end{bmatrix} \in \mathcal{R}^{n_s(N_t+1)\times n_s(N_t+1)},\ U = \begin{bmatrix} U^0 \\ U^1 \\ \vdots \\ U^{N_t} \end{bmatrix} \in \mathcal{R}^{n_s(N_t+1)},\ F = \begin{bmatrix} Mm \\ 0 \\ \vdots \\ 0 \end{bmatrix} \in \mathcal{R}^{n_s(N_t+1)},$$

so,

$$SU = F$$

Discretization of cost function

$$\min_m J(m) = \frac{1}{2}\int_{T_1}^{T}\int_{\Gamma_m} (u - u_d)^2 dxdt + \frac{\gamma_1}{2}\int_\Omega m(x)^2 dx + \frac{\gamma_2}{2}\int_\Omega |\nabla m(x)|^2 dx$$

$$(u + u_d)_h(x,t) \approx \sum_{j=1}^{n_s}(u + u_d)_j(t)\phi_j(x), \qquad m_h \approx \sum_{j=1}^{n_s} m_j\psi_j(x)$$

We will break up the cost function into three integrals

1. $$\frac{1}{2}\int_{T_1}^{T}\int_{\Gamma_m}(u - u_d)^2 dxdt = \frac{1}{2}\sum_{j=1}^{n_s}\sum_{i=1}^{n_s}\int_{T_1}^{T}(u - u_d)_i(t)(u - u_d)_j(t)\int_{\Gamma_m}\phi_i(x)\phi_j(x)dxdt = \frac{1}{2}(U - U_d)^T Q(U - U_d)$$

2. $$\frac{1}{2}\int_\Omega m(x)^2 dx = \frac{1}{2}\sum_{j=1}^{n_s}\sum_{i=1}^{n_s} m_i m_j \int_{\Gamma_m}\psi_i(x)\psi_j(x) = \frac{\gamma_1}{2}m^T M^m m$$

3. $$\frac{1}{2}\int_\Omega |\nabla m(x)|^2 dx = \frac{1}{2}\sum_{j=1}^{n_s}\sum_{i=1}^{n_s} m_i m_j \int_{\Gamma_m}\nabla\psi_i(x)\cdot\nabla\psi_j(x) = \frac{\gamma_2}{2}m^T R^m m$$

All together, we get the optimization problem in finite dimension

$$\min_{m\in\mathcal{R}^{n_s}} J_h(m) = \frac{\gamma_1}{2}m^T M^m m + \frac{\gamma_2}{2}m^T R^m m + \frac{1}{2}(U-U_d)^T Q(U-U_d)$$

where, U solves $SU = F$ and

$$Q = \begin{bmatrix} 0 & & & & & \\ & \ddots & & & & \\ & & 0 & & & \\ & & & \Delta t M^\Gamma & & \\ & & & & \ddots & \\ & & & & & \Delta t M^\Gamma \end{bmatrix} \in \mathcal{R}^{n_s(n_t+1)\times n_s(n_t+1)}$$

The Lagrangian

Now, we have everything we need to create the Lagrangian

$$\mathcal{L}(U,m,P) = \frac{\gamma_1}{2}m^T Mm + \frac{\gamma_2}{2}m^T Rm + \frac{1}{2}(U-U_d)^T Q(U-U_d) + P^T(SU-F).$$

We will only take the variation of with respect to U, m.

1. The state equation:

$$\mathcal{L}_P(U,m,P) = SU - F = 0 \quad \implies SU = F.$$

2. The adjoint equation:

$$\mathcal{L}_U(U,m,P) = Q(U-U_d) + S^T P = 0 \quad \implies S^T P = -Q(U-U_d).$$

3. The gradient:

$$\mathcal{L}_m(U,m,P) = \gamma_1 M^m m + \gamma_2 R^m m - MP^0 = 0 \quad \implies G(m) = \gamma_1 M^m m + \gamma_2 R^m m - MP^0.$$

Note we can solve the adjoint backward in time:

$$\begin{bmatrix} M & -M & & & \\ & A^T & -M & & \\ & & \ddots & \ddots & \\ & & & & -M \\ & & & & A^T \end{bmatrix} \begin{bmatrix} P^0 \\ P^1 \\ \vdots \\ P^{N_t} \end{bmatrix} = - \begin{bmatrix} 0 \\ 0 \\ \Delta t M^\Gamma (U^i - U_d^i) \\ \vdots \end{bmatrix}$$

Where $i = n_t - 1, \ldots, im$, and im is the first instance where we evaluate the misfit. We can also write the matrix system as follows

$$\begin{aligned} A^T P^{n_t} &= -\Delta t M^\Gamma (U^{n_t} - U_d^{n_t}) \\ A^T P^i &= -\Delta t M^\Gamma (U^i - U_d^i) + MP^{i+1} \quad , i = n_t - 1, \ldots, im \\ A^T P^i &= MP^{i+1} \quad , i = im - 1, \ldots, 1 \\ M^T P^0 - M^T P^1 = 0 &\implies (P^0 = P^1) \end{aligned}$$

Hessian-apply

Now, we have to construct the Hessian applied,

1. Hessian-apply

$$\mathcal{L}^H(U,m,P,\boldsymbol{U},\hat{m},\boldsymbol{P}) = \hat{m}^T(\gamma_1 Mm + \gamma_2 Rm - MP^0) + \boldsymbol{U}^T(S^T P + Q(U-U_d)) + \boldsymbol{P}(SU-F)$$

2. incremental state:

$$\mathcal{L}_P^H(U,m,P,\boldsymbol{U},\hat{m},\boldsymbol{P}) = S\boldsymbol{U} - F = 0 \implies S\boldsymbol{U} = F$$

3. incremental adjoint:

$$\mathcal{L}_U^H(U, m, P, \boldsymbol{U}, \hat{m}, \boldsymbol{P}) = S^T\boldsymbol{P} + Q\boldsymbol{U} - \hat{F} = 0 \implies S^T\boldsymbol{P} = -Q\boldsymbol{U} + \hat{F}$$

4. Hessian:

$$\mathcal{L}_m^H(U, m, P, \boldsymbol{U}, \hat{m}, \boldsymbol{P}) = \gamma_1 M^m\hat{m} + \gamma_2 R^m\hat{m} - M^m\boldsymbol{P}^0 = 0 \implies H\hat{m} = \gamma_1 M^m\hat{m} + \gamma_2 R^m\hat{m} - M^m\boldsymbol{P}^0$$

Newton system

Thus, the Newton system can be defined by,

$$H\hat{m} = -\mathcal{L}_m(U(0), 0, P) \iff \mathcal{L}_m(U, m, P) = 0$$

Hence, out newton system is defined by,

$$\gamma_1 M^m\hat{m} + \gamma_2 R^m\hat{m} - M^m\boldsymbol{P}^0 = M^m P^0$$

where $\boldsymbol{P}^0$ solves,

$$\begin{aligned} S^T\boldsymbol{P} &= -Q\boldsymbol{U} + \hat{F} \\ S\boldsymbol{U} &= F \end{aligned}$$

and P^0 solves,

$$\begin{aligned} S^T P &= QU_d \\ U = 0 \text{ since } m &= 0 \end{aligned}$$

Note, to solve this problem we can do one newton step to solve our newton system. Our Newton system will converge in one newton step, since the Newton system is linear. The proof to show that quadratic optimization problem, only need one Newton step for convergence is in the appendix.

Conclusion

In this section, we will show the two different ways we can conclude that OTD and DTO do not commute for this time-dependent PDE.

Conclusion I

The Newton system for both OTD and DTO can now be compared.

1. The Newton system for OTD:

$$\gamma_1 M^m\underline{\hat{m}} + \gamma_2 R^m\underline{\hat{m}} - M^m\underline{\hat{p}}_0 = M^m\underline{p}_0$$

where $\underline{\hat{p}}_0$ solves,

$$\begin{aligned} \hat{S}\boldsymbol{P} &= -\hat{Q}\boldsymbol{U} + \hat{F} \\ S\boldsymbol{U} &= F \end{aligned}$$

and $\underline{p}_0$ solves,

$$\begin{aligned} \hat{S}P &= \hat{Q}U_d + F^* \\ U = 0 \text{ since } m &= 0 \end{aligned}$$

2. The Newton system for DTO:

$$\gamma_1 M^m\hat{m} + \gamma_2 R^m\hat{m} - M^m\boldsymbol{P}^0 = M^m P^0$$

where $\hat{P}_0$ solves,

$$\begin{aligned} S^T\boldsymbol{P} &= -Q\boldsymbol{U} + \hat{F} \\ S\boldsymbol{U} &= F \end{aligned}$$

and P_0 solves,

$$\begin{aligned} S^T P &= QU_d \\ U = 0 \text{ since } m &= 0 \end{aligned}$$

Note that $\underline{\hat{m}} = \hat{m}$, and $\underline{\hat{p}}_0$, $\underline{p}_0$ correspond to the same vector as $\boldsymbol{P}^0, P^0$ respectively.

From the two Newton systems, we can see that both adjoint and incremental adjoint do not commute, since $S^T \neq \hat{S}$, $Q \neq \hat{Q}$ and F^* is not the zero vector. The difference between the two method happens because of their derivation. For OTD, to get the adjoint and incremental adjoint, we had to do integration by parts which fixed both of them at $t = 0, T$ which modified Q and made F^* a nonzero vector. Due to the negative $\hat{A} \neq A$. Hence, we can conclude that OTD and DTO do not commute for this time dependent problem.

Conclusion II

We can conclude that the methods do not commute by showing that the two gradients are not equal.

1. The OTD gradient:

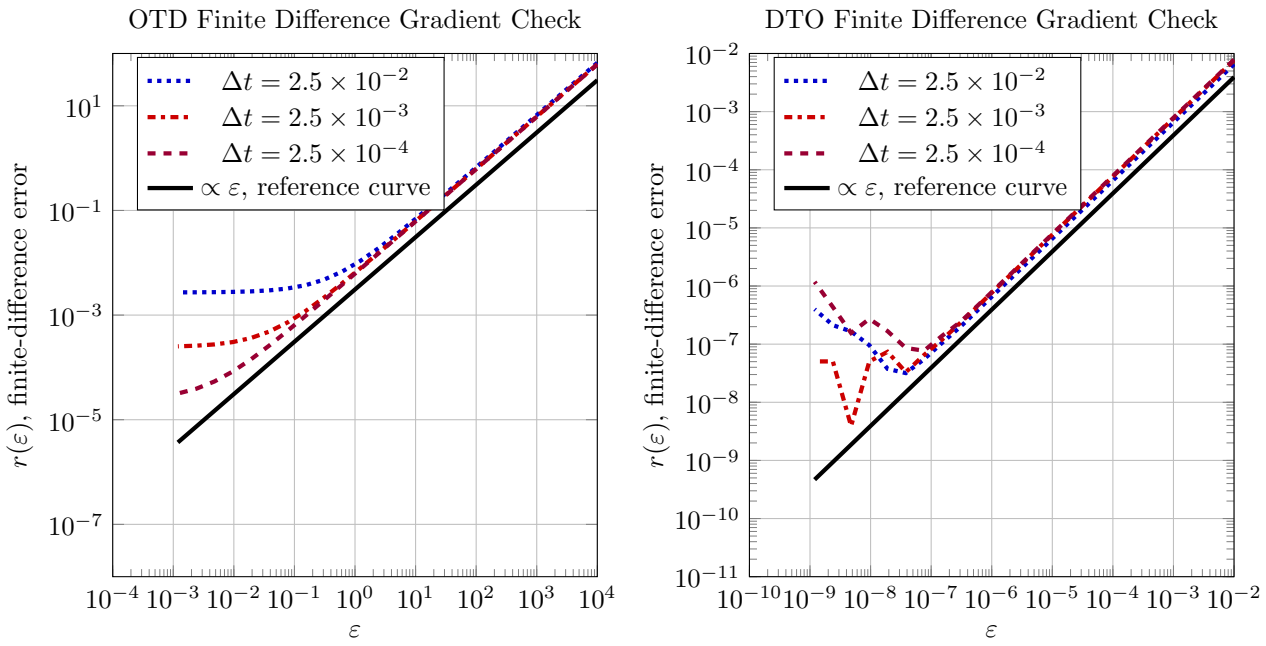
$$G(\underline{m}) = \gamma_1 M^m \underline{m} + \gamma_2 R^m \underline{m} - M^m \underline{p}_0, \quad m, \underline{p}_0 \in \mathcal{R}^{ns}.$$

2. The DTO gradient:

$$G(m) = \gamma_1 M^m m + \gamma_2 R^m m - M P^0, \quad m, P^0 \in \mathcal{R}^{ns}.$$

In OTD, we defined $\hat{p}_0(x) = \hat{p}(x, 0)$ for all x in Ω. However in DTO, we defined $P^0 = P^1$. Hence, we can conclude that OTD and DTO do not commute for this time dependent problem. We can also use this argument to show that OTD and DTO do not commute in the previous section.

We can also see this discrepancy by visualizing the finite difference gradient check for both OTD and DTO for different time-steps Δt.



The figures clearly show that the gradient via OTD is not correct. As you decrease the time-step Δt, the adjoint-based gradient gets closer to the finite difference gradient but even for the smallest Δt, we see that the error is quite larger compared to the DTO approach. Hence, DTO and OTD do not commute.

Acknowledgements

Thank you to Tucker Hartland for taking time out of your day to help me fix my misconceptions and for creating the codes that I needed to collect all my data. Lastly, thank you Noemi Petra for taking time out of your day to go through my work and checking if I was going in the correct direction with my computations.

A Appendix

A.1 Adjoint system with Forward Euler

Here we want to see what would have happened if we used a Forwards Euler approximation on equation (9). We get the following,

$$\begin{aligned} M^{\Gamma}\hat{U}^n - M^T\frac{\hat{P}^{n+1}-\hat{P}^n}{\Delta t} + N^T\hat{P}^n = 0 \\ \implies A\hat{P}^n = -\Delta t M^{\gamma} + M^T\hat{P}^{n+1} \end{aligned}$$

which can be written as the following matrix system,

$$\begin{bmatrix} M & & & & \\ & A^T & -M & & \\ & & \ddots & \ddots & \\ & & & & -M \\ & & & & A^T \end{bmatrix} \begin{bmatrix} P^0 \\ P^1 \\ \vdots \\ P^{N_t} \end{bmatrix} = - \begin{bmatrix} 0 \\ 0 \\ \Delta t M^{\Gamma}(U^i - U_d^i) \\ \vdots \\ 0 \end{bmatrix} + \begin{bmatrix} MP(0) \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

recall, M is symmetric.
This system matchs the adjoint system in the DTO case away from the boundary. However, due to the approximation we know that we will have convergence issues.

A.2 Newton one step proof

The goal here is to show that for a quadratic optimazation problem Newton method converges in one iteration. Let,

$$\min_m \mathcal{J}(m) := \frac{1}{2}(Fm-d)^T(Fm-d) \tag{10}$$

we know;

$$\begin{aligned} \nabla\mathcal{J}(m) &= F^TFm - F^Td. \\ \nabla^2\mathcal{J}(m) &= F^TF, \quad \text{assumming s.p.d} \end{aligned}$$

Suppose we start with m_0. Recall, Newton's method, minimizes a quadratic $\mathcal{L}(\hat{m})$ around m_0, that is:

$$\mathcal{J}(m_0+\hat{m}) \approx \mathcal{J}(m_0) + \hat{m}\nabla\mathcal{J}(m_0) + \hat{m}^T\nabla\mathcal{J}(m_0)\hat{m} := \mathcal{L}(\hat{m})$$

hence we look for $\hat{m}$ such that $\nabla\mathcal{L}(\hat{m}) = 0$, that is $\hat{m}$ solves,

$$\nabla^2\mathcal{J}(m_0)\hat{m} = -\nabla\mathcal{J}(m_0)$$

then, the next iteration $m_1 = m_0 + \hat{m}$.
For the quadratic problem equation 10:

$$\begin{aligned} F^TF\hat{m} = -F^TFm_0 + F^Td. \implies \\ \hat{m} = -m_0 + (F^TF)^{-1}F^Td = -m_0 + F^{-1}d. \end{aligned}$$

Now we need to check if $\nabla\mathcal{J}(m_0+\hat{m}) = 0$.

$$\begin{aligned} \nabla\mathcal{J}(m_0+\hat{m}) &= F^TF(m_0+\hat{m}) - F^Td \\ &= F^TFm_0 + F^TF(-m_0 + F^{-1}d) - F^Td \\ &= F^TF(m_0-m_0) + F^Td - F^Td \\ &= 0 \end{aligned}$$

Thus, $m_0+\hat{m}$ is the optimal solution. Hence, Newton method converged in one step.

B Codes

B.1 OTD code to collect data

```
import fenics as dl
import numpy as np
import matplotlib.pyplot as plt
from hippylib import nb
import ufl

plt.style.use('classic')
plt.rcParams.update({'font.size': 16})

mesh = dl.Mesh("ad_20.xml")
mesh = dl.refine(mesh)

dl.plot(mesh)
plt.show()

Vh = dl.FunctionSpace(mesh, "CG", 1)

v = dl.Constant((0.5, -0.25)) # advection field
kappa = 0.05                  # diffusion constant
T  = 0.25                     # final time
T1 = 0.2

sig = 0.4

mtrue = dl.interpolate( dl.Expression('std::min(0.5,std::exp(-100*(std::pow(x[0]-0.35,2)
            +  std::pow(x[1]-0.7,2))))', element=Vh.ufl_element()), Vh)

nb.plot(mtrue, mytitle=r"$m_{true}(x)$")
plt.show()

class reducedHessian:
    def __init__(me, Vh, simulation_times, kappa, v, T1, gamma):
        me.nt = len(simulation_times)
        me.dt = simulation_times[1] - simulation_times[0]
        me.Vh = Vh
        me.simulation_times = simulation_times
        me.T1 = T1
        me.gamma = gamma
        me.kappa = kappa
        me.v     = v
        utest = dl.TestFunction(Vh)
        utrial = dl.TrialFunction(Vh)
        Kform  = kappa*dl.inner(dl.grad(utest), dl.grad(utrial))*dl.dx(me.Vh.mesh())
        Mform  = utest*utrial*dl.dx(me.Vh.mesh())
        Bform  = utest*dl.inner(me.v, dl.grad(utrial))*dl.dx(me.Vh.mesh())
        BTform = utrial*dl.inner(me.v, dl.grad(utest))*dl.dx(me.Vh.mesh())

        me.K = dl.assemble(Kform)
        me.M = dl.assemble(Mform)
        me.B = dl.assemble(Bform)
        me.BT = dl.assemble(BTform)


        me.A     = me.M + me.dt*(me.K + me.B)
        me.A_adj = me.M + me.dt*(me.K + me.BT)
        me.ud = None
    def solve_fwd(me, m):
```

```
        u = [dl.Function(me.Vh) for i in range(me.nt)]
        u[0].assign(m)
        solver = dl.LUSolver()
        solver.set_operator(me.A)
        rhs = dl.Vector()
        me.A.init_vector(rhs, 0)
        for i in range(1, me.nt):
            rhs = me.M * u[i-1].vector()
            solver.solve(u[i].vector(), rhs)
        return u
    def solve_adj(me, u):
        p = [dl.Function(me.Vh) for i in range(me.nt)]
        p[-1].vector().zero()
        rhs1 = dl.Vector()
        rhs2 = dl.Vector()
        me.M.init_vector(rhs1, 0)
        me.M.init_vector(rhs2, 0)
        solver = dl.LUSolver()
        solver.set_operator(me.A_adj)
        for i in range(nt-2, -1, -1):
            rhs1 = me.M * p[i+1].vector()
            if me.simulation_times[i] > me.T1:
                rhs2 = me.M * (u[i+1].vector() - me.ud[i+1].vector())
                rhs1.axpy(-1.*me.dt, rhs2)
            solver.solve(p[i].vector(), rhs1)
        return p
    def cost(me, m):
        u = me.solve_fwd(m)
        misfit = 0.
        data_discrepancy = dl.Function(me.Vh)
        for i, t in enumerate(me.simulation_times):
            if t > me.T1:
                data_discrepancy.assign(u[i])
                data_discrepancy.vector().axpy(-1.0, me.ud[i].vector())
                misfit += 0.5*me.dt*dl.norm(data_discrepancy, 'L2')**2.
        reg = 0.5*me.gamma*dl.assemble(dl.inner(dl.grad(m), dl.grad(m))*dl.dx(me.Vh.mesh()
        return [misfit+reg, misfit, reg]
    def grad(me, m):
        u = me.solve_fwd(m)
        p = me.solve_adj(u)
        mtest = dl.TestFunction(me.Vh)
        MGform = (me.gamma*dl.inner(dl.grad(m), dl.grad(mtest)) - p[0]*mtest)*dl.dx(me.Vh.
        MG     = dl.assemble(MGform)
        return MG

nts = [10, 100, 1000]

for nt in nts:
    simulation_times = np.linspace(0., T, num=nt)
    gamma = 1.e-6
    hess = reducedHessian(Vh, simulation_times, kappa, v, T1, gamma)

    u = hess.solve_fwd(mtrue)

    ud = [dl.Function(Vh) for i in range(nt)]

    noise_lvl = 0.02
    for i in range(nt):
        ud[i].assign(u[i])
        uL2 = dl.norm(u[i], 'L2')
        noise = noise_lvl * uL2 *np.random.randn(Vh.dim())
```

```
        ud[i].vector().vec()[:] += noise
    hess.ud = ud

    m0 = dl.interpolate(dl.Expression('1.+std::exp(x[0])*std::cos(pi*x[0])*std::cos(pi*x[1
                                      element=Vh.ufl_element(), pi=np.pi), Vh)
    mdir = dl.interpolate(dl.Expression('std::cos(2.*x[0]*pi)', \
                                       element=Vh.ufl_element(), pi=np.pi), Vh)

    J0 = hess.cost(m0)[0]

    grad0 = hess.grad(m0)


    n_eps = 24
    eps = 1e4*np.power(2., -np.arange(n_eps))
    err_grad = np.zeros(n_eps)

    mh   = dl.Function(Vh)
    mhat = dl.Function(Vh).vector()
    mhat.set_local(mdir.vector().get_local())

    dir_grad0 = grad0.inner(mhat)



    for i in range(n_eps):
        mh.assign(m0)
        mh.vector().axpy(eps[i], mhat) #uh = uh + eps[i]*dir
        Jplus = hess.cost(mh)[0]
        err_grad[i] = abs( (Jplus - J0)/eps[i] - dir_grad0 )

    plt.figure()
    plt.loglog(eps, err_grad, "-ob", label="Error Grad, dt = {0:1.1e}".format(hess.dt))
    plt.loglog(eps, (.5*err_grad[0]/eps[0])*eps, "-.k", label=r"First Order, $\propto\vare
    plt.title("Finite difference check of the first variation")
    plt.xlabel(r"$\varepsilon$")
    plt.ylabel(r"$r(\varepsilon)$, finite-difference error")
    plt.legend(loc = "upper left")
    pairs = [(eps[i], err_grad[i]) for i in range(n_eps)]
    np.savetxt("fd_error_nt"+str(nt)+".dat", pairs)
plt.show()

pairs = [(eps[i], 0.5*err_grad[0]/eps[0]*eps[i]) for i in range(n_eps)]
np.savetxt("linear_referencecurve.dat", pairs)
```

B.2 ip advection diffusion deterministic incg (DTO code to collect data)

```
import fenics as dl

import numpy as np
import matplotlib.pyplot as plt
from hippylib import nb
from hippylib import CGSolverSteihaug
import ufl
import struct

plt.style.use('classic')
plt.rcParams.update({'font.size': 16})

# load mesh and define finite element function spaces
```

```
mesh = dl.Mesh("ad_20.xml")
mesh = dl.refine(mesh)

dl.plot(mesh)
plt.show()

Vh = dl.FunctionSpace(mesh, "CG", 1)
print("Number of dofs: {}".format(Vh.dim()))

# define the diffusion coefficient field and final time
kappa = 0.001 # diffusion constant
T = 4    # final time

# construct the advection velocity field by solving the Navier Stokes equation
def v_boundary(x,on_boundary):
    return on_boundary

def q_boundary(x,on_boundary):
    return x[0] < dl.DOLFIN_EPS and x[1] < dl.DOLFIN_EPS

def computeVelocityField(mesh):
    Xh = dl.VectorFunctionSpace(mesh,'Lagrange', 2)
    Wh = dl.FunctionSpace(mesh, 'Lagrange', 1)
    mixed_element = ufl.MixedElement([Xh.ufl_element(), Wh.ufl_element()])
    XW = dl.FunctionSpace(mesh, mixed_element)

    Re = dl.Constant(1e2)

    g = dl.Expression(('0.0','(x[0] < 1e-14) - (x[0] > 1 - 1e-14)'), degree=1)
    bc1 = dl.DirichletBC(XW.sub(0), g, v_boundary)
    bc2 = dl.DirichletBC(XW.sub(1), dl.Constant(0), q_boundary, 'pointwise')
    bcs = [bc1, bc2]

    vq = dl.Function(XW)
    (v,q) = ufl.split(vq)
    (v_test, q_test) = dl.TestFunctions (XW)

    def strain(v):
        return ufl.sym(ufl.grad(v))

    F = ( (2./Re)*ufl.inner(strain(v),strain(v_test))+ ufl.inner (ufl.nabla_grad(v)*v, v_t
          - (q * ufl.div(v_test)) + ( ufl.div(v) * q_test) ) * ufl.dx

    dl.solve(F == 0, vq, bcs, solver_parameters={"newton_solver":
                                        {"relative_tolerance":1e-4, "maximum_iterations":

    plt.figure(figsize=(15,5))
    vh = dl.project(v,Xh)
    qh = dl.project(q,Wh)
    nb.plot(nb.coarsen_v(vh), subplot_loc=121,mytitle="Velocity")
    nb.plot(qh, subplot_loc=122,mytitle="Pressure")
    plt.show()

    return v

v = computeVelocityField(mesh)


mtrue = dl.interpolate( dl.Expression('std::min(0.5,std::exp(-100*(std::pow(x[0]-0.35,2) +
            std::pow(x[1]-0.7,2))))', element=Vh.ufl_element()), Vh)
```

```
nb.plot(mtrue, mytitle=r"$m_{true}(x)$")
plt.show()

class reducedHessian:
    def __init__(me, Vh, simulation_times, kappa, v, T1, gamma1, gamma2):
        """
        :param Vh: finite element function space
        :param simulation_times: discretized time points
        :param kappa: diffusion coefficient
        :param v: advection velocity field
        :param T1: starting observation time
        :param gamma1: regularization parameter corresponding to L^2 regularization
        :param gamma2: regularization parameter corresponding to H^1 regularization
        """
        me.nt = len(simulation_times)
        me.dt = simulation_times[1] - simulation_times[0]
        me.Vh = Vh
        me.simulation_times = simulation_times
        me.T1 = T1
        me.gamma1 = gamma1
        me.gamma2 = gamma2
        me.kappa  = kappa
        me.v      = v
        utest  = dl.TestFunction(Vh)
        utrial = dl.TrialFunction(Vh)
        Kform  = dl.inner(dl.grad(utest), dl.grad(utrial))*dl.dx(me.Vh.mesh())
        Mform  = utest*utrial*dl.dx(me.Vh.mesh())
        Bform  = utest*dl.inner(me.v, dl.grad(utrial))*dl.dx(me.Vh.mesh())
        BTform = utrial*dl.inner(me.v, dl.grad(utest))*dl.dx(me.Vh.mesh())
        Rform  = (gamma1*utest*utrial + gamma2*dl.inner(dl.grad(utest), dl.grad(utrial)))*

        me.K  = dl.assemble(Kform)
        me.M  = dl.assemble(Mform)
        me.B  = dl.assemble(Bform)
        me.BT = dl.assemble(BTform)
        me.R  = dl.assemble(Rform)

        me.A     = me.M + me.dt*(me.kappa*me.K + me.B)
        me.A_adj = me.M + me.dt*(me.kappa*me.K + me.BT)

        me.ud    = None

    def solve_fwd(me, m):
        """
        For given m, solve the forward problem for u

        u[0]     = m
        A u[1]   = M u[0]
        ...
        A u[i]   = M u[i]
        ...
        A u[-1] = M u[-2]
        """
        u = [dl.Function(me.Vh) for i in range(me.nt)]
        u[0].assign(m)
        solver = dl.LUSolver()
        solver.set_operator(me.A)
        rhs = dl.Vector()
        me.A.init_vector(rhs, 0)
        for i in range(1, me.nt):
            rhs = me.M * u[i-1].vector()
```

```
            solver.solve(u[i].vector(), rhs)
        return u

    def solve_adj(me, u):
        """
        For given u, solve the adjoint problem for p

        A_adj p[-1] = - dt MGamma (u[-1] - ud[-1])
        A_adj p[-2] = - dt MGamma (u[-2] - ud[-2]) + M p[-1]
        ...
        A_adj p[i]  = - dt MGamma (u[i] - ud[i]) + M p[i+1]
        ...
        A_adj p[j]  = M p[j+1]
        ...
        A_adj p[1]  = M p[2]
        M p[0]      = M p[1]
        """
        p = [dl.Function(me.Vh) for i in range(me.nt)]
        rhs1 = dl.Vector()
        rhs2 = dl.Vector()
        me.M.init_vector(rhs1, 0)
        me.M.init_vector(rhs2, 0)
        solver = dl.LUSolver()
        solver.set_operator(me.A_adj)

        for i in range(nt-1, 0, -1):
            if i < nt-1:
                rhs1 = me.M * p[i+1].vector()
            if me.simulation_times[i] > me.T1:
                rhs1 = -1.*me.dt*me.MGamma*(u[i].vector() - me.ud[i].vector())
                if i != nt - 1:
                    rhs2 = me.M * p[i+1].vector()
                    rhs1.axpy(1., rhs2)
            solver.solve(p[i].vector(), rhs1)
        p[0].vector().set_local(p[1].vector())
        return p

    def cost(me, m):
        u = me.solve_fwd(m)
        misfit = 0.
        data_discrepancy = dl.Function(me.Vh)
        tempvec = dl.Vector()
        me.MGamma.init_vector(tempvec, 0)

        for i, t in enumerate(me.simulation_times):
            if t > me.T1:
                data_discrepancy.assign(u[i])
                data_discrepancy.vector().axpy(-1.0, me.ud[i].vector())

                tempvec = me.MGamma*data_discrepancy.vector()
                misfit += 0.5*me.dt*tempvec.inner(data_discrepancy.vector())
        reg = 0.5*m.vector().inner(me.R * m.vector())
        return [misfit+reg, misfit, reg]

    def grad(me, m):
        u = me.solve_fwd(m)
        p = me.solve_adj(u)
        MG = dl.Vector()
        me.M.init_vector(MG, 0)
        MG.axpy(1.0, me.R * m.vector())
        MG.axpy(-1.0, me.M * p[0].vector())
```

```
    return MG

def solve_incfwd(me, mhat):
    uhat = [dl.Function(me.Vh) for i in range(me.nt)]
    uhat[0].assign(mhat)
    solver = dl.LUSolver()
    solver.set_operator(me.A)
    rhs = dl.Vector()
    me.A.init_vector(rhs, 0)
    for i in range(1, me.nt):
        rhs = me.M * uhat[i-1].vector()
        solver.solve(uhat[i].vector(), rhs)
    return uhat

def solve_incadj(me, uhat):
    phat = [dl.Function(me.Vh) for i in range(me.nt)]
    rhs1 = dl.Vector()
    rhs2 = dl.Vector()
    me.M.init_vector(rhs1, 0)
    me.M.init_vector(rhs2, 0)
    solver = dl.LUSolver()
    solver.set_operator(me.A_adj)
    for i in range(nt-1, 0, -1):
        if i < nt-1:
            rhs1 = me.M * phat[i+1].vector()
        if me.simulation_times[i] > me.T1:
            rhs1 = -1*me.dt*me.MGamma*uhat[i].vector()
            if i != nt-1:
                rhs2 = me.M * phat[i+1].vector()
                rhs1.axpy(1., rhs2)
        solver.solve(phat[i].vector(), rhs1)
    phat[0].vector().set_local(phat[1].vector())
    return phat

def init_vector(me, x, j):
    me.M.init_vector(x, j)

def mult(me, x, y):
    mhat = dl.Function(me.Vh)
    mhat.vector().zero()
    mhat.vector().axpy(1.0, x)
    uhat = me.solve_incfwd(mhat)
    phat = me.solve_incadj(uhat)
    y.zero()
    y.axpy(1.0, me.R * x)
    y.axpy(-1.0, me.M * phat[0].vector())

def setup_Gamma_m(me, boundary):
    """
    Construct MGamma, the mass matrix defined over `boundary`.


    :param boundary: boundary on which the measurement lives
    """
    boundary_markers = dl.MeshFunction("size_t", me.Vh.mesh(), me.Vh.mesh().topology()
    boundary.mark(boundary_markers, 1)
    ds = dl.Measure("ds", domain=me.Vh.mesh(), subdomain_data=boundary_markers)
    bc_for_dofs = dl.DirichletBC(me.Vh, dl.Constant(0), boundary)

    utest = dl.TestFunction(me.Vh)
    utrial = dl.TrialFunction(me.Vh)
```

```
        # construct the measurement operator
        MGammaform = utest*utrial*ds(1)
        me.MGamma = dl.assemble(MGammaform)

        # dofs living on the boundary 'boundary'
        me.dofs_Gamma = list(bc_for_dofs.get_boundary_values().keys())

nt = 1000
simulation_times = np.linspace(0., T, num=nt)
T1    = 0.4*T
gamma1 = 1.0e-5
gamma2 = 1.0e-6
hess  = reducedHessian(Vh, simulation_times, kappa, v, T1, gamma1,  gamma2)

# boundary of the two rectangular holes
class HoleBoundary(dl.SubDomain):
    def inside(self, x, on_boundary):
        left_right = dl.near(x[0], 0) or dl.near(x[0], 1)
        top_bottom = dl.near(x[1], 0) or dl.near(x[1], 1)

        return not (left_right or top_bottom) and on_boundary

boundary_twoholes = HoleBoundary()
hess.setup_Gamma_m(boundary_twoholes)

u = hess.solve_fwd(mtrue)
nb.multi1_plot([u[0], u[-1]], [r"$u(x, t=0)$", r"$u(x, t=T)$"])
plt.show()

ud = [dl.Function(Vh) for i in range(nt)]

noise_lvl = 0.02
for i in range(nt):
    ud[i].assign(u[i])
    uL2 = dl.norm(u[i], 'L2')
    noise = noise_lvl * uL2 *np.random.randn(Vh.dim())
    ud[i].vector().vec()[:] += noise

hess.ud = ud

n_eps = 24
eps = 1e-2*np.power(2., -np.arange(n_eps))
err_grad = np.zeros(n_eps)

m0 = dl.interpolate(dl.Expression('1.+std::exp(x[0])*std::cos(pi*x[0])*std::cos(pi*x[1])',
                                  element=Vh.ufl_element(), pi=np.pi), Vh)
mdir = dl.interpolate(dl.Expression('std::cos(2.*x[0]*pi)', \
                                  element=Vh.ufl_element(), pi=np.pi), Vh)


J0 = hess.cost(m0)[0]
grad0 = hess.grad(m0)

mh   = dl.Function(Vh)
mhat = dl.Function(Vh).vector()
mhat.set_local(mdir.vector().get_local())

dir_grad0 = grad0.inner(mhat)
```

```
for i in range(n_eps):
    mh.assign(m0)
    mh.vector().axpy(eps[i], mhat) #uh = uh + eps[i]*dir
    Jplus = hess.cost(mh)[0]
    err_grad[i] = abs( (Jplus - J0)/eps[i] - dir_grad0 )


plt.figure()
plt.loglog(eps, err_grad, "-ob", label="Error Grad")
plt.loglog(eps, (.5*err_grad[0]/eps[0])*eps, "-.k", label=r"First Order, $\propto\varepsilo
plt.title("Finite difference check of the first variation")
plt.xlabel(r"$\varepsilon$")
plt.ylabel(r"$r(\varepsilon)$, finite-difference error")
plt.legend(loc = "upper left")
pairs = [(eps[i], err_grad[i]) for i in range(n_eps)]
np.savetxt("fd_error_nt"+str(nt)+".dat", pairs)
plt.savefig('gradcheck.png',  dpi=200)
plt.show()

pairs = [(eps[i], 0.5*err_grad[0]/eps[0]*eps[i]) for i in range(n_eps)]
np.savetxt("linear_referencecurve.dat", pairs)

m0 = dl.interpolate(dl.Constant(0.0), Vh)
mh.assign(m0)
g = hess.grad(m0)

solver = CGSolverSteihaug()
solver.set_operator(hess)
precond_solver = dl.LUSolver()
precond_solver.set_operator(hess.R)
solver.set_preconditioner(precond_solver)
solver.parameters["print_level"]= 1
solver.parameters["rel_tolerance"] = 1e-6
solver.solve(mh.vector(), -1.*g)

mh.vector().axpy(1.0, m0.vector())
nb.multi1_plot([m0, mtrue], ["initial parameter guess", "true parameter"], same_colorbar=F
plt.show()
nb.multi1_plot([mh, mtrue], ["infered parameter", "true parameter"])
plt.show()

uh = hess.solve_fwd(mh)
nb.multi1_plot([uh[-1], ud[-1]], ["infered final state", "observed final state"])
```

References

[1] Petra, N., Stadler, G., *Model variational inverse problems governed by partial differential equations.*, Texas University at Austin Inst for Computational Engineering and Sciences, 2011.

[2] Gunzburger, Max D, *Perspectives in flow control and optimization.*, Society for Industrial and Applied Mathematics, 2002.