ICES REPORT 11-05

March 2011

Model Variational Inverse Problems Governed by Partial Differential Equations

by

Noemi Petra and Georg Stadler

The Institute for Computational Engineering and Sciences
The University of Texas at Austin
Austin, Texas 78712

Reference: Noemi Petra and Georg Stadler, "Model Variational Inverse Problems Governed by Partial Differential Equations", ICES REPORT 11-05, The Institute for Computational Engineering and Sciences, The University of Texas at Austin, March 2011.

MODEL VARIATIONAL INVERSE PROBLEMS GOVERNED BY PARTIAL DIFFERENTIAL EQUATIONS*

NOEMI PETRA† AND GEORG STADLER†

Abstract. We discuss solution methods for inverse problems, in which the unknown parameters are connected to the measurements through a partial differential equation (PDE). Various features that commonly arise in these problems, such as inversions for a coefficient field, for the initial condition in a time-dependent problem, and for source terms are being studied in the context of three model problems. These problems cover distributed, boundary, as well as point measurements, different types of regularizations, linear and nonlinear PDEs, and bound constraints on the parameter field. The derivations of the optimality conditions are shown and efficient solution algorithms are presented. Short implementations of these algorithms in a generic finite element toolkit demonstrate practical strategies for solving inverse problems with PDEs. The complete implementations are made available to allow the reader to experiment with the model problems and to extend them as needed.

Key words. inverse problems, PDE-constrained optimization, adjoint methods, inexact Newton method, steepest descent method, coefficient estimation, initial condition estimation, generic PDE toolkit

AMS subject classifications. 35R30, 49M37, 65K10, 90C53

1. Introduction. The solution of inverse problems, in which the parameters are linked to the measurements through the solution of a partial differential equation (PDE) is becoming increasingly feasible due to the growing computational resources and the maturity of methods to solve PDEs. Often, a regularization approach is used to overcome the ill-posedness inherent in inverse problems, which results in a continuous optimization problem with a PDE as equality constraint and a cost functional that involves a data misfit and a regularization term. After discretization, these problems result in a large-scale numerical optimization problem, with specific properties that depend on the underlying PDE, the type of regularization and on the available measurements.

We use three model problems to illustrate typical properties of inverse problems with PDEs, discuss solvers and demonstrate their implementation in a generic finite element toolkit. Based on these model problems we discuss several commonly occurring features, as for instance the estimation of a parameter field in an elliptic equation, the inversion for right-hand side forces and for the initial condition in a time-dependent problem. Distributed, boundary or point measurements are used to reconstruct parameter fields that are defined on the domain or its boundary. The derivation of the optimality conditions is demonstrated for the model problems and the steepest descent method as well as inexact Newton-type algorithms for their solution are discussed.

Numerous toolkits and libraries for finite element computations based on variational forms are available, for instance *COMSOL Multiphysics* [10], *deal.II* [4], *dune* [5], the *FEniCS project* [11, 20] and *Sundance*, a package from the *Trilinos project* [17]. These toolkits are usually tailored towards the solution of PDEs and systems of PDEs,

†Institute for Computational Engineering & Sciences, The University of Texas at Austin, Austin, TX 78712, USA (noemi@ices.utexas.edu, georgst@ices.utexas.edu)

*This work is partially supported by NSF grants OPP-0941678 and DMS-0724746; DOE grants DE-SC0002710 and DE-FG02-08ER25860; and AFOSR grant FA9550-09-1-0608. N. P. also acknowledges partial funding through the ICES Postdoctoral Fellowship. We would like to thank Omar Ghattas for helpful discussions and comments.

and cannot be used straightforwardly for the solution of inverse problems with PDEs. However, several of the above mentioned packages are sufficiently flexible to be used for the solution of inverse problems governed by PDEs. Nevertheless, some knowledge of the structure underlying these packages is required since the optimality systems arising in inverse problems with PDEs often cannot be solved using generic PDE solvers, which do not exploit the optimization structure of the problems. For illustration purposes, this report includes implementations of the model problems in COMSOL Multiphysics (linked together with MATLAB)[1]. Since our implementations use little finite element functionality that is specific to COMSOL Multiphysics, only few code pieces have to be changed in order to have these implementations available in other finite element packages.

Related papers, in which the use of generic discretization toolkits for the solution of PDE-constrained optimization or inverse problems is discussed are [15,21,23]. Note that the papers [21, 23] focus on optimal control problems and, differently from our approach, the authors use the nonlinear solvers provided by COMSOL Multiphysics to solve the arising optimality systems. For inverse problems, which often involve significant nonlinearities, this approach is often not an option. In [15], finite difference-discretized PDE-constrained optimization problems are presented and short MATLAB implementations for an elliptic, a parabolic, and a hyperbolic model problem are provided. A systematic review of methods for optimization problems with implicit constraints, as they occur in inverse or optimization problems with PDEs can be found in [16]. For a comprehensive discussion of regularization methods for inverse problems, and the numerical solution of inverse problems (which do not necessarily involve PDEs) we refer the reader to the text books [9, 13, 24, 26].

The organization of this paper is as follows. The next three sections present model problems, discuss the derivation of the optimality systems, and explain different solver approaches. In Appendix A, we discuss practical computation issues and give snippets of our implementations. Complete code listings can be found in Appendix B and can be downloaded from the authors' websites.

2. Parameter field inversion in elliptic problem. We consider the estimation of a coefficient in an elliptic partial differential equation as a first model problem. Depending on the interpretation of the unknowns and the type of measurements, this model problem arises, for instance, in inversion for groundwater flow or heat conductivity. It can also be interpreted as finding a membrane with a certain spatially varying stiffness. Let $\Omega \subset \mathbb{R}^n$, $n \in \{1, 2, 3\}$ be an open, bounded domain and consider the following problem:

$$\min_a J(a) := \frac{1}{2}\int_\Omega (u - u_d)^2\,dx + \frac{\gamma}{2}\int_\Omega |\nabla a|^2\,dx, \tag{2.1a}$$

where u is the solution of

$$\begin{aligned} -\nabla\cdot(a\nabla u) &= f \text{ in } \Omega, \\ u &= 0 \text{ on } \partial\Omega, \end{aligned} \tag{2.1b}$$

and

$$a \in U_{ad} := \{a \in L^\infty(\Omega), a \geq a_o > 0\}. \tag{2.1c}$$

[1] Our implementations are based on COMSOL Multiphysics v3.5a (or earlier). In the most recent versions, v4.0 and v4.1, the scripting syntax in COMSOL Multiphysics (with MATLAB) has been changed. We plan to adjust the implementations of our model problems to these most recent versions of COMSOL Multiphysics.

Here, u_d denotes (possibly noisy) data, $f \in H^{-1}(\Omega)$ a given force, $\gamma \geq 0$ the regularization parameter, and $a_0 > 0$ the lower bound for the unknown coefficient function a. In the sequel, we denote the L^2-inner product by $(\cdot\,,\cdot)$, *i.e.*, for scalar functions u, v and vector functions $\mathbf{u}, \mathbf{v}$ defined on Ω we denote

$$(u, v) := \int_\Omega u(x)v(x)\,dx \quad \text{and} \quad (\mathbf{u}, \mathbf{v}) := \int_\Omega \mathbf{u}(x) \cdot \mathbf{v}(x)\,dx,$$

where "$\cdot$" denotes the inner product between vectors. With this notation, the variational (or weak) form of the state equation (2.1b) is: Find $u \in H_0^1(\Omega)$ such that

$$(a\nabla u, \nabla z) - (f, z) = 0 \quad \text{for all} \quad z \in H_0^1(\Omega), \tag{2.2}$$

where $H_0^1(\Omega)$ is the space of functions vanishing on $\partial\Omega$ with square integrable derivatives. It is well known that for every a, which is bounded away from zero, (2.2) admits a unique solution, u (this follows from the Lax-Milgram theorem [7]). Based on this result it can be shown that the regularized inverse problem (2.1) admits a solution [13, 26]. However, this solution is not necessary unique.

2.1. Optimality system. We now compute the first-order optimality conditions for (2.1), where, for simplicity of the presentation, we neglect the bound constraints on a, *i.e.*, $U_{ad} := L^\infty(\Omega)$. We use the (formal) Lagrangian approach (see, e.g., [14,25]) to compute the optimality conditions that must be satisfied at a solution of (2.1). For that purpose we introduce a Lagrange multiplier function p to enforce the elliptic partial differential equation (2.1b) (in the weak form (2.2)). In general, the function p inherits the type of boundary condition as u, but satisfies homogeneous conditions. In this case, this means that $p \in H_0^1(\Omega)$. The Lagrangian functional $\mathscr{L} : L^\infty(\Omega) \times H_0^1(\Omega) \times H_0^1(\Omega) \to \mathbb{R}$, which we use as a tool to derive the optimality system, is given by

$$\mathscr{L}(a, u, p) := \frac{1}{2}(u - u_d, u - u_d) + \frac{\gamma}{2}(\nabla a, \nabla a) + (a\nabla u, \nabla p) - (f, p). \tag{2.3}$$

Here, a and u are considered as independent variables. The Lagrange multiplier theory shows that, at a solution of (2.1) variations of the Lagrangian functional with respect to all variables must vanish. These variations of $\mathscr{L}$ with respect to (p, u, a) in directions $(\tilde{u}, \tilde{p}, \tilde{a})$ are given by

$$\mathscr{L}_p(a, u, p)(\tilde{p}) = (a\nabla u, \nabla\tilde{p}) - (f, \tilde{p}) \qquad = 0, \tag{2.4a}$$
$$\mathscr{L}_u(a, u, p)(\tilde{u}) = (a\nabla p, \nabla\tilde{u}) + (u - u_d, \tilde{u}) = 0, \tag{2.4b}$$
$$\mathscr{L}_a(a, u, p)(\tilde{a}) = \gamma(\nabla a, \nabla\tilde{a}) + (\tilde{a}\nabla u, \nabla p) = 0, \tag{2.4c}$$

where the variations $(\tilde{u}, \tilde{p}, \tilde{a})$ are taken from the same spaces as (u, p, a). Note that (2.4a) is the weak (or variational) form of the *state equation* (2.1b). Moreover, assuming that the solutions are sufficiently regular, (2.4b) is the weak form of the *adjoint equation*

$$-\nabla \cdot (a\nabla p) = -(u - u_d) \quad \text{in } \Omega, \tag{2.5a}$$
$$p = 0 \quad \text{on } \partial\Omega. \tag{2.5b}$$

In addition, the strong form of the *control equation* (2.4c) is given by

$$-\nabla \cdot (\nabla a) = -\nabla u \cdot \nabla p \quad \text{in } \Omega, \tag{2.6a}$$
$$\nabla a \cdot \mathbf{n} = 0 \quad \text{on } \partial\Omega. \tag{2.6b}$$

Note that the optimality conditions (2.4) (in weak form) or (2.1b), (2.5) and (2.6) (in strong form) form a system of PDEs. This system is nonlinear, even though the state equation is linear (in u). To find the solution of (2.1), these conditions need to be solved. We now summarize common approaches to solve such a system of PDEs. Naturally, PDE systems of this form can only be solved numerically, *i.e.*, they have to be discretized using, for instance, the finite element method. In the sequel, we use variational forms to present our algorithms. These forms can be interpreted as continuous (*i.e.*, in function spaces) or finite-dimensional as they arise in finite element discretized problems. For illustration purposes we also use block-matrix notation for the discretized problems.

2.2. Steepest descent method. We start with describing the steepest descent method [19,26] for the solution of (2.1). This method uses first-order derivative (*i.e.*, gradient) information only to iteratively minimize (2.1a). While being simple and commonly used, it cannot be recommended for most inverse problems with PDEs due to its unfavorable convergence properties. However, we briefly discuss this approach for completeness of the presentation.

The steepest descent method updates the parameter field a using the gradient $g := \nabla_a J(a)$ of problem (2.1). It follows from Lagrange theory (*e.g.*, [25]) that this gradient is given by the left hand side in (2.4c), provided (2.4a) and (2.4b) are satisfied. Thus, the steepest descent method for the solution of (2.4) computes iterates (u_k, p_k, a_k) $(k = 1, 2, \ldots)$ as follows: Given a coefficient field a_k, the gradient g_k is computed by first solving the state problem

$$(a_k \nabla u_k, \nabla \tilde{p}) - (f, \tilde{p}) = 0 \quad \text{for all } \tilde{p}, \tag{2.7a}$$

for u_k. With this u_k given, the adjoint equation

$$(a_k \nabla p_k, \nabla \tilde{u}) + (u_k - u_d, \tilde{u}) = 0 \quad \text{for all } \tilde{u} \tag{2.7b}$$

is solved for p_k. Finally, the gradient g_k is obtained by solving

$$\gamma(\nabla a_k, \nabla \tilde{g}) + (\tilde{g} \nabla u_k, \nabla p_k) = (g_k, \tilde{g}) \quad \text{for all } \tilde{g}. \tag{2.7c}$$

Since the negative gradient g_k is a descent direction for the cost functional J, it is used to update the coefficient a_k, *i.e.*,

$$a_{k+1} := a_k - \alpha_k g_k. \tag{2.7d}$$

Here, α_k is an appropriately chosen step length such that the cost functional is sufficiently decreased. Sufficient descent can be guaranteed, for instance, by choosing α_k that satisfies the Armijo or Wolfe condition [22]. This process is repeated until the norm of the gradient g_k is sufficiently small. A description of the implementation of the steepest descent method in COMSOL Multiphysics, as well as a complete code listing can be found in Appendix A.1 and Appendix B.1. While the steepest descent method is simple and commonly used, Newton-type methods are often preferred due to their faster convergence.

2.3. Newton methods. Next, we discuss variants of the Newton's method for the solution of the optimality system (2.4). The Newton method requires second-order variational derivatives of the Lagrangian (2.3). Written in abstract form, it computes

an update direction $(\hat{a}_k, \hat{u}_k, \hat{p}_k)$ from the following Newton step for the Lagrangian functional:

$$\mathscr{L}''(a_k, u_k, p_k)\,[(\hat{a}_k, \hat{u}_k, \hat{p}_k), (\tilde{a}, \tilde{u}, \tilde{p})] = -\mathscr{L}'(a_k, u_k, p_k)(\tilde{a}, \tilde{u}, \tilde{p}), \tag{2.8}$$

for all variations $(\tilde{a}, \tilde{u}, \tilde{p})$, where $\mathscr{L}'$ and $\mathscr{L}''$ denote the first and second variations of the Lagrangian (2.3). For the elliptic parameter inversion problem (2.1), this Newton step (written in variatonal form) is as follows: Find $(\hat{u}_k, \hat{a}_k, \hat{p}_k)$ as the solution of the linear system

$$\begin{array}{llll} (\hat{u}_k, \tilde{u}) & +(\hat{a}_k \nabla p_k, \nabla \tilde{u}) & +(a_k \nabla \tilde{u}, \nabla \hat{p}_k) & = (u_d - u_k, \tilde{u}) - (a_k \nabla p_k, \nabla \tilde{u}) \\ (\tilde{a} \nabla \hat{u}_k, \nabla p_k) & +\gamma(\nabla \hat{a}_k, \nabla \tilde{a}) & +(\tilde{a} \nabla u_k, \nabla \hat{p}_k) & = -\gamma(\nabla a_k, \nabla \tilde{a}) - (\tilde{a} \nabla u_k, \nabla p_k) \\ (a_k \nabla \hat{u}_k, \nabla \tilde{p}) & +(\hat{a}_k \nabla u_k, \nabla \tilde{p}) & & = -(a_k \nabla u_k, \nabla \tilde{p}) + (f, \tilde{p}), \end{array} \tag{2.9}$$

for all $(\tilde{u}, \tilde{a}, \tilde{p})$. To illustrate features of the Newton method, we use the matrix notation for the discretized Newton step and denote the vectors corresponding to the discretization of the functions $\hat{a}_k, \hat{u}_k, \hat{p}_k$ by $\hat{\boldsymbol{a}}_k, \hat{\boldsymbol{u}}_k$ and $\hat{\boldsymbol{p}}_k$. Then, the discretization of (2.9) is given by the following symmetric linear system

$$\begin{bmatrix} \mathbf{W}_{\text{uu}} & \mathbf{W}_{\text{ua}} & \mathbf{A}^T \\ \mathbf{W}_{\text{au}} & \mathbf{R} & \mathbf{C}^T \\ \mathbf{A} & \mathbf{C} & 0 \end{bmatrix} \begin{bmatrix} \hat{\boldsymbol{u}}_k \\ \hat{\boldsymbol{a}}_k \\ \hat{\boldsymbol{p}}_k \end{bmatrix} = - \begin{bmatrix} \boldsymbol{g}_u \\ \boldsymbol{g}_a \\ \boldsymbol{g}_p \end{bmatrix}, \tag{2.10}$$

where $\mathbf{W}_{\text{uu}}$, $\mathbf{W}_{\text{ua}}$, $\mathbf{W}_{\text{au}}$, and $\mathbf{R}$ are the components of the Hessian matrix of the Lagrangian, $\mathbf{A}$ and $\mathbf{C}$ are the Jacobian of the state equation with respect to the state and the control variables, respectively and $\boldsymbol{g}_u$, $\boldsymbol{g}_a$, and $\boldsymbol{g}_p$ are the discrete gradients of the Lagrangian with respect to $\boldsymbol{u}$, $\boldsymbol{a}$ and $\boldsymbol{p}$, respectively.

Systems of the form (2.10), which commonly arise in constrained optimization problems are called Karush-Kuhn-Tucker (KKT) systems. These systems are usually indefinite, *i.e.*, they have negative and positive eigenvalues. In many applications, the KKT systems can be very large. Thus, solving them with direct solvers is often not an option, and iterative solvers must be used; we refer to [2] for an overview of iterative methods for KKT systems.

To relate the Newton step on the first-order optimality system to the underlying optimization problem (2.1), we use a block elimination in (2.10). Also, we assume that $\boldsymbol{u}_k$ and $\boldsymbol{p}_k$ satisfy the state and the adjoint equations such that $\boldsymbol{g}_u = \boldsymbol{g}_p = 0$. To eliminate the incremental state and adjoint variables, $\hat{\boldsymbol{u}}_k$ and $\hat{\boldsymbol{p}}_k$, from the first and last equations in (2.10) we use

$$\hat{\boldsymbol{u}}_k = -\mathbf{A}^{-1}\mathbf{C}\,\hat{\boldsymbol{a}}_k, \tag{2.11a}$$

$$\hat{\boldsymbol{p}}_k = -\mathbf{A}^{-T}(\mathbf{W}_{\text{uu}}\hat{\boldsymbol{u}}_k + \mathbf{W}_{\text{ua}}\,\hat{\boldsymbol{a}}_k). \tag{2.11b}$$

This results in the following reduced linear system for the Newton step

$$\mathbf{H}\,\hat{\boldsymbol{a}}_k = -\boldsymbol{g}_a, \tag{2.12a}$$

with the *reduced Hessian* $\mathbf{H}$ given by

$$\mathbf{H} := \mathbf{R} + \mathbf{C}^T\mathbf{A}^{-T}(\mathbf{W}_{\text{uu}}\mathbf{A}^{-1}\mathbf{C} - \mathbf{W}_{\text{ua}}) - \mathbf{W}_{\text{au}}\mathbf{A}^{-1}\mathbf{C}. \tag{2.12b}$$

This reduced Hessian involves the inverse of the state and adjoint operators. This makes it a dense matrix that is often too large to be computed (and stored). However,

the reduced Hessian matrix can be applied to vectors by solving linear systems with the matrices $\mathbf{A}$ and $\mathbf{A}^T$. This allows to solve the reduced Hessian system (2.12a) using iterative methods such as the conjugate gradient method. Once the descent direction $\hat{\boldsymbol{a}}_k$ is computed, the next step is to apply a line search for finding an appropriate step size, α, as described in Section 2.2. Note that each backtracking step in the line search involves the evaluation of the cost functional, which amounts to the solution of the state equation with a trial coefficient field $\boldsymbol{a}'_{k+1}$.

The Newton direction $\hat{\boldsymbol{a}}_k$ is a descent direction for (2.1) only if the reduced Hessian (or an approximation $\tilde{\mathbf{H}}$ of the reduced Hessian) is positive definite. While $\mathbf{H}$ is positive in a neighborhood of the solution, it can be indefinite or singular away from the solution, and $\hat{\boldsymbol{a}}_k$ is not guaranteed to be a descent direction. There are several possibilities to overcome this problem. A simple remedy is to neglect the terms involving $\mathbf{W}_{\text{ua}}$ and $\mathbf{W}_{\text{au}}$ in (2.12b), which leads to the Gauss-Newton approximation of the Hessian, which is always positive definite. The resulting direction $\hat{\boldsymbol{a}}_k$ is always a descent direction, but the fast local convergence of Newton's method can be lost when neglecting the blocks $\mathbf{W}_{\text{ua}}$ and $\mathbf{W}_{\text{au}}$ in the Hessian matrix. A more sophisticated method to ensure the positive definiteness of an approximate Hessian is to terminate the conjugate gradient method for (2.12a) when a negative curvature direction is detected [12]. This approach, which is not followed here for simplicity, guarantees a descent direction while maintaining the fast Newton convergence close to the solution.

2.4. Gauss-Newton-CG method. To guarantee a descent direction $\hat{\boldsymbol{a}}_k$ in (2.12a), the Gauss-Newton method uses the approximate reduced Hessian

$$\tilde{\mathbf{H}} = \mathbf{R} + \mathbf{C}^T \mathbf{A}^{-T} \mathbf{W}_{\text{uu}} \mathbf{A}^{-1} \mathbf{C}. \tag{2.13}$$

Compared to (2.12b), using the inexact reduced Hessian (2.13) also has the advantage that the matrix blocks $\mathbf{W}_{\text{ua}}$ and $\mathbf{W}_{\text{au}}$ do not need to be assembled. Note that $\mathbf{W}_{\text{ua}}$ and $\mathbf{W}_{\text{au}}$ are proportional to the adjoint variable. If the measurements are attained at the solution (*i.e.*, $u = u_d$), the adjoint variable is zero and thus one obtains fast local convergence even when these blocks are being neglected. In general, particularly in the presence of noise, measurements are not attained exactly at the solution and the fast local convergence property of Newton's method is lost. However, often adjoint variables are small and the Gauss-Newton Hessian is a reasonable approximation for the full reduced Hessian.

The Gauss-Newton method can alternatively be interpreted as an iterative method that computes a search direction based on an auxiliary problem, which is given by a quadratic approximation of the cost functional and a linearization of the PDE constraint [22]. As for the Newton method, also for the Gauss-Newton method the optimal step length in a neighborhood of the solution is $\alpha = 1$. This property of Newton-type methods is a significant advantage compared to the steepest descent method, where no prior information on a good step length is available.

2.5. Bound constraints via the logarithmic barrier method. In the previous sections we have neglected the bound constraints on the coefficient function a in the inverse problems (2.1). However, in practical applications one often has to (or would like to) impose bound constraints on inversion parameters. This is usually due to *a priori* available physical knowledge about the parameters that are reconstructed in the inverse problem. In (2.1), for instance, a has to be bounded away from zero for physical reasons and to maintain the ellipticity (and unique solvability) of the state equation. Another example in which the result of the inversion can benefit from

imposing bounds is the problem in Section 3, where we invert for a concentration, which cannot be negative.

We now extend Newton's method to incorporate bounds of the form (2.1c). The approach used here is a very simplistic one, namely the logarithmic barrier method [22]. We add a logarithmic barrier with barrier parameter μ to the discretization of the optimization problem (2.1) to enforce $a - a_o \geq 0$, *i.e.*, a_o is the lower bound for the coefficient function a. Then, the discretized form of the Newton step on the optimality conditions is given by the following linear system

$$\begin{bmatrix} \mathbf{W}_{\text{uu}} & \mathbf{W}_{\text{ua}} & \mathbf{A}^T \\ \mathbf{W}_{\text{au}} & \mathbf{R}+\mathbf{Z} & \mathbf{C}^T \\ \mathbf{A} & \mathbf{C} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \hat{\boldsymbol{u}}_k \\ \hat{\boldsymbol{a}}_k \\ \hat{\boldsymbol{p}}_k \end{bmatrix} = \begin{bmatrix} -\boldsymbol{g}_u \\ -\boldsymbol{g}_a + \frac{\mu}{\boldsymbol{a}_k - \boldsymbol{a}_o} \\ -\boldsymbol{g}_p \end{bmatrix}, \tag{2.14}$$

where the same notations as in (2.10) are used. The terms due to the logarithmic barrier impose the bound constraints at nodal points and only appear in the control equation. The matrix $\mathbf{Z}$ is diagonal with components $\frac{\mu}{(\boldsymbol{a}_k - \boldsymbol{a}_o)^2}$. Note that both, $\mathbf{Z}$ and the right hand side term $\frac{\mu}{\boldsymbol{a}_k - \boldsymbol{a}_o}$ become large at points where a_k is close to the bound a_o, which can lead to ill-conditioning.

Neglecting the terms $\mathbf{W}_{\text{ua}}$ and $\mathbf{W}_{\text{au}}$ we obtain a Gauss-Newton method for the logarithmic barrier problem, similarly as demonstrated in Section 2.4. Once the Newton increment $\hat{\boldsymbol{a}}_k$ has been computed, a line search for the control variable update is applied. To assure that $\boldsymbol{a}_{k+1} - \boldsymbol{a}_o > 0$ the choice for the initial step length is [22]:

$$\alpha = \min\left(1, \min_{i:(\hat{\boldsymbol{a}}_k - \boldsymbol{a}_o)_i < 0} -\frac{(\boldsymbol{a}_k - \boldsymbol{a}_o)_i}{(\hat{\boldsymbol{a}}_k - \boldsymbol{a}_o)_i}\right). \tag{2.15}$$

It can be challenging to choose the barrier parameter μ appropriately. One would like μ to be small to keep the influence of the barrier function small in the inner of the feasible set U_{ad}. However, this can lead to small step lengths and severe ill-conditioning of the Newton system, which has led to the development of methods in which a series of logarithmic barrier problems are solved for a decreasing sequence of barrier parameters μ. For more sophisticated approaches to deal with bound constraints, such as the (primal-dual) interior point method, or (primal-dual) active set methods, we refer the reader to [3, 6, 22, 27]. An alternative way to impose bound constraints in the optimization problem is choosing a parametrization of the parameter field that already incorporates the constraint. For example, if $a_0 = 0$ one can parametrize the coefficient field as $a = \exp(b)$, and invert for b. Thus, a satisfies the non-negativity constraint by construction. This approach comes at the price of adding additional nonlinearity to the optimality system.

2.6. Numerical tests. In this section, we show results obtained with the steepest descent and the Gauss-Newton-CG methods as described in Sections 2.2 and 2.4. In particular, compare the number of iterations needed by these methods to achieve convergence for a particular tolerance. In Fig. 2.1, we show the "true" coefficient (left), which is used to synthesize measurement data by solving the state equation (a similar test example is used in [3]). We add noise to this synthetic data (see Fig. 2.1, center) to lessen the "inverse crime" [18], which occurs when the same numerical method is used for the synthetization of the data and for the inversion. An additional strategy to avoid inverse crimes would be solving the state equation to synthesize measurement data using a different discretization or, at least, a finer mesh. The recovered coefficient, *i.e.*, the solution of the inverse problem is shown on the right in

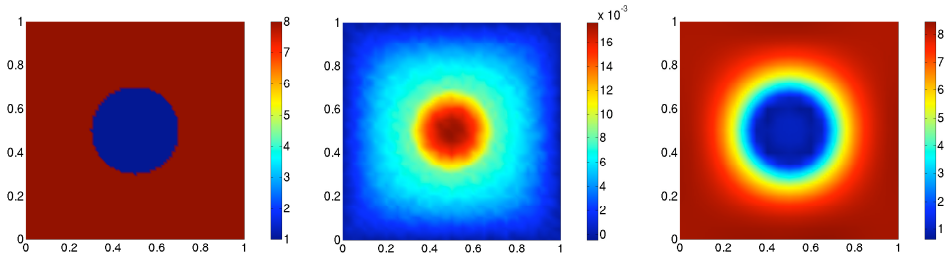

FIG. 2.1. *Results for the elliptic coefficient field inversion model problem* (2.1) *with* $\gamma = 10^{-9}$ *and* 1% *noise in the synthesized data. "True" coefficient field* a *(left), noisy data* u_d *(center), and recovered coefficient field* a *(right).*

Figure 2.1. Note that while the "true" coefficient is discontinuous, the reconstruction is continuous. This is a result of the regularization, which does not allow for discontinuous fields (the regularization term would be infinite if discontinuities were present in the reconstruction). A more appropriate regularization that allows discontinuous reconstructions is the total variation regularization, see Section 5.

Table 2.1 compares the number of iterations needed by the steepest descent and the Gauss-Newton-CG method. As can be seen, for all four meshes the number of iterations needed by the Gauss-Newton method is significantly smaller than the number of iterations needed by the steepest descent method. We note that while computing the steepest descent direction requires only the solution of one state, one adjoint and one control equation at each iteration, the Gauss-Newton method additionally requires the solution of the incremental equations for the state and adjoint at each CG iteration. Since an accurate solution of the Gauss-Newton system is only needed close to the solution, we use an inexact CG method to reduce the number of CG iterations, and hence the number of state-adjoint solves. For this purpose, we adjust the CG-tolerance by relating it to the norm of the gradient, *i.e.*,

$$\text{tol} = \min\left(0.5, \sqrt{\frac{\|\boldsymbol{g}_a^k\|}{\|\boldsymbol{g}_a^0\|}}\right)\|\boldsymbol{g}_a^k\|,$$

where $\boldsymbol{g}_a^0$ and $\boldsymbol{g}_a^k$ are the initial gradient and the gradient at iteration k, respectively. This choice of the tolerance leads to an early termination of the CG iteration away from the solution and enforces a more accurate solution of the Newton system as the gradient becomes small (*i.e.*, close to the solution). While compared to a more accurate computation of the Newton direction, this inexactness can result in a larger number of Newton iterations, but reduces the overall number of CG iterations significantly.

With the early termination of CG, the Gauss-Newton method requires significantly less number of forward-adjoint solves than does the steepest descent method, as can be seen in Table 2.1. For example, on a mesh of 40×40 elements, the Gauss-Newton method takes 11 outer (*i.e.*, Gauss-Newton) iterations and overall 27 inner (*i.e.*, CG) iterations, which amounts to 39 forward-adjoint solves overall. The steepest descent method, on the other hand, requires 267 state-adjoint solves. This performance difference becomes more evident on finer meshes: While for the Gauss-Newton method the iteration numbers remain almost constant, the steepest descent method requires significantly more iterations as the mesh is refined (see Table 2.1).

Mesh	Steepest descent #iter	Gauss-Newton (CG) #iter
10×10	68	10 (30)
20×20	97	10 (22)
40×40	267	11 (27)
80×80	>1000	12 (31)

TABLE 2.1

Number of iterations for the steepest descent and the Gauss-Newton methods for $\gamma = 10^{-9}$ and 1% noise in the synthetic data. Both iterations were terminated when the L^2-norm of the gradient dropped below 10^{-8}, or the maximum number of iterations was reached.

3. Initial condition inversion in advective-diffusive transport. We consider a time-dependent advection-diffusion equation, in which we invert for an unknown initial condition. The problem can be interpreted as finding the initial distribution of a contaminant from measurements taken after the contaminant has been subjected to diffusive transport [1]. Let $\Omega \subset \mathbb{R}^n$ be open and bounded (we choose $n = 2$ in the sequel) 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 is formulated as follows:

$$\min_{u_0} J(u_0) := \frac{1}{2}\int_{T_1}^{T}\int_{\Gamma_m} (u - u_d)^2 \, dx\, dt + \frac{\gamma_1}{2}\int_{\Omega} u_0^2 \, dx + \frac{\gamma_2}{2}\int_{\Omega} |\nabla u_0|^2 \, dx \tag{3.1a}$$

where u is the solution of

$$u_t - \kappa\Delta u + \mathbf{v}\cdot\nabla u = 0 \quad \text{in } \Omega \times [0,T], \tag{3.1b}$$

$$u(0,x) = u_0 \quad \text{in } \Omega, \tag{3.1c}$$

$$\kappa\nabla u \cdot \mathbf{n} = 0 \quad \text{on } \partial\Omega \times [0,T]. \tag{3.1d}$$

Here, u_d denotes measurements on Γ_m, γ_1 and γ_2 are regularization parameters corresponding to L^2- and H^1-regularizations, respectively, and $\kappa > 0$ is the diffusion coefficient. The boundary $\partial\Omega$ is split into disjoint parts Γ_l, Γ_r, Γ and Γ_m as shown in Figure 3.1 (left). The velocity field, $\mathbf{v}$, is computed by solving the steady Navier-Stokes equation with the side walls driving the flow (see the sketch on the right in Fig. 3.1):

$$-\frac{1}{\mathrm{Re}}\Delta\mathbf{v} + \nabla q + \mathbf{v}\cdot\nabla\mathbf{v} = 0 \quad \text{in } \Omega, \tag{3.2a}$$

$$\nabla\cdot\mathbf{v} = 0 \quad \text{in } \Omega, \tag{3.2b}$$

$$\mathbf{v} = \mathbf{g} \quad \text{on } \partial\Omega, \tag{3.2c}$$

where q is pressure, Re is the Reynolds number, and $\mathbf{g} = (g_1, g_2)^\top = 0$ on $\partial\Omega$ but $g_2 = 1$ on Γ_l and $g_2 = -1$ on Γ_r.

The inverse problem (3.1) has several different properties compared to the elliptic parameter estimation problem (2.1). First, the state equation is time-dependent; second, the inversion is based on boundary data only; third, the inversion is for an initial condition rather than a coefficient field; forth, both L^2-regularization and H^1-regularization for the initial condition can be used; and, fifth, the adjoint operator (i.e., the operator in the adjoint equation) is different from the operator in the state

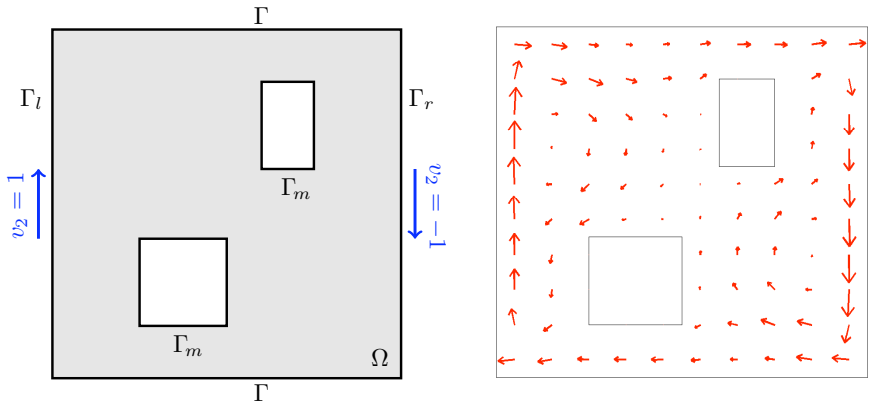

FIG. 3.1. *Left: Sketch of domain for the advective-diffusive inverse transport problem* (3.1). *Right: The velocity field* $\boldsymbol{v}$ *computed from the solution of the Navier-Stokes equation* (3.2) *with* $\mathrm{Re} = 100$.

equation (3.1b) since the advection operator is not self-adjoint. To compute the optimality system, we use the Lagrangian function

$$\mathscr{L}(u, u_0, p, p_0) := J(u_0) + \int_0^T \int_\Omega (u_t + \mathbf{v} \cdot \nabla u) p \, dx \, dt + \int_0^T \int_\Omega \kappa \nabla u \cdot \nabla p \, dx \, dt + \int_\Omega (u(0) - u_0) p_0 \, dx.$$

The optimality conditions for (3.1) are obtained by setting variations of the Lagrangian with respect to all variables to zero. Variations with respect to p and p_0 reproduce the state equation (3.1b)–(3.1d). The variation with respect to u in a direction $\tilde{u}$ is

$$\mathscr{L}_u(u, u_0, p, p_0)(\tilde{u}) = \int_{T_1}^T \int_{\Gamma_m} (u - u_d) \tilde{u} \, dx \, dt + \int_0^T \int_\Omega (\tilde{u}_t + \mathbf{v} \cdot \nabla \tilde{u}) p \, dx \, dt + \int_0^T \int_\Omega \kappa \nabla \tilde{u} \cdot \nabla p \, dx \, dt + \int_\Omega \tilde{u}(0) p_0 \, dx.$$

Partial integration in time for the term $\tilde{u}_t p$ and in space for $(\mathbf{v} \cdot \nabla \tilde{u}) p = \nabla \tilde{u} \cdot (\mathbf{v} p)$ and $\kappa \nabla \tilde{u} \cdot \nabla p$ results in

$$\mathscr{L}_u(u, u_0, p, p_0)(\tilde{u}) = \int_{T_1}^T \int_{\Gamma_m} (u - u_d) \tilde{u} \, dx \, dt + \int_0^T \int_\Omega (-\tilde{p}_t - \nabla \cdot (\mathbf{v} p) - \kappa \Delta p) \tilde{u} \, dx \, dt + \int_\Omega \tilde{u}(T) p(T) - \tilde{u}(0) p(0) + \tilde{u}(0) p_0 \, dx + \int_0^T \int_{\partial\Omega} (\mathbf{v} p + \kappa \nabla p) \cdot \mathbf{n} \tilde{u} \, dx \, dt.$$

Since at a stationary point the variation vanishes for arbitrary $\tilde{u}$, we obtain $p_0 = p(0)$, as well as the following strong form of the adjoint system (where we assume that the variables are sufficiently regular for integration by parts)

$$-p_t - \nabla \cdot (p\mathbf{v}) - \kappa \Delta p = 0 \qquad \text{in } \Omega \times [0, T], \tag{3.3a}$$

$$p(T) = 0 \qquad \text{in } \Omega, \tag{3.3b}$$

$$(\mathbf{v} p + \kappa \nabla p) \cdot \mathbf{n} = -(u - u_d) \qquad \text{on } \Gamma_m \times [T_1, T], \tag{3.3c}$$

$$(\mathbf{v} p + \kappa \nabla p) \cdot \mathbf{n} = 0 \qquad \text{on } (\partial\Omega \times [0, T]) \setminus (\Gamma_m \times [T_1, T]). \tag{3.3d}$$

Note that (3.3) is a final value problem, since p is given at $t = T$ rather than at $t = 0$. Thus, (3.3) has to be solved backwards in time, which amounts to the solution of an advection-diffusion equation with velocity $-\mathbf{v}$. Finally, the variation of the Lagrangian with respect to the initial condition u_0 in direction $\tilde{u}_0$ is

$$\mathscr{L}_{u_0}(u, u_0, p, p_0)(\tilde{u}_0) = \int_\Omega \gamma_1 u_0 \tilde{u}_0 + \gamma_2 \nabla u_0 \cdot \nabla \tilde{u}_0 - p_0 \tilde{u}_0 \, dx.$$

This variation vanishes for all $\tilde{u}_0$, if

$$\nabla \cdot (\gamma_2 \nabla u_0) + \gamma_1 u_0 - p_0 = \nabla \cdot (\gamma_2 \nabla u_0) + \gamma_1 u_0 - p(0) = 0 \tag{3.4}$$

holds, combined with homogeneous Neumann conditions for u_0 on all boundaries. Note that the optimality system (3.1b)–(3.1d), (3.3) and (3.4) is affine and thus a single Newton iteration is sufficient for its solution. The system can be solved using a conjugate gradient method for the unknown initial condition. The discretization of this initial value problem is discussed next. Its implementation is summarized in Appendix A.3 and listed in Appendix B.3.

3.1. Discretization. To highlight some aspects of the discretization, we introduce the linear solution and measurement operators $\mathcal{S}$ and $\mathcal{Q}$. For a given initial condition u_0 we denote the solution (in space and time) of (3.1) by $\mathcal{S}u_0$. The measurement operator $\mathcal{Q}$ is the trace on $\Gamma_m \times [T_1, T]$, such that the measurement data for an initial condition u_0 can be written as $\mathcal{Q}\mathcal{S}u_0$. With this notation, the adjoint state variable at time $t = 0$ becomes $p(0) = \mathcal{S}^\star \mathcal{Q}^\star(-(\mathcal{Q}\mathcal{S}u_0 - u_d))$, where "$\star$" denotes the adjoint operator. Using this equation in the control equation, (3.4) results in

$$\mathcal{S}^\star \mathcal{Q}^\star \mathcal{Q}\mathcal{S}u_0 + \gamma_1 u_0 + \nabla \cdot (\gamma_2 \nabla u_0) = \mathcal{S}^\star \mathcal{Q}^\star u_d. \tag{3.5}$$

Note that the operator on the left hand side in (3.5) is symmetric, and it is positive definite if $\gamma_1 > 0$ or $\gamma_2 > 0$.

We use the finite element method for the spatial discretization and the implicit Euler scheme for the discretization in time. To ensure that the discretization of the matrix corresponding to the linear operator on the left hand side in (3.5) is symmetric, we discretize the optimization problem and compute the corresponding discrete optimality conditions, which is often referred to as discretize-then-optimize. This approach is sketched next. The discretized cost function is

$$\min_{\boldsymbol{u}_0} \mathbf{J}(\boldsymbol{u}_0) := \frac{1}{2}(\bar{\boldsymbol{u}} - \bar{\boldsymbol{u}}_d)^T \bar{\mathbf{Q}}(\bar{\boldsymbol{u}} - \bar{\boldsymbol{u}}_d) + \frac{\gamma_1}{2} \boldsymbol{u}_0^T \tilde{\mathbf{M}} \boldsymbol{u}_0 + \frac{\gamma_2}{2} \boldsymbol{u}_0^T \mathbf{R} \boldsymbol{u}_0, \tag{3.6}$$

where $\bar{\boldsymbol{u}} = [\boldsymbol{u}^0 \, \boldsymbol{u}^1 \ldots \boldsymbol{u}^N]^T$ corresponds to the space-time discretization of u (N is the number of time steps and $\boldsymbol{u}^i$ are the spatial degrees of freedom at the i-th time step), which satisfies the discrete forward problem $\bar{\mathbf{S}}\bar{\boldsymbol{u}} = \bar{\boldsymbol{f}}$. Here, $\bar{\boldsymbol{u}}_d$ are the discrete space-time measurement data, $\boldsymbol{u}_0$ is the initial condition, and $\bar{\mathbf{Q}}$ is the discretized (space-time) measurement operator, *i.e.*, $\bar{\mathbf{Q}}$ is a block diagonal matrix with $\Delta t \mathbf{Q}$ (Δt denotes the time step size and $\mathbf{Q}$ the discrete trace operator) as diagonal block for time steps in $[T_1, T]$, and zero blocks else. Moreover, $\tilde{\mathbf{M}}$ and $\mathbf{R}$ are the matrices corresponding to the integration scheme used for the regularization terms, and $\bar{\boldsymbol{f}} = [\mathbf{M}\boldsymbol{u}_0 \, \mathbf{0} \ldots \mathbf{0}]^T$,

where $\mathbf{M}$ is the mass matrix. The discrete forward operator, $\bar{\mathbf{S}}$, is

$$\bar{\mathbf{S}} = \begin{bmatrix} \mathbf{M} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{-M} & \mathbf{L} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{-M} & \mathbf{L} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{L} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{-M} & \mathbf{L} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{-M} & \mathbf{L} \end{bmatrix}, \tag{3.7}$$

where $\mathbf{L} := \mathbf{M} + \Delta t \mathbf{N}$, with $\mathbf{N}$ being the discretization of the advection-diffusion operator. The discrete Lagrangian for (3.6) is

$$\mathcal{L}(\bar{\boldsymbol{u}}, \boldsymbol{u}_0, \bar{\boldsymbol{p}}) := \mathbf{J}(\boldsymbol{u}_0) + \bar{\boldsymbol{p}}^T(\bar{\mathbf{S}}\bar{\boldsymbol{u}} - \bar{\boldsymbol{f}}),$$

where $\bar{\boldsymbol{p}} = [\boldsymbol{p}^0 \ldots \boldsymbol{p}^N]^T$ is the discrete (space-time) Lagrange multiplier. Thus, the discrete adjoint equation is

$$\mathcal{L}_{\bar{\boldsymbol{u}}}(\bar{\boldsymbol{u}}, \boldsymbol{u}_0, \bar{\boldsymbol{p}}) = \bar{\mathbf{S}}^T\bar{\boldsymbol{p}} + \bar{\mathbf{Q}}(\bar{\boldsymbol{u}} - \bar{\boldsymbol{u}}_d) = 0, \tag{3.8}$$

and the discrete control equation is

$$\mathcal{L}_{\boldsymbol{u}_0}(\bar{\boldsymbol{u}}, \boldsymbol{u}_0, \bar{\boldsymbol{p}}) = \gamma_1\tilde{\mathbf{M}}\boldsymbol{u}_0 + \gamma_2\mathbf{R}\boldsymbol{u}_0 - \mathbf{M}\boldsymbol{p}^0 = 0. \tag{3.9}$$

Since (3.8) involves the block matrix $\bar{\mathbf{S}}^T$, the discrete adjoint equation is a backwards-in-time implicit Euler discretization of its continuous counterpart. From the last row in (3.8) we obtain

$$\mathbf{L}\boldsymbol{p}^N = -\Delta t\mathbf{Q}(\boldsymbol{u}^N - \boldsymbol{u}_d^N) \tag{3.10}$$

as the discretization of the homogeneous terminal conditions (3.3). Discretizing (3.8) simply by $\boldsymbol{p}^N = \mathbf{0}$ rather than (3.10) does not result in a symmetric discretization for the left hand side of (3.5). Such an inconsistent discretization means that the conjugate gradient method will likely not converge, which shows the importance of a discretization that has an underlying discrete optimization problem, as guaranteed in a discretize-then-optimize approach. Note that as $\Delta t \to 0$, the discrete condition (3.10) tends to its continuous counterpart. The system (3.9) (with $\boldsymbol{p}^0$ computed by solving the state and adjoint eqations) is solved using the conjugate gradient method for the unknown initial condition.

3.2. Numerical tests. Next, we present numerical tests for the initial condition inversion problem (3.1), which is solved with the conjugate gradient method. To illustrate properties of the forward problem, Figure 3.2 shows three time instances of the field u, using the advective velocity $\mathbf{v}$ from Figure 3.1. Note that the diffusion quickly blurs the initial condition. Since the discretization uses standard finite elements, a certain amount of physical diffusion (*i.e.*, κ cannot be too small) is necessary for numerical stability of the advection-diffusion equation. For advection-dominated flows, a stabilized finite element methods (such as SUPG; see [8]) has to be used.

The solutions of the inverse problem for various choices of regularization parameters are shown in Figure 3.3. In the left and middle plots, we show the recovered initial condition with L^2-type regularization, while the right plot shows the inversion

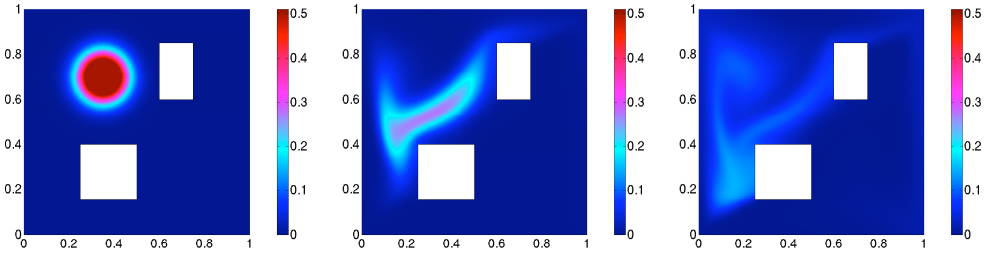

FIG. 3.2. *Forward advective-diffusive transport at initial time $t = 0$ (left), at $t = 2$ (center), and at final time $t = 4$ (right).*

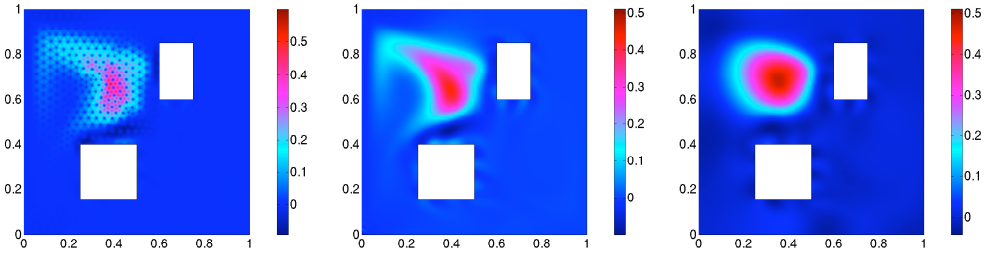

FIG. 3.3. *The recovered initial condition u_0 for $\tilde{\boldsymbol{M}} = \boldsymbol{I}$, $\gamma_1 = 10^{-5}$ and $\gamma_2 = 0$ (left), for $\tilde{\boldsymbol{M}} = \boldsymbol{M}$, $\gamma_1 = 10^{-2}$ and $\gamma_2 = 0$ (center), and for $\gamma_1 = 0$, $\gamma_2 = 10^{-6}$ (right). Other parameters are $\kappa = 0.001$, $T_1 = 1$, and $T = 4$.*

results obtained with H^1-regularization. For these examples, quadratic finite elements in space with 3889 degrees of freedom, and 20 implicit time steps for the time discretization were used, *i.e.*, the state is discretized with overall 77780 degrees of freedom. The CG iterations are terminated when the relative residual drops below 10^{-4}, which requires 32 iterations for the regularization with the identity matrix (left plot in Figure 3.3), 43 iterations for the L^2-regularization with the mass matrix (middle plot) and 157 iterations for the H^1-regularization (right plot). Figure 3.3 shows that the L^2-type regularization with the identity matrix allows spurious oscillations in the reconstruction, since these high-frequency components correspond to very small (or zero) eigenvectors of the misfit Hessian as well as the regularization. The smoothing effect of the H^1-regularization prevents this behavior and leads to a much improved reconstruction. Since the measurements are restricted to Γ_m, they do not provide information on the initial condition in the upper left part of the domain. Thus, in that part the reconstruction of the initial condition is controlled by the regularization only, which explains the significant differences for the different regularizations. Finally, note that the reconstructed initial concentrations also contain negative values. This unphysical behavior could be avoided by enforcing the bound constraint $u_0 \geq 0$ in the inversion procedure.

4. Source terms inversion in a nonlinear elliptic problem. As our last model problem we consider the estimation of the volume and boundary source in a nonlinear elliptic equation. We assume a situation where only point measurements are available, which results in a problem formulated as

$$\min_{f,g} J(f,g) := \frac{1}{2}\int_\Omega (u-u_d)^2 b(x)\,dx + \frac{\gamma_1}{2}\int_\Omega |\nabla f|^2\,dx + \frac{\gamma_2}{2}\int_\Gamma g^2\,dx, \tag{4.1a}$$

where u is the solution of

$$-\Delta u + u + cu^3 = f \qquad \text{in } \Omega, \tag{4.1b}$$

$$\nabla u \cdot \mathbf{n} = g \qquad \text{on } \Gamma, \tag{4.1c}$$

where $\Omega \subset \mathbb{R}^n$, $n \in \{1, 2, 3\}$ is an open, bounded domain with boundary $\Gamma := \partial\Omega$. The constant $c \geq 0$ controls the amount of nonlinearity in the state equation (4.1b) and $b(x)$ denotes the point measurement operator defined by

$$b(x) = \sum_{j=1}^{N_r} \delta(x - x_j) \text{ for } j = 1, ..., N_r, \tag{4.2}$$

where N_r denotes the number of point measurements, and $\delta(x - x_j)$ is the Dirac delta function. Moreover, $f \in H^1(\Omega)$ and $g \in L^2(\Gamma)$ are the source terms, $\gamma_1 > 0$ and $\gamma_2 > 0$ the regularization parameters, and $\boldsymbol{n}$ is the outwards normal for Γ. Note that while the notation in (4.1) suggests that u_d is a function given on all of Ω, due to the definition of b only the point data $u_d(x_j)$ are needed. We assume that the domain Ω is sufficiently smooth, which implies that the solution of (4.1b) and (4.1c) is sufficiently regular for the point evaluation to be well defined.

Problem (4.1) (we refer to [9] for a similar problem) is used to demonstrate features of inverse problems that are not present in the elliptic parameter estimation problem (Section 2) or the initial-time inversion (Section 3). Namely, the state equation is nonlinear, we invert for sources rather than for a coefficient, and the inversion is for two fields, for which different regularizations are used. Moreover, the inversion is based on discrete point rather than distributed or boundary measurements.

The computation of the optimality system for (4.1) is based on the Lagrangian functional

$$\mathscr{L}(u, f, g, p) := J(u, f, g) + (\nabla u, \nabla p) + (u + cu^3 - f, p) - (g, p)_\Gamma,$$

where $(\cdot\,, \cdot)_\Gamma$ denotes the L^2-product over the boundary Γ, and, as before, $(\cdot\,, \cdot)$ is the L^2-product over Ω. We compute the variations of the Lagrangian with respect to all variables and set them to zero to derive the (necessary) optimality conditions. This results in the weak form of the first-order optimality system:

$$0 = \mathscr{L}_p(u, f, g, p)(\tilde{p}) = (\nabla u, \nabla \tilde{p}) + (u + cu^3 - f, \tilde{p}) - (g, \tilde{p})_\Gamma, \tag{4.3a}$$

$$0 = \mathscr{L}_u(u, f, g, p)(\tilde{u}) = (\nabla p, \nabla \tilde{u}) + ((1 + 3cu^2)p, \tilde{u}) + ((u - u_d)b(x), \tilde{u}), \tag{4.3b}$$

$$0 = \mathscr{L}_f(u, f, g, p)(\tilde{f}) = \gamma_1(\nabla f, \nabla \tilde{f}) - (p, \tilde{f}), \tag{4.3c}$$

$$0 = \mathscr{L}_g(u, f, g, p)(\tilde{g}) = (\gamma_2 g - p, \tilde{g})_\Gamma, \tag{4.3d}$$

for all variations $(\tilde{u}, \tilde{f}, \tilde{g}, \tilde{p})$. Invoking Green's (first) identity where needed, and rearranging terms, the strong form for this system is as follows:

$$-\Delta u + u + cu^3 = f \qquad \text{in } \Omega, \qquad \text{(state equation)} \tag{4.4a}$$

$$\nabla u \cdot \mathbf{n} = g \qquad \text{on } \Gamma,$$

$$-\Delta p + (1 + 3cu^2)p = (u_d - u)b(x) \qquad \text{in } \Omega, \qquad \text{(adjoint equation)} \tag{4.4b}$$

$$\nabla p \cdot \boldsymbol{n} = 0 \qquad \text{on } \Gamma,$$

$$-\nabla \cdot (\gamma_1 \nabla f) = p \qquad \text{in } \Omega, \qquad (f\text{-gradient}) \tag{4.4c}$$

$$\nabla f \cdot \boldsymbol{n} = 0 \qquad \text{on } \Gamma,$$

$$\gamma_2\, g - p = 0 \qquad \text{in } \Gamma. \qquad (g\text{-gradient}) \tag{4.4d}$$

We solve the optimality system (4.3) (or equivalently (4.4)) using the Gauss-Newton method described in Section 2.3. After discretizing and taking variations of (4.3a)–(4.3d) with respect to (u, f, g, p) we obtain the Gauss-Newton step (where we assume that the state and adjoint equations have been solved exactly and thus $\boldsymbol{g}_u = \boldsymbol{g}_p = 0$)

$$\begin{bmatrix} \mathbf{B} & \mathbf{0} & \mathbf{0} & \mathbf{A}^T \\ \mathbf{0} & \mathbf{R}_1 & \mathbf{0} & \mathbf{C}_1^T \\ \mathbf{0} & \mathbf{0} & \mathbf{R}_2 & \mathbf{C}_2^T \\ \mathbf{A} & \mathbf{C}_1 & \mathbf{C}_2 & \mathbf{0} \end{bmatrix} \begin{bmatrix} \hat{\boldsymbol{u}}_k \\ \hat{\boldsymbol{f}}_k \\ \hat{\boldsymbol{g}}_k \\ \hat{\boldsymbol{p}}_k \end{bmatrix} = - \begin{bmatrix} \mathbf{0} \\ \boldsymbol{g}_f \\ \boldsymbol{g}_g \\ \mathbf{0} \end{bmatrix}. \tag{4.5}$$

Here, $\mathbf{B}$ is the matrix corresponding to the point measurements, $\mathbf{R}_1$ and $\mathbf{R}_2$ are stiffness and boundary mass matrices corresponding to the regularization for f and g, respectively, and $\mathbf{A}$, $\mathbf{C}_1$ and $\mathbf{C}_2$ are the Jacobians of the state equation with respect to the state variables, and of the adjoint equation with respect to both control variables, respectively. With $\boldsymbol{g}_f$ and $\boldsymbol{g}_g$ we denote the discrete gradients for f and g, respectively. Finally, $\hat{\boldsymbol{u}}_k$, $\hat{\boldsymbol{f}}_k$, $\hat{\boldsymbol{g}}_k$ and $\hat{\boldsymbol{p}}_k$ are the search directions for the state, control (with respect to f and g), and adjoint variables, respectively. To compute the right hand side in (4.5), we solve the (nonlinear) state and adjoint equations given by equations (4.4a) and (4.4b), respectively, for iterates f_k and g_k. To obtain the Gauss-Newton system in the inversion variables only, we eliminate the blocks corresponding to the Newton updates $\hat{\boldsymbol{u}}$ and $\hat{\boldsymbol{p}}$ and obtain

$$\begin{aligned} \hat{\boldsymbol{u}}_k &= -\mathbf{A}^{-1}(\mathbf{C}_1\, \hat{\boldsymbol{f}}_k + \mathbf{C}_2\, \hat{\boldsymbol{g}}_k), \\ \hat{\boldsymbol{p}}_k &= -\mathbf{A}^{-T}\mathbf{B}\hat{\boldsymbol{u}}_k. \end{aligned}$$

Thus, the reduced (Gauss-Newton) Hessian becomes

$$\mathbf{H} = \begin{bmatrix} \mathbf{R}_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{R}_2 \end{bmatrix} + \begin{bmatrix} \mathbf{C}_1^T \\ \mathbf{C}_2^T \end{bmatrix} \mathbf{A}^{-T}\mathbf{B}\mathbf{A}^{-1} \begin{bmatrix} \mathbf{C}_1 & \mathbf{C}_2 \end{bmatrix}, \tag{4.7}$$

and the reduced linear system reads

$$\mathbf{H} \begin{bmatrix} \hat{\boldsymbol{f}}_k \\ \hat{\boldsymbol{g}}_k \end{bmatrix} = - \begin{bmatrix} \boldsymbol{g}_f \\ \boldsymbol{g}_g \end{bmatrix}.$$

This symmetric positive system is solved by the preconditioned conjugate gradient method, where a simple preconditioner is given by the inverse of the regularization operator (the first block matrix in (4.7)).

4.1. Numerical tests. Here, we present some numerical results for the nonlinear inverse problem (4.1). The upper row in Figure 4.1 shows the noisy measurement data (left; only the data at the points is used in the inversion), the "true" volume source f (middle) and boundary source g (right) used to construct the synthetic data. The lower row depicts the results of the inversions for the regularization parameters $\gamma_1 = 10^{-5}$ and $\gamma_2 = 10^{-4}$. The middle and right plots on the same row show the reconstruction for f and g, and the left plot shows the state solution corresponding to these reconstructions. Note that the regularization for the volume source f leads to a smooth reconstruction. The L^2-regularization for the boundary source g favors reconstructions with small L^2-norm but does not prevent oscillations.

The optimality system (4.3) is solved using an inexact Gauss-Newton-CG method as described in Section 2.6. The inversion is based on 56 measurement points (out of

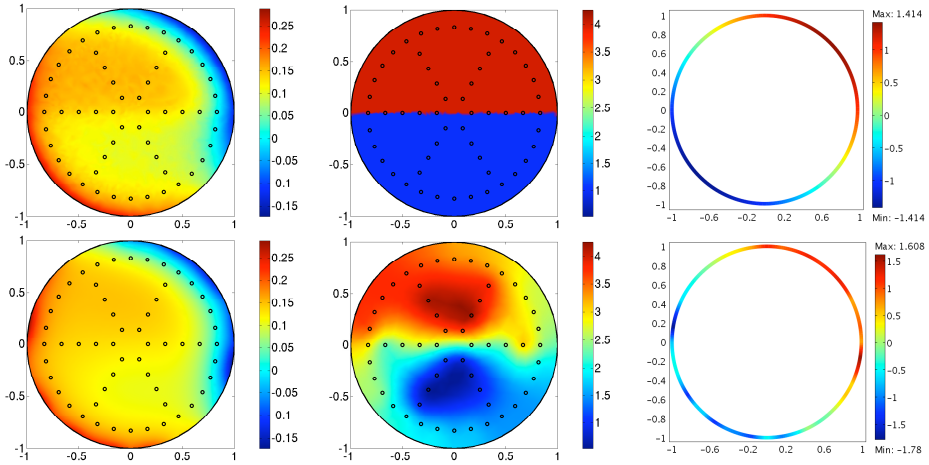

FIG. 4.1. *Results for the nonlinear elliptic source inversion problem with* $c = 10^3$, $\gamma_1 = 10^{-5}$ *and* $\gamma_2 = 10^{-4}$, *and* 1% *noise in the synthetic data. Noisy data* u_d *and recovered solution* u *(left column), "true" volume source and recovered volume source* f *(center column), and "true" boundary source and recovered boundary source* g *(right column).*

which more than half are located near the boundary to facilitate the inversion for the boundary field). The mesh consisted of 1206 triangular elements, and we used linear Lagrange elements for the discretization of the source fields and quadratic elements for the state and the adjoint. The iterations are terminated when the L^2-norms of the f-gradient ($\boldsymbol{g}_f$) and the g-gradient ($\boldsymbol{g}_g$) drop below 10^{-7}. For this particular example, the number of Gauss-Newton iterations was 11, and the total number of CG iterations (with an adaptive tolerance for the CG solve ranging from 5×10^{-1} to 6×10^{-4}) was 177. This amounted to a total number of 11 nonlinear state and linear adjoint solves (one at each Gauss-Newton iteration), and 177 (linear) state-adjoint solves (one at each CG iteration).

5. Extensions and other frequently occurring features. The model problems used in this report illustrate several, but certainly not all important features that arise in inverse problems with PDEs. A few more typical properties that are common in inverse problems governed by PDEs, which have not been covered by our model problems are mentioned next.

If the inversion field is expected to have discontinuities but is otherwise piecewise smooth, the use of non-quadratic regularization is preferable over the quadratic gradient regularization used in (2.1). The total variation regularization

$$TV(a) := \int_\Omega |\nabla a| \, dx$$

for a parameter field a defined on Ω can, in this case, result in improved reconstructions since $TV(a)$ is finite at discontinuities, while quadratic gradient regularization becomes infinite at discontinuities. Thus, with quadratic gradient regularization discontinuities are smoothed, while they often can be reconstructed when using $TV(\cdot)$ as regularization.

Another frequently occurring aspect in inverse problems is that data from multiple experiments is available, which amounts to an optimization problem with several PDE

constraints (each PDE corresponding to an experiment), and makes the computation of first- and second-order derivatives costly.

Finally, we mention that many inverse problems involve vector systems as state equations, which can make the derivation of the corresponding optimality systems more involved compared to the scalar equations used in our model problems.

Appendix A. Implementation of model problems. In this appendix our implementations for solving the model problems presented in this paper are summarized. While in the following we use the COMSOL Multiphysics (v3.5a) [10] finite element package (and the MATLAB syntax), the implementations will be similar in other toolkits. In this section, we describe parts of the implementation in detail. Complete code listings are provided in Appendix B.

A.1. Steepest descent method for linear elliptic inverse problem. Our implementation starts with specifying the geometry and the mesh (lines 2 and 3), and with defining names for the finite element basis functions, their polynomial order and the regularization parameter (lines 4–8). In this example, we use linear elements on a hexahedral mesh for the coefficient a (and for the gradient), and quadratic finite element functions for the state u, the adjoint p and the desired state u_d. The latter is used to compute and store the synthetic measurements, which are computed from a given coefficient `atrue` (defined in line 6).

```
2 fem.geom = rect2(0,1,0,1);
3 fem.mesh = meshmap(fem,'Edgelem',{1,20,2,20});
4 fem.dim = {'a' 'grad' 'p' 'u' 'ud'};
5 fem.shape = [1 1 2 2 2];
6 fem.equ.expr.atrue = '1+7*(sqrt((x-0.5)^2+(y-0.5)^2)>0.2)';
7 fem.equ.expr.f = '1';
8 fem.equ.expr.gamma = '1e-9';
```

Homogeneous Dirichlet boundary conditions for the state and adjoint equations are used. In line 14 the conditions $u = 0$, $p = 0$ and $u_d = 0$ on $\partial\Omega$ are enforced. The weak form for the state equation with the target coefficient `atrue`, which is used to compute the synthetic measurements, is given in line 15, followed by the state equation with the unknown, to-be-reconstructed coefficient a (line 16). Lines 17–19 define the weak forms of the adjoint and the gradient equations, respectively. Note that in COMSOL Multiphysics, variations (or test functions) are denoted by adding "`_test`" to the variable name.

```
14 fem.bnd.r = {{'u' 'p' 'ud'}};
15 fem.equ.expr.goal = '-(atrue*(udx*udx_test+udy*udy_test)-f*ud_test)';
16 fem.equ.expr.state = '-(a*(ux*ux_test+uy*uy_test)-f*u_test)';
17 fem.equ.expr.adjoint = '-(a*(px*px_test+py*py_test)-(ud-u)*p_test)';
18 fem.equ.expr.control = ['(grad*grad_test-gamma*(ax*gradx_test '...
19                         '+ay*grady_test)-(px*ux+py*uy)*grad_test)'];
```

To synthesize measurement data, the state equation with the given coefficient `atrue` is solved (lines 20–22).

```
20 fem.equ.weak = 'goal';
21 fem.xmesh = meshextend(fem);
22 fem.sol = femlin(fem, 'Solcomp', {'ud'});
```

COMSOL allows the user to access its internal finite element structures such as the degrees of freedom for each finite element function. Our implementation of the steepest descent iteration works on the finite element coefficients, and the indices for the

degrees of freedom are extracted from the finite element data structure in the lines 23–29. Note that internally the unknowns are ordered alphabetically (independently from the order given in line 4). Thus, the indices for the finite element function a can be extracted from the first column of `dofs`; see line 25. If different order element functions are used in the same finite element structure (in the present case, linear and quadratic polynomials), COMSOL pads the list of indices for the lower-order function with zeros. These zero indices are removed by only choosing the positive indices (lines 25–29). The index vectors are used to access the entries in `X`, the vector containing all finite element coefficients, which can be accessed as shown in line 30.

```
23 nodes = xmeshinfo(fem ,'out', 'nodes');
24 dofs = nodes.dofs';
25 AI = dofs(dofs(:,1)>0,1);
26 GI = dofs(dofs(:,2)>0,2);
27 PI = dofs(dofs(:,3)>0,3);
28 UI = dofs(dofs(:,4)>0,4);
29 UDI = dofs(dofs(:,5)>0,5);
30 X = fem.sol.u;
```

We add noise to our synthetic data (line 32) to lessen the "inverse crime" [18], which occurs due to the fact that the same numerical method is used in the inversion as for creating the synthetic data.

```
32 X(UDI) = X(UDI) + datanoise * max(abs(X(UDI))) * randn(length(UDI),1);
```

We initialize the coefficient a (line 33; the initialization is a constant function) and (re-)define the weak form as the sum of the state, adjoint, and control equations (line 34). Note that, since the test functions for these three weak forms differ, one can regain the individual equations by setting the appropriate test functions to zero. To compute the initial value of the cost functional, in line 36 we solve the system with respect to u only. For the variables not solved for, the finite element functions specified in `X` are used. Then, the solution of the state equation is copied into `X`, and is used in the evaluation of the cost functional (lines 37 and 38).

```
33 X(AI) = 8.0;
34 fem.equ.weak = 'state + adjoint + control';
35 fem.xmesh = meshextend(fem);
36 fem.sol = femlin(fem, 'Solcomp', {'u'}, 'U', X);
37 X(UI) = fem.sol.u(UI);
38 cost_old = evaluate_cost (fem, X);
```

Next, we iteratively update a in the steepest descent direction. For a current iterate of the coefficient a, we first solve the state equation (line 40) for u. Given the state solution, we solve the adjoint equation (line 42) and compute the gradient from the control equation (line 44). A line search to satisfy the Armijo descent criterion [22] is used (line 56). If, for a step length α, the cost is not sufficiently decreased, backtracking is performed, *i.e.*, the step length is reduced (line 61). In each backtracking step, the state equation has to be solved to evaluate the cost functional (lines 53-54).

```
39 for iter = 1:maxiter
40   fem.sol = femlin(fem, 'Solcomp', {'u'}, 'U', X);
41   X(UI) = fem.sol.u(UI);
42   fem.sol = femlin(fem, 'Solcomp', {'p'}, 'U', X);
43   X(PI) = fem.sol.u(PI);
44   fem.sol = femlin(fem, 'Solcomp', {'grad'}, 'U', X);
45   X(GI) = fem.sol.u(GI);
46   grad2 = postint (fem, 'grad * grad');
```

```
47    Xtry = X;
48    alpha = alpha * 1.2;
49    descent = 0;
50    no_backtrack = 0;
51    while (~descent && no_backtrack < 10)
52        Xtry(AI) = X(AI) - alpha * X(GI);
53        fem.sol = femlin(fem, 'Solcomp', {'u'}, 'U', Xtry);
54        Xtry(UI) = fem.sol.u(UI);
55        [cost, misfit, reg] = evaluate_cost (fem, Xtry);
56        if (cost < cost_old - alpha * c * grad2)
57            cost_old = cost;
58            descent = 1;
59        else
60            no_backtrack = no_backtrack + 1;
61            alpha = 0.5 * alpha;
62        end
63    end
64    X = Xtry;
65    fem.sol = femsol(X);
66    if (sqrt(grad2) < tol && iter > 1)
67         fprintf (['Gradient method converged after %d iterations.'...
68                   '\n\n'], iter);
69    break;
70    end
71  end
```

The steepest descent iteration is terminated when the norm of the gradient is sufficiently small. The complete code listing of the implementation of the gradient method applied to solve problem (2.1) can be found in Appendix B.1.

A.2. Gauss-Newton CG method for linear elliptic inverse problem. Differently from the implementation of the steepest descent method, which relies to a large extent on solvers provided by the finite element package, this implementation makes explicit use of the discretized operators corresponding to the state and the adjoint equations. For brevity of the description, we skip steps that are analogous to the steepest descent method and refer to Appendix B.2 for a complete listing of the implementation.

After setting up the mesh, the finite element functions a, u and p corresponding to the coefficient, state and adjoint variables, as well as their increments $\hat{a}, \hat{u}$ and $\hat{p}$, are defined in lines 4 and 5.

```
4  fem.dim = {'a' 'a0' 'delta_a' 'delta_p' 'delta_u' 'p' 'u' 'ud'};
5  fem.shape = [1 1 1 2 2 2 2 2];
```

Homogeneous Dirichlet conditions are used for the state and adjoint variable, as well as for their increment functions; see line 16. After initializing parameters, the weak forms for the construction of the synthetic data (lines 17 and 18), and the weak forms for the Gauss-Newton system are defined (lines 19–24).

```
16 fem.bnd.r = {{'delta_u''delta_p' 'u' 'ud'}};
17 fem.equ.expr.goal = '-(atrue*(udx*udx_test+udy*udy_test)-f*ud_test)';
18 fem.equ.expr.state = ['-(a*(ux*ux_test+uy*uy_test)-f*u_test)'];
19 fem.equ.expr.incstate = ['-(a*(delta_ux*delta_px_test+delta_uy '...
20         '*delta_py_test)+delta_a*(ux*delta_px_test+uy*delta_py_test))'];
21 fem.equ.expr.incadjoint = ['-(a*(delta_px*delta_ux_test+
22 delta_py*delta_uy_test)+delta_u*delta_u_test)'];
23 fem.equ.expr.inccontrol = ['-(gamma*(delta_ax*delta_ax_test+delta_ay '...
24         '*delta_ay_test)+(delta_px*ux+delta_py*uy)*delta_a_test)'];
```

As in the implementation of the steepest descent method, the synthetic data is based on a "true" coefficient `atrue`, and noise is added to the synthetic measurements; the corresponding finite element function is denoted by `ud`. The indices pointing to the coefficients for each finite element function in the coefficient vector `X` are stored in `AI,dAI,dPI,...`. After setting the coefficient a to a constant initial guess, the reduced gradient is computed, which amounts to the right hand side in the reduced Hessian equation. For that purpose, the state equation (lines 50–52) is solved.

```
50 fem.xmesh = meshextend(fem);
51 fem.sol = femlin(fem, 'Solcomp', {'u'}, 'U', X);
52 X(UI) = fem.sol.u(UI);
```

Next, the KKT system is assembled in line 57. Note that the system matrix `K` does not take into account Dirichlet boundary conditions. These conditions are enforced through the constraint equation `N*X=M` (where N and M are the left and right hand sides of the boundary conditions and can be returned by the `assemble` function).

```
56 fem.xmesh = meshextend(fem);
57 [K, N] = assemble(fem, 'U', X, 'out', {'K', 'N'});
```

Our implementation explicitly uses the individual blocks of the KKT system—these blocks are extracted from the KKT system using the index vectors `dAI, dUI, dPI`; see lines 58–61. Note that the choice of the test function influences the location of these blocks in the KKT system matrix. Since Dirichlet boundary conditions are not taken care of in `K` (and thus in the matrix `A`, which corresponds to the state equation), these constraints are enforced by a modification of `A` (see lines 58–61). The modification puts zeros in rows and columns of Dirichlet nodes and ones into the diagonals; see lines 62–68. Additionally, changes to the right hand sides are made using a vector `chi`, which contains zeros for Dirichlet degrees of freedom and ones in all other components (lines 69–70).

```
58 W = K(dUI,dUI);
59 A = K(dUI,dPI);
60 C = K(dPI,dAI);
61 R = K(dAI,dAI);
62 ind = find(sum(N(:, dUI),1)~=0);
63 A(:,ind) = 0;
64 A(ind,:) = 0;
65 for (k = 1:length(ind))
66   i = ind(k);
67   A(i,i) = 1;
68 end
69 chi = ones(size(A,1),1);
70 chi(ind) = 0;
```

An alternative approach to eliminate the degrees of freedom corresponding to Dirichlet boundary conditions is to use a null space basis of the constraints originating from the Dirichlet conditions (or other essential boundary conditions, such as periodic conditions). COMSOL's function `femlin` can be used to compute the matrix `Null`, whose columns form a basis of the null space of the constraint operator (*i.e.*, the left hand side matrix of the constraint equation `N*X=M`). For instance, the following lines show how to eliminate the Dirichlet boundary conditions and solve the adjoint problem using this approach.

```
X(PI) = 0; fem.sol = femsol(X);
[K, L, M, N] = assemble(fem, 'U', X);
```

```
[Ke,Le,Null,ud] = femlin('in',{'K' K(PI,PI) 'L' L(PI) 'M' M 'N'
  N(:,PI)});
X(PI) = Null * (Ke \ Le);
```

Note that the `assemble` function linearizes nonlinear weak forms at the provided linearization point (`X` in our case—the linearization point is zero if it is not provided). By setting the adjoint variable to zero, we avoid unwanted contributions to the linearized matrices. In the above code snippet, we use `femlin` to provide a basis of the constraint space, which allows us to compute the solution in the (smaller) constraint space. This solution `Ke\Le` is then lifted to the original space by multiplying it with `Null`.

After this side remark, we continue with the description of the Gauss-Newton-CG method description for the elliptic parameter estimation problem. Since the state operator `A` has to be inverted repeatedly, we compute its Choleski factorization beforehand; see line 71. Now, to compute the right hand side of the Gauss-Newton system, the adjoint equation can be solved by a simple forward and backward elimination (line 72). The resulting adjoint is used to compute the reduced gradient (line 73). Note that `MG` denotes the coefficients of the reduced gradient, multiplied by the mass matrix, which corresponds to the right hand side in the reduced Gauss-Newton equation (see equation (2.14) in Section 2.5).

```
71 AF = chol(A);
72 X(PI) = AF \ (AF' \ (chi.*(W * (X(UDI) - X(UI)))));
73 MG =  C' * X(PI) + R * X(AI) -  mu * (1./ (X(AI) - X(A0I)));
```

To solve the reduced Hessian system and obtain a descent direction, we use MATLAB's conjugate gradient function `pcg`. Thereby, the function `elliptic_apply_logb`, which is specified below, implements the application of the reduced Hessian to a vector. The right hand side in the system is given by the negative gradient multiplied by the mass matrix. We use a loose tolerance early in the CG iteration, and tighten the tolerance as the iterates get closer to the solution, as described in Section 2.6 (see line 80).

```
80 tolcg = min(0.5, sqrt(gradnorm/gradnorm_ini));
81 P = R + 1e-10*eye(length(AI));
82 [D, flag, relres, CGiter, resvec] = pcg (@(D)elliptic_apply_logb
83                  (V, chi, W, AF, C, R, X, AI, A0I, mu),-MG, tolcg, 300, P);
```

The vector `D` resulting from the (approximate) solution of the reduced Hessian system is used to update the coefficients of the FE function a. A line search with an Armijo criterion is used to globalize the Gauss-Newton method. In the presence of inequality constraints, we render the logarithmic barrier parameter `mu` to a positive value (*e.g.*, `mu=1e-10`) and check for constraint violations (lines 89-99). This is done by computing the maximum allowable step length in the line search to maintain `X(AI) > X(A0I)` (lines 97-98).

```
89 idx = find(D < 0);
90 Aviol = X(AI) - X(A0I);
91 if min(Aviol) <= 1e-9
92    error('point is not feasible');
93 end
94 if (isempty(idx))
95    alpha = 1;
96 else
97    alpha1 = min(-Aviol(idx)./D(idx));
98    alpha  = min(1, 0.9995*alpha1);
```

```
99 end
```

We conclude the description of the Gauss-Newton implementation with giving the function `elliptic_apply_logb` for both approaches to incorporate the Dirichlet boundary conditions. This function applies the reduced Hessian to a vector `D`. Note that, since the precomputed Choleski factor `AF` is triangular, solving the state and the adjoint equation only amounts to two forward and two backward elimination steps.

```
function GNV = elliptic_apply_logb(V, chi, W, AF, C, R, X, AI, A0I, mu)
du = AF \ (AF' \ (chi .* (-C * V)));
dp = AF \ (AF' \ (chi .* (-W * du)));
Z = mu * spdiags (1 ./ (X(AI) - X(A0I)).^2, 0, length(AI), length(AI));
GNV = C' * dp + (R + Z)* V;
```

The second approach, which uses the basis of the constraint null space `Null`, reads:

```
function GNV = elliptic_apply_logb(V, Null, W, AF, C, R, X, AI, A0I, mu)
du = AF \ (AF' \ (Null' * (-C * V)));
dp = AF \ (AF' \ (Null' * (-W * (Null * du))));
Z = mu * spdiags( 1 ./ (X(AI)-X(A0I)).^2, 0, length(AI), length(AI));
GNV = C' * (Null * dp) + (R + Z) * V;
```

A.3. Conjugate gradient method for inverse advective-diffusive transport. Here we summarize implementation aspects that are particular to the inverse advective-diffusive transport problem. The matrices corresponding to the stationary advection-diffusion operator and the mass matrix are assembled similarly as in the elliptic model problem. The discretization of the measurement operator $\mathcal{Q}$ is a mass matrix for the boundary Γ_m. To obtain this matrix, we set the weak form corresponding to Ω to zero (line 49), and use a mass matrix for the boundary Γ_m only (line 50).

```
49 fem.equ.weak = '0';
50 fem.bnd.weak =  {{} {} {} {'-u * u_test'} {}};
51 fem.xmesh = meshextend(fem);
52 MB = assemble(fem, 'out', 'K');
53 Q = MB(UI,UI);
```

The most interesting part of the implementation for this time-dependent problem is the application of the state-adjoint solve (see Section 3.1), which is discussed next. The function `ad_apply` that applies the left hand side from (3.5) to an input vector `U0` is listed below in lines 1–17. The time steps for the state variable `UU` and the adjoint `PP` are stored in columns, by `L` we denote the discrete advection-diffusion operator, and by `M` the domain mass matrix. The lines 5–8 correspond to the solution of the state equation, and lines 9–16 to the solution of the adjoint equation, which is solved backwards in time. The factor computed in line 12 controls if measurements are taken for time instances or not. Note that the adjoint equation is solved using the discrete adjoint scheme (in space and time) we described in Section 3.1.

```
1  function out = ad_apply(U0, L, M, R, Q, dt, Tm, ntime, gamma1, gamma2, ...
2      UU, PP)
3  global cgcount;
4  cgcount = cgcount + 1;
5  UU(:,1) = U0;
6  for k = 1:(ntime-1)
7      UU(:,k+1) = L \ (M * UU(:,k));
8  end
9  F = Q * (- dt * UU(:,ntime));
```

```
10 PP(:,ntime) = L' \ F;
11 for k = (ntime-1):-1:2
12     m = (k > Tm * ntime);
13     F = M * PP(:,k+1) + m * Q * (- dt * UU(:,k));
14     PP(:,k) = L' \ F;
15 end
16 MP(:,1) = M * PP(:,2);
17 out = - MP(:,1) + gamma1 * M * U0  +  gamma2 * R * U0;
```

A complete code listing for this problem can be found in Appendix B.3.

A.4. Gauss-Newton-CG for nonlinear source inversion problem. Some implementation details for the nonlinear inverse problem (4.1) are described next. Most parts of the code are discussed above, hence we focus on a few new aspects only.

We recall that the inversion in (4.1) is based on discrete measurement points rather than distributed measurements. The discretization of this operator is a mass matrix for the data points x_j, $j = 1, ..., N_r$. To obtain this matrix, we set the weak form corresponding to Ω and the boundary Γ to zero (lines 89-90) and add a mass matrix contribution for the data points (line 91).

```
89   fem.equ.weak = '0';
90   fem.bnd.weak = '0';
91   fem.pnt.weak =  {{},{'-p * p_test'}};
92   fem.xmesh = meshextend(fem);
93   K = assemble(fem, 'out', 'K');
94   B = K(PI,PI);
```

The second novelty in problem (4.1) is that we invert for two fields, f and g, where g is defined on the boundary. Therefore, the g-gradient (control) equation is defined on the boundary. In line 105, we define the weak forms corresponding to the state equation and the incremental equations. We note that the weak form of the equation for the gradient with respect to g is defined on the boundary (see line (106)). The remaining terms in this definition correspond to the weak forms of the second variation of the Lagrangian with respect to p and g, and the boundary weak term corresponding to the state equation.

```
105 fem.equ.weak = 'state + incstate + incadjoint + inccontrolf';
106 fem.bnd.weak = {{'-gamma2*delta_g*delta_g_test'...
107     '-delta_p*delta_g_test-delta_g*delta_p_test-g*u_test'}};
```

Since the state equation is nonlinear, we use COMSOL's built-in nonlinear solver `femnlin` (lines 109-111) to solve for the state variable u. Alternatively, one could implement a Newton method for the solution of the nonlinear (state) equation instead of relying on `femnlin`.

```
109 fem.xmesh = meshextend(fem);
110 fem.sol = femnlin(fem, 'Solcomp', {'u'}, 'U', X, 'ntol', 1e-9);
111 X(UI) = fem.sol.u(UI);
```

Next, we compute the Choleski factor of `A` (note that the state operator is always positive definite), the matrix corresponding to the state operator and use it to compute the adjoint solution by a forward and a backward elimination (line 124), where `B` is computed in line 94.

```
124 X(PI) = AF \ (AF' \ (B * (X(UDI) - X(UI))));
```

Finally, we depict the function `nl_apply`, which applies the reduced Hessian to a vector as described in Section 4. Note that the control equation is a system for

f and g, hence the apply function computes the action of the Hessian on a vector corresponding to both updates $\hat{\boldsymbol{f}}_k$ (denoted by V_1) and $\hat{\boldsymbol{g}}_k$ (denoted by V_2) (lines 7-8).

```
1 function GNV = nl_apply(V, B, AF, C1, C2, R1, R2)
2 [n,m] = size(R1);
3 V1 = V(1:n);
4 V2 = V(n+1:end);
5 du = AF \ (AF' \ (-C1 * V1 - C2 * V2));
6 dp = AF \ (AF' \ (-B * du));
7 GNV1 = C1' * dp + R1 * V1;
8 GNV2 = C2' * dp + R2 * V2;
9 GNV = [GNV1; GNV2];
```

The complete code listing for this problem can be found in Appendix B.4.

Appendix B. Complete code listings.

B.1. Steepest descent for elliptic inverse problem. The complete implementation of the steepest descent method to solve the elliptic parameter inversion problem (2.1) is given below. Parts of the implementation are discussed in Appendix A.1.

```
clear all; close all;
fem.geom = rect2(0,1,0,1);
fem.mesh = meshmap(fem, 'Edgelem', {1,20,2,20});
fem.dim = {'a' 'grad' 'p' 'u' 'ud'};
fem.shape = [1 1 2 2 2];
fem.equ.expr.atrue = '1 + 7*(sqrt((x-0.5)^2 + (y-0.5)^2) > 0.2)';
fem.equ.expr.f = '1';
fem.equ.expr.gamma = '1e-9';
datanoise = 0.01;
maxiter = 500;
tol = 1e-8;
c = 1e-4;
alpha = 1e5;
fem.bnd.r =  {{'u' 'p' 'ud'}};
fem.equ.expr.goal = '-(atrue*(udx*udx_test+udy*udy_test)-f*ud_test)';
fem.equ.expr.state = '-(a*(ux*ux_test+uy*uy_test)-f*u_test)';
fem.equ.expr.adjoint = '-(a*(px*px_test+py*py_test)-(ud-u)*p_test)';
fem.equ.expr.control = ['(grad*grad_test-gamma*(ax*gradx_test+ay'...
        '*grady_test)-(px*ux+py*uy)*grad_test)'];
fem.equ.weak = 'goal';
fem.xmesh = meshextend(fem);
fem.sol = femlin(fem, 'Solcomp', {'ud'});
nodes = xmeshinfo(fem ,'out', 'nodes');
dofs = nodes.dofs';
AI = dofs(dofs(:,1)>0,1);
GI = dofs(dofs(:,2)>0,2);
PI = dofs(dofs(:,3)>0,3);
UI = dofs(dofs(:,4)>0,4);
UDI = dofs(dofs(:,5)>0,5);
X = fem.sol.u;
randn('seed', 0);
X(UDI) = X(UDI) + datanoise * max(abs(X(UDI))) * randn(length(UDI),1);
X(AI) = 8.0;
fem.equ.weak = 'state + adjoint + control';
fem.xmesh = meshextend(fem);
fem.sol = femlin(fem, 'Solcomp', {'u'}, 'U', X);
X(UI) = fem.sol.u(UI);
cost_old = evaluate_cost (fem, X);
for iter = 1:maxiter
```

```
    fem.sol = femlin(fem, 'Solcomp', {'u'}, 'U', X);
    X(UI) = fem.sol.u(UI);
    fem.sol = femlin(fem, 'Solcomp', {'p'}, 'U', X);
    X(PI) = fem.sol.u(PI);
    fem.sol = femlin(fem, 'Solcomp', {'grad'}, 'U', X);
    X(GI) = fem.sol.u(GI);
    grad2 = postint (fem, 'grad * grad');
    Xtry = X;
    alpha = alpha * 1.2;
    descent = 0;
    no_backtrack = 0;
    while (~descent && no_backtrack < 10)
        Xtry(AI) = X(AI) - alpha * X(GI);
        fem.sol = femlin(fem, 'Solcomp', {'u'}, 'U', Xtry);
        Xtry(UI) = fem.sol.u(UI);
        [cost, misfit, reg] = evaluate_cost (fem, Xtry);
        if (cost < cost_old - alpha * c * grad2)
            cost_old = cost;
            descent = 1;
        else
            no_backtrack = no_backtrack + 1;
            alpha = 0.5 * alpha;
        end
    end
    X = Xtry;
    fem.sol = femsol(X);
    if (sqrt(grad2) < tol && iter > 1)
        fprintf (['Gradient method converged after %d iterations.'...
                  '\n\n'], iter);
        break;
    end
end
```

The cost function evaluation for given solution vector `X` is listed below:

```
function [cost, misfit, reg] = evaluate_cost(fem, X)
fem.sol = femsol(X);
misfit = postint (fem, '0.5 * (u - ud)^2');
reg = postint (fem, '0.5 * gamma * (ax^2+ay^2)');
cost = misfit + reg;
```

B.2. Gauss-Newton-CG for linear elliptic inverse problems. Below, we give the complete COMSOL Multiphysics implementation of the Gauss-Newton-CG method to solve the elliptic coefficient inverse problem described in Section 2.

```
clear all; close all;
fem.geom = rect2(0,1,0,1);
fem.mesh = meshmap(fem, 'Edgelem', {1,20,2,20});
fem.dim = {'a' 'a0' 'delta_a' 'delta_p' 'delta_u' 'g' 'p' 'u' 'ud'};
fem.shape = [1 1 1 2 2 1 2 2 2];
fem.equ.expr.atrue = '1 + 7*(sqrt((x-0.5)^2 + (y-0.5)^2) > 0.2)';
datanoise = 0.01;
fem.equ.expr.f = '1';
fem.equ.expr.gamma = '1e-9';
maxiter = 100;
tol     = 1e-8;
c        = 1e-4;
rho      = 0.5;
mu = 0;
nrCGiter = 0;
fem.bnd.r =  {{'delta_u' 'delta_p' 'u' 'ud'}};
fem.equ.expr.goal = '-(atrue*(udx*udx_test+udy*udy_test)-f*ud_test)';
```

```
fem.equ.expr.state = '-(a*(ux*ux_test+uy*uy_test)-f*u_test)';
fem.equ.expr.incstate = ['-(a*(delta_ux*delta_px_test+delta_uy*'...
        'delta_py_test)+delta_a*(ux*delta_px_test+uy*delta_py_test))'];
fem.equ.expr.incadjoint = ['-(a*(delta_px*delta_ux_test+delta_py*'...
        'delta_uy_test)+delta_u*delta_u_test)'];
fem.equ.expr.inccontrol = ['-(gamma*(delta_ax*delta_ax_test+delta_ay'...
        '*delta_ay_test)+(delta_px*ux+delta_py*uy)*delta_a_test)'];
fem.equ.weak = 'goal';
fem.xmesh = meshextend(fem);
fem.sol = femlin(fem, 'Solcomp', 'ud');
nodes = xmeshinfo(fem ,'out', 'nodes');
dofs = nodes.dofs';
AI  = dofs(dofs(:,1)>0,1);
A0I = dofs(dofs(:,2)>0,2);
dAI = dofs(dofs(:,3)>0,3);
dPI = dofs(dofs(:,4)>0,4);
dUI = dofs(dofs(:,5)>0,5);
GI  = dofs(dofs(:,6)>0,6);
PI  = dofs(dofs(:,7)>0,7);
UI  = dofs(dofs(:,8)>0,8);
UDI = dofs(dofs(:,9)>0,9);
X = fem.sol.u;
fem.equ.weak = '- g * g_test';
fem.xmesh = meshextend(fem);
K = assemble(fem, 'out', 'K');
M = K(GI,GI);
randn('seed', 0);
X(UDI) = X(UDI) + datanoise * max(abs(X(UDI))) * randn(length(UDI),1);
X(AI)  = 8.0;
X(A0I) = 1.0;
fem.equ.weak = 'state + incstate + incadjoint + inccontrol';
for iter = 1:maxiter
    fem.xmesh = meshextend(fem);
    fem.sol = femlin(fem, 'Solcomp', {'u'}, 'U', X);
    X(UI) = fem.sol.u(UI);
    if (iter == 1)
        [cost_old,misfit,reg,logb] = elliptic_cost_logb(fem,X,AI,A0I,mu);
    end
    fem.xmesh = meshextend(fem);
    [K, N] = assemble(fem, 'U', X, 'out', {'K', 'N'});
    W = K(dUI,dUI);
    A = K(dUI,dPI);
    C = K(dPI,dAI);
    R = K(dAI,dAI);
    ind = find(sum(N(:, dUI),1)~=0);
    A(:,ind) = 0;
    A(ind,:) = 0;
    for (k = 1:length(ind))
        i = ind(k);
        A(i,i) = 1;
    end
    chi = ones(size(A,1),1);
    chi(ind) = 0;
    AF = chol(A);
    X(PI) = AF \ (AF' \ (chi .* (W * (X(UDI) - X(UI)))));
    MG =  C' * X(PI) + R * X(AI);
    RHS = -MG + mu ./ (X(AI) - X(A0I));
    X(GI) = M \ (MG -  mu * (1./ (X(AI) - X(A0I))));
    gradnorm = sqrt (postint (fem, 'g^2', 'U', X));
    if iter == 1
        gradnorm_ini = gradnorm;
    end
```

```
80      tolcg = min(0.5, sqrt(gradnorm/gradnorm_ini));
81      P = R + 1e-10*eye(length(AI));
82      [D,flag,relres,CGiter,resvec] = pcg (@(V) elliptic_apply_logb(V,chi,W,...
83          AF,C,R,X,AI,A0I,mu),RHS,tolcg,300,P);
84      nrCGiter = nrCGiter + CGiter;
85      Xtry = X;
86      if mu <= 0
87          alpha = 1;
88      else
89          idx = find(D < 0);
90          Aviol = X(AI) - X(A0I);
91          if min(Aviol) <= 1e-9
92              error('point is not feasible');
93          end
94          if (isempty(idx))
95              alpha = 1;
96          else
97              alpha1 = min(-Aviol(idx)./D(idx));
98              alpha  = min(1, 0.9995*alpha1);
99          end
100     end
101     descent = 0;
102     no_backtrack = 0;
103     while (~descent && no_backtrack < 20)
104         Xtry(AI)  = X(AI) + alpha * D;
105         Xtry(dAI) = D;
106         fem.xmesh = meshextend(fem);
107         fem.sol = femlin(fem, 'Solcomp', 'u', 'U', Xtry);
108         Xtry(UI)  = fem.sol.u(UI);
109         [cost,misfit,reg,logb] = elliptic_cost_logb(fem,Xtry,AI,A0I,mu);
110         if (cost < cost_old + c * alpha * MG' * D)
111             cost_old = cost;
112             descent = 1;
113         else
114             no_backtrack = no_backtrack + 1;
115             alpha = rho * alpha;
116         end
117     end
118     if (descent)
119         X = Xtry;
120     else
121         error('Linesearch failed \n');
122     end
123     fem.sol = femsol(X);
124     a_update = sqrt (postint (fem, 'delta_a^2'));
125     dist = sqrt (postint(fem, '(atrue - a)^2'));
126     if ((a_update < tol && iter > 1) || gradnorm < tol)
127         fprintf (' *** GN converged after %d iterations. ***\n', iter);
128         break;
129     end
130 end
```

The function that implements the cost function evaluation for given solution vector `X` is:

```
1 function [cost, misfit, reg, logb] = elliptic_cost_logb(fem, X,
2     AI, A0I, mu)
3 fem.sol = femsol(X);
4 misfit = postint (fem, '0.5*(u-ud)^2');
5 reg = postint (fem, '0.5*gamma*(ax^2+ay^2)');
6 logb = mu * sum(log(X(AI)-X(A0I)));
7 cost = misfit + reg - logb;
```

Next, we list the function that applies the reduced Hessian to a vector `V`:

```
function GNV = elliptic_apply_logb(V, chi, W, AF, C, R, X, AI, A0I, mu)
du = AF \ (AF' \ (chi .* (-C * V)));
dp = AF \ (AF' \ (chi .* (-W * du)));
Z = mu * spdiags( 1 ./ (X(AI)-X(A0I)).^2, 0, length(AI), length(AI));
GNV = C' * dp + (R + Z)* V;
```

B.3. Conjugate gradient method for inverse advective-diffusive transport. In this section, we give the implementation for the conjugate gradient method (corresponding to a single step of the Gauss-Newton-CG method) for the initial condition inversion in advective-diffusive transport as described in Section 3.

```
clear all; close all;
global cgcount; cgcount = 0;
T = 4; Tm_start = 1; Tm = Tm_start / T;
ntime = 20;
gamma1 = 0; %1e-5;
gamma2 = 1e-6;
datanoise = 0.005;
s1 = square2('1','base','corner','pos',{'0','0'});
s2 = rect2('0.25','0.25', 'base','corner','pos',{'0.25','0.15'});
s3 = rect2('0.15','0.25','base','corner','pos',{'0.6','0.6'});
fem.geom = s1-s2-s3;
fem.bnd.ind = [1  2  3  4  4  4  4  4  4  4  4  5 ];
fem.pnt.ind = [1  2  2  2  2  2  2  2  2  2  2  2 ];
fem.mesh = meshinit(fem, 'hauto', 3);
fem.dim = {'p' 'q' 'u' 'u0' 'v1' 'v2'};
fem.shape = [2 1 2 2 2 2];
fem.const  = {'Re', 1e2, 'kappa', .001};
fem.expr.true_init = 'min(0.5,exp(-100*((x-.35)*(x-.35) + (y-.7)*(y-.7))))';
fem.equ.weak = ['-(2/Re * (v1x * v1x_test + 0.5*(v1y+v2x) * (v1y_test+v2x_test)' ...
        '+ v2y * v2y_test) + (v1 * v1x + v2 * v2x) * v1_test'...
        '+ (v1 * v1y + v2 * v2y) * v2_test'...
        '- q * (v1x_test + v2y_test) - q_test * (v1x + v2y) )'];
fem.bnd.r =  {{'v1' 'v2-1'} {'v1' 'v2'} {'v1' 'v2'} {'v1' 'v2'} {'v2+1' 'v1'}};
fem.pnt.constr = {{'q'} {}};
fem.xmesh = meshextend(fem);
fem.sol = femnlin(fem, 'Solcomp', {'q', 'v1', 'v2'});
figure('DefaultAxesFontSize',20); box on; hold on;
postplot(fem,'arrowdata',{'v1','v2'}, 'arrowxspacing', 15 , ...
    'arrowyspacing', 15,  'axis',[0,1,0,1]); axis equal tight;
nodes = xmeshinfo(fem ,'out', 'nodes');
dofs  = nodes.dofs';
PI    = dofs(dofs(:,1)>0,1);
UI    = dofs(dofs(:,3)>0,3);
U0I   = dofs(dofs(:,4)>0,4);
X = fem.sol.u;
fem.equ.weak = ['-(kappa * (ux * ux_test + uy * uy_test)'...
    '+ (v1 * ux + v2 * uy) * u_test)'];
fem.xmesh = meshextend(fem);
K = assemble(fem, 'U', X, 'out', 'K');
N = K(UI,UI);
fem.equ.weak = '-(ux * ux_test + uy * uy_test)';
fem.xmesh = meshextend(fem);
K = assemble(fem, 'U', X, 'out', 'K');
R = K(UI,UI);
fem.equ.weak = '- (u * u_test)';
fem.xmesh = meshextend(fem);
K = assemble(fem, 'out', 'K');
M = K(UI,UI);
fem.equ.weak = '0';
```

```
fem.bnd.weak =  {{} {} {} {'-u * u_test'} {}};
fem.xmesh = meshextend(fem);
MB = assemble(fem, 'out', 'K');
Q = MB(UI,UI);
dt = T / ntime;
L = M + dt * N;
fem.equ.weak = '-(u0 * u0_test - true_init * u0_test)';
fem.xmesh = meshextend(fem);
fem.sol = femlin(fem, 'Solcomp', 'u0');
X(U0I) = fem.sol.u(U0I);
UU = zeros(length(UI), ntime);
PP = zeros(length(PI), ntime);
UD = zeros(length(UI), ntime);
UU(:,1) = X(U0I);
for k = 1:(ntime-1)
    UU(:,k+1) = L \ (M * UU(:,k));
end
randn('seed', 0);
for it = 1:ntime
    UD(:, it) = UU(:, it) + datanoise * max(abs(UU(:, it))) *...
        randn(length(UU(:, it)),1);
end
F = Q * (dt * UD(:,ntime));
PP(:,ntime) = L' \ F;
for k = (ntime-1):-1:2
    m = (k > Tm * ntime);
    F = M * PP(:,k+1) + m * Q * (dt * UD(:,k));
    PP(:,k) = L' \ F;
end
PP(:,1) = M * PP(:,2);
Prec = gamma2 * R + max(1e-11, gamma1) * M;
[U0, flag, relres, CGiter, resvec] = pcg(@(U0)ad_apply(U0, L, M, R,...
    Q, dt, Tm, ntime, gamma1, gamma2, UU, PP), PP(:,1), 1e-4, 1000, Prec);
```

The function that applies the reduced Hessian to a vector is listed below:

```
function out = ad_apply(U0, L, M, R, Q, dt, Tm, ntime, gamma1, gamma2, ...
    UU, PP)
global cgcount;
UU(:,1) = U0;
for k = 1:(ntime-1)
    UU(:,k+1) = L \ (M * UU(:,k));
end
F = Q * (- dt * UU(:,ntime));
PP(:,ntime) = L' \ F;
for k = (ntime-1):-1:2
    m = (k > Tm * ntime);
    F = M * PP(:,k+1) + m * Q * (- dt * UU(:,k));
    PP(:,k) =  L' \ F;
end
MP(:,1) = M * PP(:,2);
out = - MP(:,1) + gamma1 * M * U0 + gamma2 * R * U0;
cgcount = cgcount + 1;
```

B.4. Gauss-Newton-CG for nonlinear elliptic inverse problems. In what follows, we list the complete code for the Gauss-Newton-CG method applied for the nonlinear elliptic inverse problem, in which we invert for volume and boundary source terms as described in Section 4.

```
clear all; close all;
g1=circ2('1','base','center','pos',{'0','0'},'rot','0');
theta = [0:2*pi/6:2*pi];
```

```
npts  = 5;
k = 1;  s='g1';
for i = 1:npts
    if i == npts
        theta = [0:2*pi/32:2*pi];
    end
    for j = 1:length(theta)-1
        x(j,i) =  (i/(npts+1))*cos(theta(j));
        y(j,i) =  (i/(npts+1))*sin(theta(j));
        eval(['p', num2str(k), '=point2(x(',num2str(j),',',...
                num2str(i),'), y(',num2str(j),',',num2str(i),'));']);
        s = [s, ', p', num2str(k)];
        k = k+1;
    end
end
eval(['fem.geom = geomcoerce(''solid'',{',s,'});']);
figure; geomplot(fem.geom,'edgelabels', 'on', 'pointlabels', 'on');
sind = []; spnt = []; smesh = []; totpnts = k+3;
for it = 1:totpnts
    if it == 1 || it == totpnts/2-1 || it == totpnts/2-1+3 || it == totpnts
        ind = 1;
    else
        ind = 2;
        spnt = [spnt, ' ', num2str(it)];
        smesh = [smesh, num2str(it), ', 0.1, '];
    end
    sind = [sind, ' ', num2str(ind)];
end
eval(['fem.mesh=meshinit(fem, ''hauto'', 7, ''hmaxvtx'', [',smesh,']);'])
eval(['fem.pnt.ind = [',sind,'];']);
eval(['pts = [',spnt,'];']);
eval(['fem.geom = geomcoerce(''solid'',{',s,'});']);
figure, meshplot(fem);
fem.dim = {'delta_f' 'delta_g' 'delta_p' 'delta_u' 'f' 'g' 'grad'
           'p' 'u' 'ud'};
fem.shape = [1 1 2 2 1 1 1 2 2 2];
fem.equ.expr.ftrue = '1 + 3*(y>=0)';
fem.bnd.expr.gtrue = 'x + y';
datanoise = 0.01;
fem.equ.expr.gamma1 = '1e-5';
fem.equ.expr.gamma2 = '1e-4';
maxiter = 100;
tol = 1e-7;
ntol = 1e-9;
fem.const  = {'c', 1000};
c1 = 1e-4;
rho = 0.5;
nrCGiter = 0;
fem.equ.expr.goal =['-(udx*udx_test+udy*udy_test+ud*ud_test+'...
        c*ud^3*ud_test-ftrue*ud_test)'];
fem.equ.expr.state = ['-(ux*ux_test+uy*uy_test+u*u_test+'...
        'c*u^3*u_test-f*u_test)'];
fem.equ.expr.adjoint = ['-(px*px_test+py*py_test+p*p_test+'...
        '3*c*u^2*p*p_test)'];
fem.equ.expr.incstate = ['-(delta_ux*delta_px_test+'...
        'delta_uy*delta_py_test+delta_u*delta_p_test+'...
        '3*c*u^2*delta_u*delta_p_test-delta_f*delta_p_test)'];
fem.equ.expr.incadjoint = ['-(delta_px*delta_ux_test+'...
        'delta_py*delta_uy_test+delta_p*delta_u_test+'...
        '3*c*u^2*delta_p*delta_u_test)'];
fem.equ.expr.inccontrolf = ['-(gamma1*(delta_fx*delta_fx_test+'...
        'delta_fy*delta_fy_test)-delta_f_test*delta_p '...
```

```
66          '+gamma1*(fx*delta_fx_test+fy*delta_fy_test)-delta_f_test*p)'];
67 fem.bnd.expr.inccontrolg = [''];
68 fem.equ.weak = 'goal';
69 fem.bnd.weak = {{'-gtrue * ud_test'}};
70 fem.xmesh = meshextend(fem);
71 fem.sol = femnlin(fem, 'Solcomp', 'ud', 'ntol', ntol);
72 nodes = xmeshinfo(fem ,'out', 'nodes');
73 dofs = nodes.dofs';
74 dFI = dofs(dofs(:,1)>0,1);
75 dGI = dofs(dofs(:,2)>0,2);
76 dPI = dofs(dofs(:,3)>0,3);
77 dUI = dofs(dofs(:,4)>0,4);
78 FI  = dofs(dofs(:,5)>0,5);
79 GI  = dofs(dofs(:,6)>0,6);
80 GRI = dofs(dofs(:,7)>0,7);
81 PI  = dofs(dofs(:,8)>0,8);
82 UI  = dofs(dofs(:,9)>0,9);
83 UDI  = dofs(dofs(:,10)>0,10);
84 X = fem.sol.u;
85 fem.equ.weak = '- grad * grad_test';
86 fem.xmesh = meshextend(fem);
87 K = assemble(fem, 'out', 'K');
88 M = K(GRI,GRI);
89 fem.equ.weak = '0';
90 fem.bnd.weak = '0';
91 fem.pnt.weak =  {{},{'-p * p_test'}};
92 fem.xmesh = meshextend(fem);
93 K = assemble(fem, 'out', 'K');
94 B = K(PI,PI);
95 fem.equ.weak = '0';
96 fem.bnd.weak =  {{'-g * g_test'}};
97 fem.xmesh = meshextend(fem);
98 K = assemble(fem, 'out', 'K');
99 QP = K(GI,GI);
100 randn('seed', 0);
101 X(UDI) = X(UDI) + datanoise * max(abs(X(UDI))) * randn(length(UDI),1);
102 X(FI) = 3.0;
103 X(GI) = 0.0;
104 fem.sol = femsol(X);
105 fem.equ.weak = 'state + incstate + incadjoint + inccontrolf';
106 fem.bnd.weak = {{'-gamma2*delta_g*delta_g_test'...
107     '-delta_p*delta_g_test-delta_g*delta_p_test-g*u_test'}};
108 for iter = 1:maxiter
109     fem.xmesh = meshextend(fem);
110     fem.sol = femnlin(fem, 'Solcomp', {'u'}, 'U', X, 'ntol', ntol);
111     X(UI) = fem.sol.u(UI);
112     if (iter == 1)
113        [cost_old, misfit, reg] = nl_cost(fem, X, pts);
114     end
115     fem.xmesh = meshextend(fem);
116     K = assemble(fem, 'out', 'K', 'U', X);
117     W = K(dUI,dUI);
118     A = K(dUI,dPI);
119     C1 = K(dPI,dFI);
120     R1 = K(dFI,dFI);
121     R2 = K(dGI,dGI);
122     C2 = K(dPI,dGI);
123     AF = chol(A);
124     X(PI) = AF \ (AF' \ (B * (X(UDI) - X(UI))));
125     MGF =  C1' * X(PI) + R1 * X(FI);
126     MGG =  C2' * X(PI) + R2 * X(GI);
127     MG  = [MGF; MGG];
```

```
128     X(GRI) = M \ MGF;
129     gradnormf = sqrt (postint (fem, 'grad^2', 'U', X));
130     X(GRI) = QP \ MGG;
131     gradnormg = sqrt (postint (fem, 'grad^2', 'U', X));
132     if iter == 1
133         gradnormf_ini = gradnormf;
134         gradnormg_ini = gradnormg;
135     end
136     tolcg = sqrt((gradnormf/gradnormf_ini)^2...
137           + (gradnormg/gradnormg_ini)^2);
138     tolcg = min(0.5, sqrt(tolcg));
139     P = [R1 zeros(length(GI)); zeros(length(FI)) R2]
140       + 1e-10*eye(length(GI) + length(FI));
141     [D, flag, relres, CGiter, resvec] = pcg (@(V) nl_apply(V,B,AF,C1,...
142         C2,R1,R2),-MG,tolcg,400,P);
143     nrCGiter = nrCGiter + CGiter;
144     Xtry = X;
145     alpha = 1.0;
146     descent = 0;
147     no_backtrack = 0;
148     while (~descent && no_backtrack < 20)
149         Xtry(FI)  = X(FI) + alpha * D(1:length(FI));
150         Xtry(dFI) = D(1:length(FI));
151         Xtry(GI)  = X(GI) + alpha * D(length(FI)+1:end);
152         Xtry(dGI) = D(length(FI)+1:end);
153         fem.xmesh = meshextend(fem);
154         fem.sol = femnlin(fem, 'Solcomp', 'u', 'U', Xtry, 'ntol', ntol);
155         Xtry(UI)  = fem.sol.u(UI);
156         [cost, misfit, reg] = nl_cost(fem, Xtry, pts);
157         if (cost < cost_old + c1 * alpha * MG' * D)
158             cost_old = cost;
159             descent = 1;
160         else
161             no_backtrack = no_backtrack + 1;
162             alpha = rho * alpha;
163         end
164     end
165     if (descent)
166         X = Xtry;
167     else
168         error('Linesearch failed \n');
169     end
170     fem.sol = femsol(X);
171     f_update = sqrt (postint (fem, 'delta_f^2'));
172     g_update = sqrt (postint (fem, 'delta_g^2'));
173     dist = sqrt (postint(fem, '(ftrue - f)^2'));
174     graddir = sqrt(-MG' * D);
175     if (f_update < tol && g_update < tol) && (iter > 1) ||...
176             (gradnormf < tol && gradnormg < tol)
177         fprintf (' *** GN converged after %d iterations. ***\n', iter);
178         break;
179     end
180 end
```

The function that evaluates the cost function for a given solution vector is:

```
1 function [cost, misfit, reg] = nl_cost(fem, X, pts)
2 misfit = 0;
3 for it = 1:length(pts)
4 misfit = misfit + postint (fem, '0.5*(u-ud)^2','dl', pts(it), ...
5     'edim',0, 'U', X);
6 end
7 reg1 = postint (fem, '0.5*gamma1*(fx*fx+fy*fy)');
```

```
reg2 = postint (fem, '0.5*gamma2*g*g', 'dl',[1,2,3,4], 'edim',1);
reg  = reg1 + reg2;
cost = misfit + reg;
```

Finally, the function that applies the reduced Hessian to a vector `V` is as follows:

```
function GNV = nl_apply(V, B, AF, C1, C2, R1, R2)
[n,m] = size(R1);
V1 = V(1:n);
V2 = V(n+1:end);
du = AF \ (AF' \ (-C1 * V1 - C2 * V2));
dp = AF \ (AF' \ (-B * du));
GNV1 = C1' * dp + R1 * V1;
GNV2 = C2' * dp + R2 * V2;
GNV = [GNV1; GNV2];
```

REFERENCES

[1] V. Akçelik, G. Biros, A. Draganescu, O. Ghattas, J. Hill, and B. van Bloeman Waanders, *Dynamic data-driven inversion for terascale simulations: Real-time identification of airborne contaminants*, in Proceedings of SC2005, Seattle, 2005.

[2] V. Akçelik, G. Biros, O. Ghattas, J. Hill, D. Keyes, and B. van Bloeman Waanders, *Parallel PDE-constrained optimization*, in Parallel Processing for Scientific Computing, M. Heroux, P. Raghaven, and H. Simon, eds., SIAM, 2006.

[3] W. Bangerth, *A framework for the adaptive finite element solution of large-scale inverse problems*, SIAM Journal on Scientific Computing, 30 (2008), pp. 2965–2989.

[4] W. Bangerth, R. Hartmann, and G. Kanschat, *deal.II – a general-purpose object-oriented finite element library*, ACM Transactions on Mathematical Software, 33 (2007), p. 24.

[5] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander, *A generic grid interface for parallel and adaptive scientific computing. Part I: Abstract framework*, Computing, 82 (2008), pp. 103–119.

[6] M. Bergounioux, M. Haddou, M. Hintermüller, and K. Kunisch, *A comparison of a Moreau-Yosida based active set strategy and interior point methods for constrained optimal control problems*, SIAM Journal on Optimization, 11 (2000), pp. 495–521.

[7] D. Braess, *Finite Elements. Theory, Fast Solvers, and Applications in Solid Mechanics*, Cambridge University Press, Cambridge, New York, 1997.

[8] A. N. Brooks and T. J. R. Hughes, *Streamline upwind/Petrov–Galerkin formulations for convection dominated flows with particular emphasis on the incompressible Navier–Stokes equations*, Computer Methods in Applied Mechanics and Engineering, 32 (1982), pp. 199–259.

[9] G. Chavent, *Nonlinear Least Squares for Inverse Problems*, Springer, 2009.

[10] COMSOL AB, *COMSOL Multiphysics Reference Guide*, 2009. (`http://www.comsol.com`).

[11] T. Dupont, J. Hoffman, C. Johnson, R. Kirby, M. Larson, A. Logg, and R. Scott, *The FEniCS project*, tech. rep., 2003.

[12] S. C. Eisenstat and H. F. Walker, *Choosing the forcing terms in an inexact Newton method*, SIAM Journal on Scientific Computing, 17 (1996), pp. 16–32.

[13] H. Engl, M. Hanke, and A. Neubauer, *Regularization of Inverse Problems*, Springer Netherlands, 1996.

[14] M. D. Gunzburger, *Perspectives in Flow Control and Optimization*, SIAM, Philadelphia, 2003.

[15] E. Haber and L. Hanson, *Model problems in PDE-constrained optimization*, Tech. Rep. TR-2007-009, Emory University, 2007.

[16] M. Heinkenschloss, *Numerical solution of implicitly constrained optimization problems*, Tech. Rep. TR08-05, Department of Computational and Applied Mathematics, Rice University, 2008.

[17] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, *An Overview of the Trilinos Project*, ACM Transaction on Mathematical Software, (2005).

[18] J. Kaipio and E. Somersalo, *Statistical and Computational Inverse Problems*, vol. 160 of Applied Mathematical Sciences, Springer-Verlag, New York, 2005.

[19] C. T. Kelley, *Iterative Methods for Optimization*, SIAM, Philadelphia, 1999.
[20] A. Logg, K.-A. Mardal, and G. N. Wells, eds., *Automated Scientific Computing*, Springer, 2011.
[21] I. Neitzel, U. Prüfert, and T. Slawig, *Strategies for time-dependent PDE control with inequality constraints using an integrated modeling and simulation environment*, Numerical Algorithms, 50 (2009), pp. 241–269.
[22] J. Nocedal and S. J. Wright, *Numerical Optimization*, Springer Verlag, Berlin, Heidelberg, New York, second ed., 2006.
[23] T. Slawig, *PDE-constrained control using COMSOL Multiphysics – Control of the Navier-Stokes equations*, Numerical Algorithms, 42 (2006), pp. 107–126.
[24] A. Tarantola, *Inverse Problem Theory and Methods for Model Parameter Estimation*, SIAM, Philadelphia, PA, 2005.
[25] F. Tröltzsch, *Optimal Control of Partial Differential Equations: Theory, Methods and Applications*, vol. 112 of Graduate Studies in Mathematics, American Mathematical Society, 2010.
[26] C. R. Vogel, *Computational Methods for Inverse Problems*, Frontiers in Applied Mathematics, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2002.
[27] M. Weiser, *Interior point methods in function space*, SIAM Journal on Control and Optimization, 44 (2005), pp. 1766–1786.