Tutorial

J. Krämer, A. Sommer, A. Helfrich-Schkarbanenko

Direct and Inverse Problem in Electrostatics

modified on November 16, 2017

Version 1.4

Contents

Applying HiFlow^3 for solving direct and inverse problem in electrostatics

1 Introduction

HiFlow^3 is a multi-purpose finite element software providing powerful tools for efficient and accurate solution of a wide range of problems modeled by partial differential equations (PDEs). Based on object-oriented concepts and the full capabilities of C++ the HiFlow^3 project follows a modular and generic approach for building efficient parallel numerical solvers. It provides highly capable modules dealing with the mesh setup, finite element spaces, degrees of freedom, linear algebra routines, numerical solvers, and output data for visualization. Parallelism - as the basis for high performance simulations on modern computing systems - is introduced on two levels: coarse-grained parallelism by means of distributed grids and distributed data structures, and fine-grained parallelism by means of platform-optimized linear algebra back-ends.

1.1 How to Use the Tutorial?

You find the example code (direct_inverse_tutorial.cc, direct_inverse_tutorial.h) and a parameter file for the first numerical example (direct_inverse_tutorial.xml) in the folder `/hiflow/examples/direct_inverse`. The geometry data (*.inp, *.vtu) is stored in the folder `/hiflow/examples/data`.

1.1.1 Using HiFlow^3 as a Developer

First build and compile HiFlow^3. Go to the directory `/build/example/direct\_inverse`, where the binary **direct_inverse_tutorial** is stored. Type **./direct_inverse_tutorial**, to execute the program in sequential mode. To execute in parallel mode with four processes, type **mpirun -np 4 ./direct_inverse_tutorial**. In both cases, you need to make sure that the default parameterfile direct_inverse_tutorial.xml is stored in the same directory as the binary, and that the geometry data specified in the parameter file is stored in `/hiflow/examples/data`. Alternatively, you can specify the path of your own xml-file with the name of your xml-file (first) and the path of your geometry data (second) in the comment line, i.e. **./direct_inverse_tutorial** `/"path_to_parameterfile"/"name_of_parameterfile".xml /"path_to_geometry_data"/`.

2 Mathematical Setup

In this tutorial we study an elliptic boundary value problem with Robin boundary condition arising in electrostatics in a bounded Lipschitz domain $\Omega \subset \mathbb{R}^3$. We aim to reconstruct the current source $f : \overline{\Omega} \to \mathbb{R}$ with $supp(f) \subset \Omega$ on the basis of some local electric potential data u_d, assuming the electrical conductivity $\sigma : \overline{\Omega} \to \mathbb{R}_{\geq 0}$ is known. By local data we mean the restriction $u_d := u|_\Gamma$ of $u : \overline{\Omega} \to \mathbb{R}$ to a subdomain (path) $\Gamma \subset \Omega$ with $supp(f) \cap \Gamma = \emptyset$. To set up this so called "inverse problem" we firstly have to focus on the direct problem.

2.1 Direct Problem

Our aim is to find $u \in C^2(\Omega) \cap C^1(\overline{\Omega})$ that satisfies

$$\begin{aligned} -\nabla \cdot (\sigma \nabla u) &= f, \quad \text{in } \Omega, \\ \tfrac{\partial u}{\partial \nu} + gu &= 0, \quad \text{on } \partial\Omega, \end{aligned} \tag{1}$$

where $f \in C(\Omega)$ and $\sigma \in C^1(\Omega) \cap C(\overline{\Omega})$ are given. The function $g \in L^\infty_{>0}(\partial\Omega)$ describes the reciprocal distance between the boundary and the center of $supp(f)$. The Robin boundary condition on the artificial boundary $\partial\Omega$ forces the electric potential u to decrease reciprocally proportional to the distance between $\partial\Omega$ and the source. Doing so we imitate Ω to be unbounded, i.e. we minimize the influence of the artificial introduced boundary $\partial\Omega$ on the solution u, cf. Figure 1.

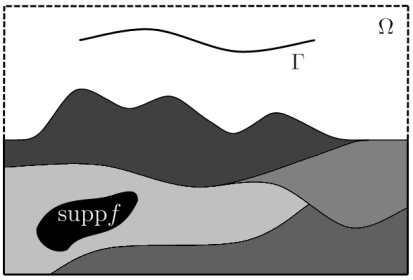

Figure 1: Scenario for the problem to investigate.

Referring to the Poisson Tutorial, finding classical solutions of (1) is difficult. To produce relief we have to reformulate this problem to the corresponding weak formulation, which requires weaker assumptions on the regularity of u, f and σ.

2.1.1 Weak Formulation

We consider the Hilbert space $L^2(\Omega)$, where the evolution equation and the boundary condition in (1) can be combined to one integral equation. Let u, f and σ be as in the strong formulation (1). To set up the weak formulation we multiply both sides of conductivity equation in (1) with a test function $\varphi \in C^\infty(\Omega)$ and integrate over Ω:

$$\int_\Omega -\nabla \cdot (\sigma \nabla u)\, \varphi \, \mathrm{d}x = \int_\Omega f\, \varphi \, \mathrm{d}x.$$

Since Ω has Lipschitz boundary $\partial\Omega$, Green's first identity and the Gaussian integral theorem can be applied

$$\int_\Omega \sigma \nabla u \cdot \nabla \varphi \, \mathrm{d}x - \int_{\partial\Omega} \sigma (\nabla u \cdot n) \varphi \, \mathrm{d}o = \int_\Omega f \varphi \, \mathrm{d}x, \tag{2}$$

where n is the (outer) normal vector on $\partial\Omega$. Since our boundary value problem contains natural boundary conditions, the Robin boundary condition $(\nabla u \cdot n) = -gu$ on $\partial\Omega$ can be integrated in (2):

$$\int_\Omega \sigma \nabla u \cdot \nabla \varphi \, \mathrm{d}x + \int_{\partial\Omega} \sigma\, g\, u\, \varphi \, \mathrm{d}o = \int_\Omega f \varphi \, \mathrm{d}x \qquad \forall \varphi \in C^\infty(\Omega). \tag{3}$$

Introducing the Sobolev space $H^2(\Omega) := \{u \in L^2(\Omega) : D^\alpha u \in L^2(\Omega) \quad \forall\, |\alpha| \leq 2\}$ for the solution u, see Braess [1], completes the derivation of the weak formulation. Because $C^\infty(\Omega)$ is dense in $H^2(\Omega)$ it is convenient to choose $\varphi \in H^2(\Omega)$, which yields

$$\int_\Omega \sigma \nabla u \cdot \nabla \varphi \,\mathrm{d}x + \int_{\partial\Omega} \sigma\, g\, u\, \varphi \,\mathrm{d}o = \int_\Omega f \varphi \,\mathrm{d}x \qquad \forall \varphi \in H^2(\Omega). \tag{4}$$

Consequently, we need the following weaker assumptions: $f \in L^2(\Omega)$, with $supp(f) \subset \Omega$ and $\sigma \in L^2(\Omega)$ with $0 < 1/C < \sigma(x) < C$ for all $x \in \Omega$ and $C > 0$. Now our aim is to find the so called weak solution $u \in H^2(\Omega)$ that solves equation (4).
Actually, the Sobolev space $H^1(\Omega)$ is sufficient for the weak formulation, see Sommer [7]. But with respect to the inverse problem in Section 2.2.1 we will see that the regularity $u \in H^2(\Omega)$ is necessary, because of the measurement methodology. This does not cause any difficulties, because applying the regularity theorem [6, Thm. 8.13, 8.14] yields $u \in H^2(\Omega)$ if $f \in L^2(\Omega)$ and $\partial\Omega$ is sufficient smooth, i.e. C^2 bound.
The weak formulation can be rewritten in brief formulation as follows: Find $u \in H^2(\Omega)$ that fulfills the integral equation

$$a(u, \varphi) \;=\; l(\varphi) \quad \forall \varphi \in H^2(\Omega), \tag{5}$$

where the bilinear form $a : H^2(\Omega) \times H^2(\Omega) \to \mathbb{R}$ defined by

$$a(u, \varphi) = \int_\Omega \sigma \nabla u \cdot \nabla \varphi \,\mathrm{d}x + \int_{\partial\Omega} \sigma\, g\, u\, \varphi \,\mathrm{d}o$$

is continuous and elliptic. The linear form $l : L^2(\Omega) \to \mathbb{R}$

$$l(\varphi) = \int_\Omega f\, \varphi \,\mathrm{d}x$$

is continuous, see Braess [1, Def. 2.4]. Because of these properties the direct problem satisfies the Lax-Milgram Lemma [2] and is therefore uniquely solvable.
Due to Céas lemma [1] we can discretize the problem in the sense of Galerkin without disturbing the unique solvability. That means, we can solve the weak formulation (5) uniquely in a finite dimensional subspace

$$H_h^2(\Omega) \subset H^2(\Omega)$$

for an example via FEM. A detailed proof is given by Sommer [7]. So we apply HiFlow^3 to find $u_h \in H_h^2(\Omega)$, which approximates the weak solution $u \in H^2(\Omega)$.

In respect of the inverse problem this weak formulation of boundary value problem (1) can be transferred to a brief operator equation

$$\Lambda[f] \;=\; u, \tag{6}$$

where the bounded, linear forward operator denoted by

$$\Lambda : \left\{ \begin{array}{ccc} L^2(\Omega) & \to & H^2(\Omega), \\ f & \mapsto & u, \end{array} \right.$$

maps given current density f onto electric potential u. Especially, Λ represents the governing physics.

2.1.2 Numerical Implementation

In the following we emphasize that the numerical linear operators are matrices and the discrete functions were represented by vectors. Recalling by Céa Lemma [1] we can solve the direct problem in a finite dimensional subspace $H_h^2(\Omega) \subset H^2(\Omega)$ without loosing the uniqueness of the solution. Thus, the weak formulation (5) can be presented in matrix notation

$$\boldsymbol{A}\boldsymbol{u} = \boldsymbol{l} \quad \text{for all } \varphi_k \in H_h^2(\Omega), \tag{7}$$

where $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ is the stiffness matrix, cf. HiFlow^3 Source Code in Listings 1, 2, 5 and 6, with elements

$$\boldsymbol{A}_{i,j} = \int_\Omega \sigma \nabla\varphi_i \cdot \nabla\varphi_j \, dx + \int_{\partial\Omega} \sigma \, g \, \varphi_i \, \varphi_j \, ds,$$

$\boldsymbol{l} \in \mathbb{R}^n$ the right hand side, cf. Listing 3, with entries

$$\boldsymbol{l}_i = \int_\Omega f \, \varphi_i \, dx,$$

$\boldsymbol{u} \in \mathbb{R}^n$ is the nodal electric potential vector and $f : L^2(\Omega) \to \mathbb{R}$ is the current source density function. The basis functions $\{\varphi_i\}_{i=1}^n$ are in the finite dimensional subspace $H_h^2(\Omega)$ of the Sobolev space $H^2(\Omega)$. The number of degrees of freedom of the model is denoted by n. Solving the direct problem means solving equation (7). But to set up the discrete forward operator $\boldsymbol{\Lambda}$ it is useful to represent the source function f by its nodal source vector $\boldsymbol{f} \in \mathbb{R}^n$, cf. Listing 4, so that (7) can be extended to

$$\boldsymbol{A}\boldsymbol{u} = \boldsymbol{M}\boldsymbol{f} \quad \text{for all } \varphi_k \in H_h^2(\Omega), \tag{8}$$

where $\boldsymbol{M} \in \mathbb{R}^{n\times n}$ is the mass matrix with entries

$$\boldsymbol{M}_{i,j} = \int_\Omega \varphi_i \, \varphi_j \, dx. \tag{9}$$

Based on equation (8) we find the discrete forward operator

$$\boldsymbol{\Lambda} = \boldsymbol{A}^{-1}\boldsymbol{M},$$

cf. Listing 9. Since the stiffness matrix $\boldsymbol{A}$ is symmetric and positive definite, see Sommer [7], it is invertible. It can be shown that the mass matrix $\boldsymbol{M}$ is symmetric, positive definite, too. Hence, the product $\boldsymbol{\Lambda}$ is invertible, but in general not positive definite and non-symmetric.

In the actual HiFlow^3 Source Code the inverse operators are being avoided, since the full invertation of a $n \times n$-matrix is very expensive for large n. Therefore $\boldsymbol{\Lambda}$ are computed by a series of linear systems of equations

$$\boldsymbol{A}\boldsymbol{\Lambda}_i = \boldsymbol{M}_i,$$

for all $i = 1, ..., n$, cf. Listing 9, where $\boldsymbol{\Lambda}_i$ and $\boldsymbol{M}_i$ are the i^{th} column of $\boldsymbol{\Lambda}$ and $\boldsymbol{M}$, respectively.

2.2 Inverse Problem

In this section we present an approach how to reconstruct f from some local data u_d.

In practice it is impossible to measure u in the whole domain Ω. Thus, we have to introduce a subdomain $\Gamma \subset \Omega$ with $supp(f) \cap \Gamma = \emptyset$, on which measurements are available. To restrict u to the local synthetic data $u_d := u|_\Gamma$ we define an operator

$$T : \begin{cases} H^2(\Omega) & \to & L^2(\Gamma), \\ u & \mapsto & u_d := u|_\Gamma, \end{cases} \tag{10}$$

which is necessary to generate local synthetic data

$$u_d = Tu = T\Lambda[f].$$

Note that the restriction to $u_d \in L^2(\Omega)$ is only possible if $u \in H^2(\Omega)$ is assumed, as shown by Sommer [7]. This is the reason why in the weak formulation the regularity $u \in H^2(\Omega)$ was chosen. Combination of Λ with T leads to the following ill-posed inverse problem:

Problem. *For given local measurements $u_d \in L^2(\Gamma)$ find $f \in L^2(\Omega)$ such that*

$$T\Lambda[f] = u_d \tag{11}$$

holds.

The essential point here is the usage of the restriction operator T. It restricts an electric potential u to local synthetic data u_d. The consequence is that $T\Lambda$ is non-injective. For example one small dipole source below ground level generates the same data u_d in the air as one big source in greater depth, see Sommer [7, Chap. 4.2.1]. Because of that non-injectivty of $T\Lambda$ the inverse problem (11) is ill-posed in sense of Hadamard [3]. In the discrete case the Matrix $\boldsymbol{T\Lambda}$ is strongly under-determined and thus singular. Hence, it can not be inverted and a regularizing technique should be applied.

2.2.1 Regularized Inverse Problem

We know that a solution of the inverse problem (11) exists, but it is non-unique. Hence the inverse problem (11) is ill-posed. However, applying the idea of Tikhonov [8], which is explained in next paragraph, we can enforce this uniqueness.

The concept of Tikhonov regularization is the following: If no true solution can be computed, we modify the problem in such way that the corresponding new solution, which is called "pseudo-solution", is unique and in relation to the true solution. In detail, the Tikhonov regularization consists of perturbation of an operator via spectral shift to enforce the uniqueness of a pseudo-solution. For further information we refer to Kirsch [4], Rieder [5]. Doing so we obtain the regularized inverse problem for local measurements:

Problem. *Let $u_d \in L^2(\Gamma)$ be local synthetic data on $\Gamma \subset \Omega$. Find $f \in L^2(\Omega)$ by solving the minimization problem*

$$f_{L,\alpha} = \underset{f \in L^2(\Omega)}{\operatorname{argmin}} \left\{ \tfrac{1}{2} \|T\Lambda[f] - u_d\|^2_{L^2(\Gamma)} + \tfrac{\alpha}{2} \|f\|^2_{L^2(\Omega)} \right\}, \tag{12}$$

for $\alpha > 0$.

It is well known that for $\alpha > 0$ the minimization problem (12) has a unique pseudo-solution f_α, which can be computed by

$$f_\alpha := (\Lambda^* T^* T \Lambda + \alpha I)^{-1} \Lambda^* T^* [u_d]. \tag{13}$$

where Λ^* and T^* are the adjoint operators of Λ and T. The operator I is the identity operator. As a consequence of the regularization the operator $\Lambda^* T^* T \Lambda + \alpha\, I$ is continuously invertible for $\alpha > 0$, see Rieder [5, Chap. 4]. Obviously, the pseudo-solution f_α depends on a regularizing parameter α, which shifts the nonnegative eigenvalues of $\Lambda^* T^* T \Lambda$ away from zero. Later we will see that the identity operator I can be generalized to involve apriori information about the true solution f.

Finding an optimal pseudo-solution means finding the corresponding optimal α, too. If α is too small, the operator $\Lambda^* T^* T \Lambda + \alpha I$ is ill-conditioned, so that the inverse does not exist. If α is too large, the influence of αI is too strong, so that the pseudo-solution has no relation to the true solution. The optimal α lies somewhere in the middle. In Section 2.2.2 we will elaborate on this choise of α. In the numerical example 4.2 we reconstruct the source $\boldsymbol{f}$ for three different α. The influence of the regularization parameter α to the pseudo-solution and its importance is presented in Figure 4 – 6. For further details see Rieder [5].

For now we know that the regularized inverse problem (12) has an unique pseudo-solution, which depends on the regularization parameter α. For the numerical computation of f_α we will use the discrete version of equation (13). Now we can turn to numerically implementation.

2.2.2 Numerical Implementation of the Regularized Inverse Problem

As told before the main difference between the direct and the inverse problem is the usage of the restriction operator T. Let the discrete restriction operator $\boldsymbol{T} \in \mathbb{R}^{k \times n}$, cf. Listing 8, where n is the quantity of degrees of freedom and $k \ll n$ is the number of measuring grid points, have the entities

$$\boldsymbol{T}_{i,j} := \begin{cases} 1, & \text{if the measuring grid point } i \text{ has} \\ & \text{the global grid point number } j, \\ 0, & \text{else.} \end{cases}$$

Now we are able to generate synthetic data by solving

$$\boldsymbol{u}|_{\Gamma_h} = \boldsymbol{T}\boldsymbol{\Lambda}\, \boldsymbol{f}, \tag{14}$$

where $\Gamma_h := \{p \mid p_i \text{ is measuring grid point},\ i = 1, \ldots, k\}$ is a set of measuring grid points and represents the discrete FE model of the subdomain Γ. Then the pseudo-solution $\boldsymbol{f}_\alpha$ can be computed by

$$\boldsymbol{f}_\alpha = \left(\boldsymbol{\Lambda}^\top \boldsymbol{T}^\top \boldsymbol{T} \boldsymbol{\Lambda} + \alpha \boldsymbol{I}\right)^{-1} \boldsymbol{\Lambda}^\top \boldsymbol{T}^\top \boldsymbol{u}_d, \quad \alpha > 0, \tag{15}$$

for synthetic data $\boldsymbol{u}_d$ generated by (14), cf. (13). In numerical sense the regularization parameter decides how far the nonnegative eigenvalues of the strong under-determined operator $\boldsymbol{\Lambda}^\top \boldsymbol{T}^\top \boldsymbol{T} \boldsymbol{\Lambda}$ will be shifted away from zero. Hence, α regulates the compromise between approximation and numerical stability, see Section 2.2.1.

One question remains: "Could α be computed apriori?" In numerical implementation the value of α depends on lots of parameters, like the geometry of Ω, the quantity of degrees of freedom, the condition number of $\boldsymbol{\Lambda}^\top \boldsymbol{T}^\top \boldsymbol{T} \boldsymbol{\Lambda}$, etc. In general some aposteriori parameter choice rules like

the discrepancy principle or the L-curve exist to restrict the co-domain of α. But finding apriori an optimal α is expensive. Heuristic investigations results that $\alpha := 0.05 \cdot \max_{i=1,\dots,n} |\boldsymbol{\Lambda}_{i,i}|$ is a good starting point for the first regularization, cf. Section 4.2.

Moreover, the eigenvalue shift in (15) treats every eigenvalue of the singular Matrix $\boldsymbol{\Lambda}^\top \boldsymbol{T}^\top \boldsymbol{T} \boldsymbol{\Lambda}$ equally. In our scenario this basic idea of Tikhonov yields a pseudo-solution where the source overlaps a part of the subdomain Γ. But this contradicts our requirement $supp(f) \cap \Gamma = \emptyset$, wherefore this simple regularization is useless. However, the identity matrix $\boldsymbol{I}$ in (15) can be replaced by a symmetric, positive definite matrix $\boldsymbol{D}$, without loosing the unique solvability of the regularized inverse problem, as shown by Rieder [5] and Sommer [7]. This special penalty operator $\boldsymbol{D}$ includes apriori informations about the source f like our requirement $supp(f) \cap \Gamma = \emptyset$.

Applying FEM we know the allocation of the eigenvalues of $\boldsymbol{D}$ to the grid points. So we can involve apriori information by penalizing every eigenvalue, respectively every grid point different. Let $\boldsymbol{f}_i$ represents the nodal source vector at the i-th grid point p_i. After identifying every grid point where $\boldsymbol{f}_i \overset{!}{\approx} 0$ has to be fulfilled we can expend our computation by a special penalty operator $\boldsymbol{D}$. We chose the entities of the diagonal matrix $\boldsymbol{D}$ as follows:

$$\boldsymbol{D}_{i,i} := \begin{cases} 10^\beta, & \text{if } \boldsymbol{f}_i \overset{!}{\approx} 0 \\ 1, & \text{else,} \end{cases} \tag{16}$$

cf. Listing 12, with $\beta > 0$. It forces the pseudo-solution to vanish at the i-th grid point. Note that all eigenvalues of $\boldsymbol{D}$ are greater than one.

regularization	**impact on pseudo-solution**	**penalty operator**
zero-order	minimal amplitude	$\boldsymbol{I}$
zero-order with damping	include apriori information	$\boldsymbol{D}$

Table 1: Possible penalty terms, where $\boldsymbol{I} \in \mathbb{R}^{n \times n}$ is the identity matrix and $\boldsymbol{D}$ from (16). Both matrices are symmetric, positive definite.

To sum up, the pseudo-solution $\boldsymbol{f}_\alpha$ is computed by the linear system of equations

$$\left(\boldsymbol{\Lambda}^\top \boldsymbol{T}^\top \boldsymbol{T} \boldsymbol{\Lambda} + \alpha \boldsymbol{D}\right) \boldsymbol{f}_\alpha = \boldsymbol{\Lambda}^\top \boldsymbol{T}^\top \boldsymbol{u}_d.$$

3 The Commented Source Code

3.1 Preliminaries

The direct and inverse problem tutorial needs the following two input files:

- A parameter file: The parameter file is an xml-file, which contains all parameters needed to execute the program. It is read in by the program. It is not necessary to recompile the program, when parameters in the xml-file are changed. By default the direct and inverse problem tutorial reads in the parameter file direct_inverse_tutorial.xml, see Section 1.1.1, which contains the parameters of the numerical example, see Section 4. This file is stored in `/hiflow/examples/direct_inverse/`.

- Geometry data: The file containing the geometry is specified in the parameter file (direct_inverse_tutorial.xml).

HiFlow^3 does not generate meshes for the domain Ω. Meshes in *.inp and *.vtu format can be read in. It is possible to extend the reader for other formats. Furthermore it is possible to generate other geometries by using external programs (Mesh generators) or by hand.
Furthermore it is possible to generate other geometries by using external programs (Mesh generators) or by hand.
Since the poisson equation is used to model the electrical potential, this tutorial is closely related to the HiFlow^3 Poisson tutorial. Therefore most of the source code will be omitted here and only the local assembly class and the solve function will be showed in detail.

3.2 LocalDirectInverseTutorialAssembler class

This class implements the system matrix and right-hand side locally for each cell. Additionally, this computes σ, g and f.

```
class LocalDirectInverseTutorialAssembler : private AssemblyAssistant<DIMENSION,
                                                                       double>
{
  public:
```

Listing 1: Computes local stiffness matrix A. (7)

```
    void operator() ( const Element<double>& element,
                      const Quadrature<double>& quadrature,
                      LocalMatrix& lm )
    {
        AssemblyAssistant<DIMENSION, double>::initialize_for_element ( element,
                                                                       quadrature );

        const double sigma = eval_sigma ( element );

        // Computes the local matrix.
        const int num_q = num_quadrature_points ( );
        for ( int q = 0; q < num_q; ++q )
        {
            const double wq = w ( q );
            const int n_dofs = num_dofs ( 0 );
            for ( int i = 0; i < n_dofs; ++i )
            {
                for ( int j = 0; j < n_dofs; ++j )
                {
                    lm ( dof_index ( i, 0 ), dof_index ( j, 0 ) ) +=
                          sigma * wq * dot ( grad_phi ( j, q ), grad_phi ( i, q ) ) *
                          std::abs ( detJ ( q ) );
                }
            }
        }
    }
```

Listing 2: Computes robin boundary condition.

```
    void operator() ( const Element<double>& element,
                      int facet_num,
                      const Quadrature<double>& quad,
                      LocalMatrix& lm )
```

```
{
    AssemblyAssistant<DIMENSION, double>::initialize_for_facet ( element,
                                                                 quad,
                                                                 facet_num );

    // Computes contribution to local matrix from boundary facet integral.
    // This terms has its origin in the Robin boundary condition.
    const int num_q = num_quadrature_points ( );

    const int n_dofs = num_dofs ( 0 );
    const double sigma = eval_sigma ( element );

    for ( int q = 0; q < num_q; ++q )
    {
        const double wq = w ( q );
        const double g = eval_g ( x ( q ) );

        for ( int i = 0; i < n_dofs; ++i )
        {
            for ( int j = 0; j < n_dofs; ++j )
            {
                lm ( dof_index ( i, 0 ), dof_index ( j, 0 ) ) +=
                        wq * sigma * g * phi ( i, q ) * phi ( j, q ) *
                        std::abs ( ds ( q ) );
            }
        }
    }
}
```

Listing 3: Computes right hand side, see (8).

```
void operator() ( const Element<double>& element,
                  const Quadrature<double>& quadrature,
                  LocalVector& lv )
{
    AssemblyAssistant<DIMENSION, double>::initialize_for_element( element,
                                                                  quadrature );

    const int num_q = num_quadrature_points ( );
    for ( int q = 0; q < num_q; ++q )
    {
        const double wq = w ( q );
        const int n_dofs = num_dofs ( 0 );
        for ( int i = 0; i < n_dofs; ++i )
        {
            lv[dof_index ( i, 0 )] += wq * f ( x ( q ) ) * phi ( i, q ) *
                        std::abs ( detJ ( q ) );
        }
    }
}
```

Listing 4: Defines f.

```
// Computes f. This is 1 in the left half of the oilfield,
// -1 in the other half  and 0 elsewhere.
double f ( Vec<DIMENSION, double> pt )
{
    double rhs_sol;

    rhs_sol = 0;
```

```
        if ( pt[1] <= 6 )
        {
            if ( pt[1] >= 4 )
            {
                if ( pt[2] <= -2 )
                {
                    if ( pt[2] >= -4 )
                    {
                        if ( pt[0] >= 6 )
                        {
                            if ( pt[0] <= 10 )
                            {
                                rhs_sol = 1;
                            }
                            else if ( pt[0] <= 14 )
                            {
                                rhs_sol = -1;
                            }
                        }
                    }
                }
            }
        }

        return rhs_sol;
    }
```

Listing 5: Defines σ.

```
    // Computes Sigma, which is constant in the air and in the ground,
    // but has a different value in the air than in the ground.
    double eval_sigma ( const Element<double>& element )
    {
        const int mat = element.get_cell ( ).get_material_number ( );
        if ( mat == 2 )
        { // The air has the material number 2.
            return 1;
        }
        else if ( mat == 1 )
        { // The ground has the material number 1.
            return 2;
        }
        throw "Unknown material number!";
        return 0.;
    }
```

Listing 6: Defines g.

```
    // Computes g. It punishes the distance to the center.
    double eval_g ( Vec<DIMENSION, double> pt )
    {
        std::vector<double> center ( 3, 0.0 );
        center[0] = 10.0;
        center[1] = 5.0;
        center[2] = -3.0;
        const Vec<3, double> center_vec ( center );
        return 1. / ( norm ( pt - center_vec ) );
    }
};
```

3.2.1 solve_system()

In the member function solve_system() the direct problem is solved to obtain the synthetic measurement data for the inverse problem.

```
void DirectInverseTutorial::solve_system ( )
{
    LinearSolver<LAD>* solver_;
    LinearSolverFactory<LAD> SolFact;
    solver_ = SolFact.Get (
            params_["LinearSolver"]["Name"].get<std::string>( ) )->
            params ( params_["LinearSolver"] );

    solver_->SetupOperator ( *matrix_ );
    solver_->Solve ( *rhs_, sol_ );

    ////////////////////////////// INVERSE PROBLEM ////////////////////////////////////
    // Here the solution which was computed above is reduced to a small set of
    // solution points. This refers to some discrete measure point in the reality.
    // After that f is inversely computed by
    //
    // f_alpha = (lambda' * T' * T * lambda + alpha * penalty) lambda' * T' * u_d
    //
    // with lambda being A^(-1)M, with A and M from the direct problem, u_d being
    // the reduced solution, T being a map-matrix from u to u_d and penalty being
    // a diagonal matrix with entries, where the dofs are on the boundary or in the
    // air.

    // Dimension of the (square) mass matrix m and the (square) stiffness matrix a.
    const int n = matrix_->nrows_global ( );

    // Number of cells for the iteration.
    int number_of_cell = master_mesh_->num_entities ( DIMENSION );

    std::vector<int> dof_count_trajectory ( n, -1 );
    std::vector<int> penalty_dof_count ( n, -1 );
    std::vector<std::vector<double> > coord_vec;
```

Listing 7: Computes u.

```
    // Computes the values of u_trajectory, which is the array of the sample data.
    // Iteration over all cells.
    for ( int k = 0; k < number_of_cell; k++ )
    {
        space_.dof ( ).get_coord_on_cell ( 0, k, coord_vec );

        int number_dof_on_cells = space_.dof ( ).get_nb_dofs_on_cell ( k );

        for ( int p = 0; p < number_dof_on_cells; p++ )
        {
            // Checks if the dof lies on the trajectory.
            if ( coord_vec[p][2] == 0 || coord_vec[p][2] == 1 )
            {
                // Computes the global dof number of the local dof on the cell.
                int place = space_.dof ( ).mapl2g ( 0, k, p );
                dof_count_trajectory[place] = place;
            }

            // If the dof lies on the facet or if it is in the air, a penalty is
            // later added, to improve the solution.
```

```
        if ( ( coord_vec[p][2] >= 0 ) || ( coord_vec[p][0] == 0 ) ||
             ( coord_vec[p][0] == 20 ) || ( coord_vec[p][1] == 0 ) ||
             ( coord_vec[p][1] == 10 ) || ( coord_vec[p][2] == -6 ) )
        {
            int place2 = space_.dof ( ).mapl2g ( 0, k, p );
            penalty_dof_count[place2] = place2;
        }
    }
}
```

Listing 8: Computes the synthetic measurement data u_d.

```
// Fills the vector u_trajectory with the values at the dofs on the trajectory.
// The vector has full size (n) and is zero if the dof is not on the trajectory
u_trajectory.resize ( n );
for ( int i = 0; i < n; i++ )
{
    u_trajectory[i] = 0;
    if ( dof_count_trajectory[i] != -1 )
    {
        sol_->GetValues( &dof_count_trajectory.at ( i ), 1, &u_trajectory.at(i));
    }
}

// Initialises the matrices needed in the following.
SeqDenseMatrix<Scalar> lambda;
lambda.Resize ( n, n );
lambda.Zeros ( );

SeqDenseMatrix<Scalar> penalty_mat;
penalty_mat.Resize ( n, n );
penalty_mat.Zeros ( );

SeqDenseMatrix<Scalar> identity;
identity.Resize ( n, n );
lambda.Zeros ( );

SeqDenseMatrix<Scalar> lambda_trans;
lambda_trans.Resize ( n, n );
lambda_trans.Zeros ( );

SeqDenseMatrix<Scalar> product;
product.Resize ( n, n );
product.Zeros ( );
```

Listing 9: Creates Λ.

```
// Creates lambda. Since A^(-1) is needed, it is retrieved by a series of
// linear equations.
// col_entries and row_entries contain the structure of the (sparse) mass
// matrix, i.e. they contain the indices of all dofs which are not zero.
for ( int j = 0; j < n; j++ )
{
    for ( int i = 0; i < static_cast < int > ( col_entries_.size ( ) ); i++ )
    {
        if ( col_entries_[i] == j )
        {
            double value_;
            mass_matrix_->diagonal ( ).get_value ( row_entries_[i], j, &value_ );
            row_->SetValues ( &row_entries_[i], 1, &value_ );
```

```
        }
    }

    // Solves A*L = M with L being lambda. Every column of M is used as one
    // RHS to get one column of L.
    solver_->Solve ( *row_, coupled_sol_ );

    for ( int i = 0; i < n; i++ )
    {
        double val;
        coupled_sol_->GetValues ( &i, 1, &val );
        lambda ( j, i ) = val;
    }
    row_->Zeros ( );
    coupled_sol_->Zeros ( );
}
```

Listing 10: Computes the restricted Operator $T\Lambda$.

```
// Sets the unnecessary rows of lambda to 0.
for ( int i = 0; i < n; i++ )
{
    if ( dof_count_trajectory[i] == -1 )
    {
        for ( int j = 0; j < n; j++ )
        {
            lambda ( i, j ) = 0;
        }
    }
}

std::vector<double> right_hand_side ( n, 0 );
solution_vec.resize ( n );
lambda.transpose_me ( lambda_trans );
lambda_trans.MatrixMult ( lambda, product );
```

Listing 11: Computes alpha.

```
// Computes alpha. Alpha = 0.1 * max(product_ii)
double alpha = 0;
for ( int i = 0; i < n; i++ )
{
    if ( alpha < product ( i, i ) )
    {
        alpha = product ( i, i );
    }
}
alpha = 0.1 * alpha;
```

Listing 12: Computes D.

```
double penalty_val = 100;

for ( int i = 0; i < n; i++ )
{
    if ( penalty_dof_count[i] != -1 )
    {
        penalty_mat ( i, i ) = penalty_val * alpha;
    }
    else
```

```
    {
        penalty_mat ( i, i ) = alpha;
    }
}

product.Add ( penalty_mat );
lambda_trans.VectorMult ( u_trajectory, right_hand_side );
product.Solve ( right_hand_side, solution_vec );
delete solver_;
```

4 Numerical Example

4.1 Direct Problem

In this example we aim to solve the direct problem numerically. Therefore we consider a block $\Omega := (0, 10) \times (0, 20) \times (2, -6)$, which consists of two layers: air layer and ground layer. Let the conductivity $\sigma \in L^{\infty}_{>0}(\overline{\Omega})$ given by

$$\sigma(x) := \begin{cases} 1\, Sm^{-1}, & \text{if } x \in (0, 20) \times (0, 10) \times (0, 2), \\ 2\, Sm^{-1}, & \text{if } x \in (0, 20) \times (0, 10) \times (-6, 0), \end{cases} \tag{17}$$

which represents the air and the ground layer, cf. Listing 5. The source is modeled by

$$\boldsymbol{f}(x) := \begin{cases} 1, & \text{if } x \in (6, 10) \times (4, 6) \times (-2, -3), \\ -1, & \text{if } x \in (10, 14) \times (4, 6) \times (-2, -3), \end{cases} \tag{18}$$

so that $\boldsymbol{f}$ has dipole character, cf. Listing 4. In Figure 2 two isosurfaces of $\boldsymbol{u}$ are visualized. One can clearly see its dipole character and the role of the Robin boundary condition.

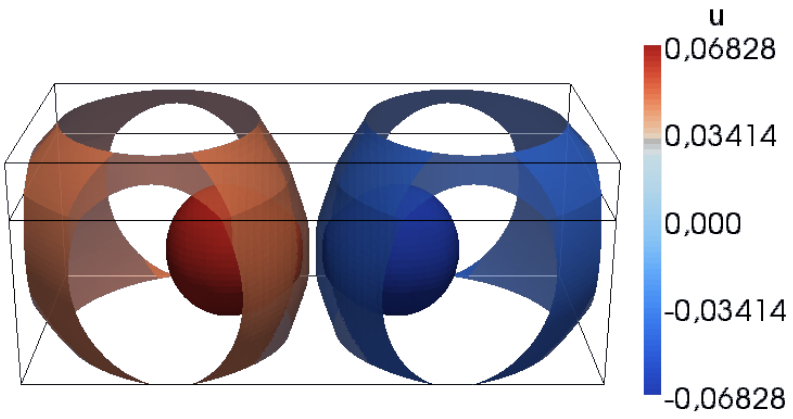

Figure 2: Solution $\boldsymbol{u}$ of the direct problem; Three isosurfaces.

4.2 Inverse Problem

Here we solve the corresponding inverse problem and emphasize the effect of α. Firstly, we have to choose the measuring grid points. For the sake of simplicity we measure $\boldsymbol{u}|_{\Gamma_h}$ on two planes

$$\Gamma_h := \{(x, y, z) \in \mathbb{R}^3 : 0 \leq x \leq 20,\ 0 \leq y \leq 10,\ z = 1 \text{ or } z = 1.5\},$$

i.e. on every grid point in these planes. These planes and the dipole can be seen in Figure 3. Now it is easy to implement the discrete restriction operator $\boldsymbol{T}$, see (14) and cf. Listing 8.

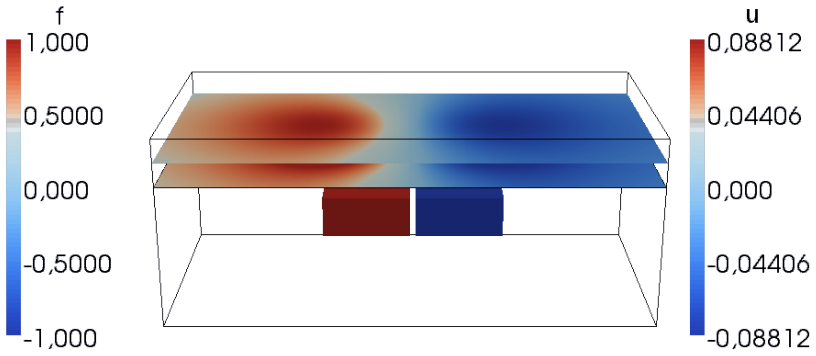

Figure 3: Two planes on which the electric potential is measured and the dipole which we choose.

In addition, we have to implement the penalty operator $\boldsymbol{D}$, which includes the following information. We know that the support of f must be underneath the ground surface and it cannot contact the boundary. Consequently, we penalize the boundary grid points and the grid points in the air. Let p_i^x be the x-coordinate, p_i^y be the y-coordinate and p_i^z be the z-coordinate of the i-th grid point p, then we chose a diagonal matrix $\boldsymbol{D}$ with corresponding matrix elements

$$\boldsymbol{D}_{i,i} := \begin{cases} 1, & \text{if } 0 < p_i^x < 20, \text{ or } 0 < p_i^y < 10, \text{ or } -6 < p_i^z < 0 \\ 100, & \text{else,} \end{cases}$$

see (16) and cf. Listing 12. Using this diagonal matrix inside the regularized term we get a good pseudo-solutions.

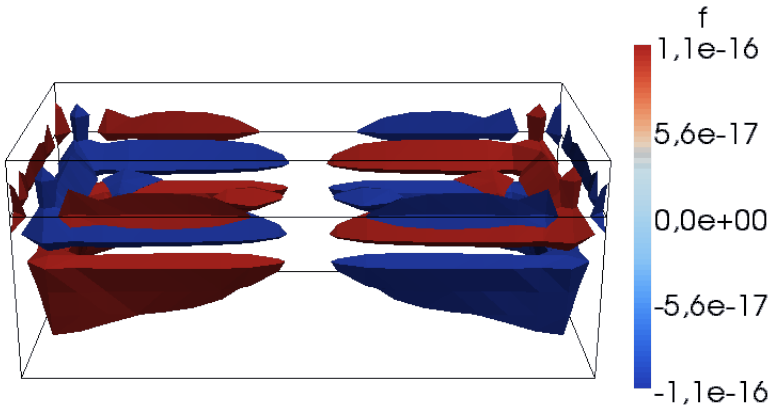

Figure 4: Pseudo-solution $\boldsymbol{f}_\alpha$ for α which is too small.

Recalling we are searching for an optimal α, which is large enough to apply CG method and small enough to save the relation to the true solution. To show the influence of the regularization parameter α to the pseudo-solution we are going to reconstruct the source $\boldsymbol{f}$ for three different

α. In the first case we set $\alpha_1 = 10^{-6} \cdot \max_{i=1,\dots,n} |\mathbf{\Lambda}_{i,i}|$, so that the inverse cannot be computed. The result can be seen in Figure 4. The second case shows a well fitted regularization parameter $\alpha_2 = 0.1 \cdot \max_{i=1,\dots,n} |\mathbf{\Lambda}_{i,i}|$, see Figure 5 and cf. Listing 11, and in the third case $\alpha_3 = 1.000 \cdot \max_{i=1,\dots,n} |\mathbf{\Lambda}_{i,i}|$ regularized the pseudo-solution too strong, so that $\boldsymbol{f}_\alpha$ is approximately zero almost everywhere, as seen in Figure 6.
Note that if the L-curve criterion would be applied to determine the optimal α, one has to compute the pseudo-solution $\boldsymbol{f}_\alpha$ for several α. Subsequently, this heuristic parameter choise rule determines the best of all these solutions. That's why finding an optimal α by the L-curve criterion is expensive, but comfortable.

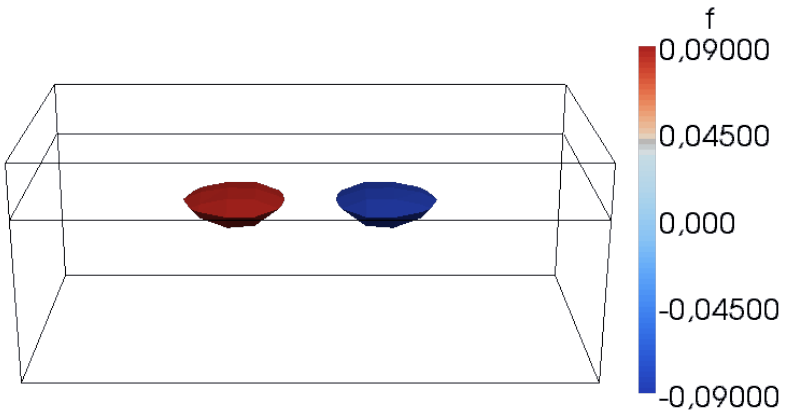

Figure 5: Pseudo-Solution $\boldsymbol{f}_\alpha$ for α which fits the scenario.

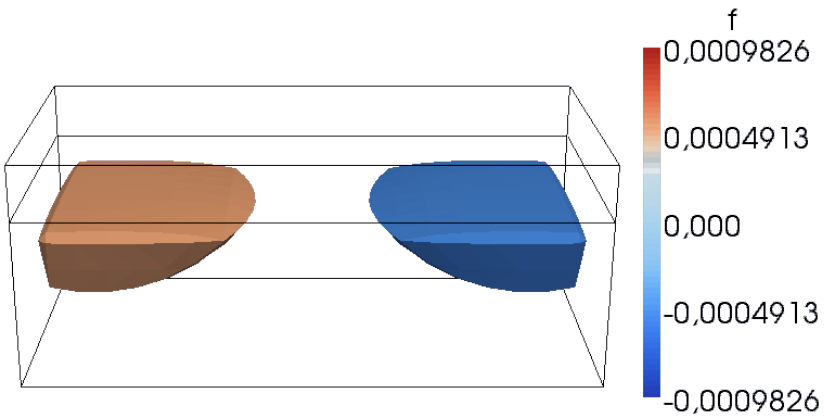

Figure 6: Pseudo-Solution $\boldsymbol{f}_\alpha$ for α which is too large.

References

[1] D. Braess: *Finite Elemente*, Theorie, schnelle Löser und Anwendungen in der Elastizitätstheorie, Springer-Verlag, 2003.

[2] D.Gilbarg, N.S.Trudinger: Elliptic Partial Differential Equations of Second order, Second Edition, Springer-Verlag, Berlin, 2001

[3] J.Hadamard, Four lectures on mathematics, Columbia University Press, New York, 1915.

[4] A. Kirsch: *An Introduction to the Mathematical Theory of Inverse Problems*, Springer-Verlag, 1996.

[5] A. Rieder: *Keine Probleme mit Inversen Problemen* Eine Einführung in ihre stabile Lösung, Vieweg Verlag, 2003

[6] S.Salsa: Partial Differential Equations in Action, From Modelling to Theory, Springer, 2008

[7] A.Sommer: Passive Airborne Oil Exploration - Theorie und Numerik eines linearen inversen Problems, KIT, Diploma Theses, 2012.

[8] A.N.Tikhonov, Solution of incorrectly formulated problems and the regularization method, Soviet Mathematics Doklady 4(1963), 1035-1038