R. Hiptmair
L. Scarabosio
C. Urzua Torres

Spring Term 2015

Numerical Methods for Partial Differential Equations

ETH Zürich
D-MATH

Exam Preparation Sheet 14

Introduction. This problem sheet collects a few rather complex problems involving both theoretical and C++ implementation parts. They all have a concrete application background and touch on various topics addressed during the course. So they are very well suited to test one's skills when preparing for the main examination on August 22, 2015.

Problem 14.1 Far field computation

In [NPDE, Section 5.6.1] we discussed the convergence of linear output functionals. More precisely, by means of duality techniques, one can prove that the convergence of the functional is of one order higher than the convergence of the solution in the energy norm ([NPDE, Thm. 5.6.7]).

In [NPDE, Section 5.6.2], the computation of heat boundary flux was considered and it was demonstrated how a modification of an output functional can render it continuous with respect to the energy norm, which is an essential prerequisite for applying the duality argument. In this problem, we pursue a similar policy for the computation of another linear output functional, which is important in the simulation of electromagnetic waves.

This problem focuses on time-harmonic wave propagation in linear media. As in Problem 13.4, in this case all time-dependent fields can be represented as

$$U(\boldsymbol{x},t) = \mathrm{Re}(u(\boldsymbol{x})\exp(\mathrm{i}\omega t)) \;, \tag{14.1.1}$$

where $u(\boldsymbol{x}) \in \mathbb{C}$ is a *complex amplitude*, $\omega > 0$ stands for the so-called angular frequency, and Re extracts the real part of a complex number. All equations will be equations for complex amplitudes, from which the actual wave can be recovered by (14.1.1). Hence, in this problem, all unknowns will be *complex valued*. The file `Pardiso.hpp` in the repository has been modified so that it can also handle complex-valued matrices and vectors.

Note: If you work on a local copy of the NPDE library, you must substitute your old `Pardiso.hpp` header with the new one.

The propagation of the so-called TE-mode (TE for transverse electric) of an electromagnetic wave and its interaction with an (infinitely long and straight) penetrable scatterer can be described by the following two-dimensional second-order elliptic boundary value problem for the complex amplitude u of the axial component of the electric field:

$$\begin{aligned} -\Delta u - k^2(\boldsymbol{x})u &= f && \text{in } D \subset \mathbb{R}^2 \;, \\ \mathbf{grad}\, u \cdot \boldsymbol{n} + \mathrm{i}k_d u &= 0 && \text{on } \partial D \;. \end{aligned} \tag{14.1.2}$$

Here, i is the imaginary unit, $D \subset \mathbb{R}^2$ an artificially truncated bounded computational domain, and the piecewise constant discontinuous coefficient $k(\boldsymbol{x})$ is the *wave number*, given by

$$k(\boldsymbol{x}) = \begin{cases} k_s, & \text{for } \boldsymbol{x} \in S \ , \\ k_d, & \text{for } \boldsymbol{x} \in D \setminus \overline{S} \ , \end{cases} \qquad k_s, k_d > 0 \ . \tag{14.1.3}$$

In the following, we use the concrete values $k_s = \sqrt{2}k_0$, $k_d = k_0$, $k_0 = \frac{2\pi}{3}$.

The bounded sub-domain $S \subset D$ is the space occupied by the scattering object, see Figure 14.1. The source function $f = f(\boldsymbol{x})$ is given by

$$f(\boldsymbol{x}) = (k^2(\boldsymbol{x}) - k_d^2)u_i(\boldsymbol{x}) \ , \tag{14.1.4}$$

with the so-called incident wave

$$u_i(\boldsymbol{x}) = \mathrm{e}^{\mathrm{i}k_d x_1} \ , \tag{14.1.5}$$

a plane wave impinging from the right, see Figure 14.1.

Remark: The solution u of (14.1.2) represents the so-called scattered field, that is, the perturbation of the incident field due to the presence of the scattering objects. The total field that can be measured is described by the complex amplitude $u_{\mathrm{tot}} = u + u_i$.

Remark: Actually the wave propagation problem is posed on the unbounded domain $\mathbb{R}^2$, which, however, is outside the scope of every mesh based discretization. Therefore, computations are done on an artificially truncated domain D, and one tries to take into account the effect of the discarded part of space $\mathbb{R}^2 \setminus \overline{D}$ by means of so-called *absorbing boundary conditions*. The Robin boundary condition in (14.1.2) is a simple variant of these.

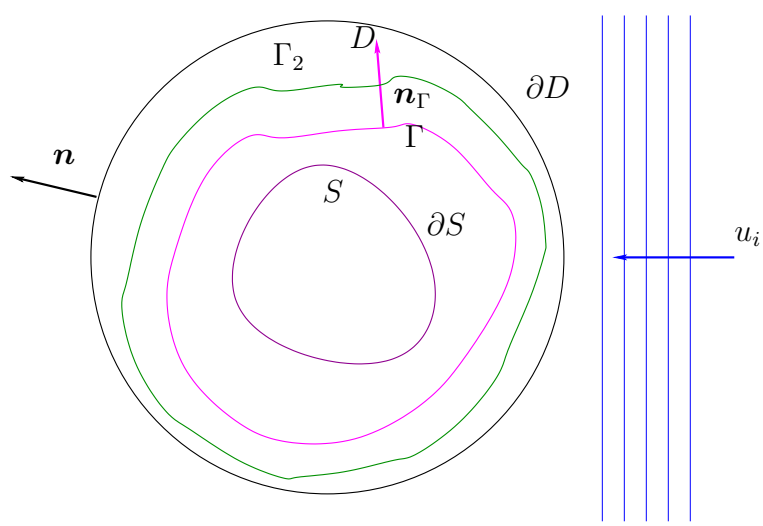

Figure 14.1: Arrangement for 2D scattering problem

Once the scattered field has been computed, the *far field mapping* $F : H^2(D) \mapsto \mathcal{C}^\infty(\mathbb{S}^1)$ is given by

$$F(u)(\hat{\boldsymbol{x}}) = \frac{\mathrm{e}^{\mathrm{i}\pi/4}}{\sqrt{8\pi k_d}} \int_\Gamma u(\boldsymbol{y})(\mathbf{grad}\, w_{\hat{\boldsymbol{x}}})(\boldsymbol{y}) \cdot \boldsymbol{n}_\Gamma(\boldsymbol{y}) \\ - (\mathbf{grad}\, u)(\boldsymbol{y}) \cdot \boldsymbol{n}_\Gamma(\boldsymbol{y}) w_{\hat{\boldsymbol{x}}}(\boldsymbol{y}) \ \mathrm{d}S(\boldsymbol{y}) \ , \quad \hat{\boldsymbol{x}} \in \mathbb{S}^1 \ . \tag{14.1.6}$$

with

$$w_{\hat{\boldsymbol{x}}}(\boldsymbol{y}) = \exp(-\mathrm{i}k_d \hat{\boldsymbol{x}} \cdot \boldsymbol{y}) \ , \quad \hat{\boldsymbol{x}} \in \mathbb{S}^1, \ \boldsymbol{y} \in \mathbb{R}^3 \ . \tag{14.1.7}$$

This means that the image of F is a function defined on the unit circle $\mathbb{S}^1 = \{\hat{\boldsymbol{x}} \in \mathbb{R}^2 \mid \|\hat{\boldsymbol{x}}\| = 1\}$. Here, $\Gamma \subset D \setminus \overline{S}$ is a simple closed path around the scatterer with (exterior) unit normal vector field $\boldsymbol{n}_\Gamma$, see Figure 14.1. For fixed $\hat{\boldsymbol{x}} \in \mathbb{S}^1$ $u \to F(u)(\hat{\boldsymbol{x}})$ is a linear output functional depending on the solution of (14.1.2). The objective of this problem is to investigate its stable numerical evaluation.

Remark: The far field $F(u_s)$ represents the intensity of the scattered field at large distances away S. The integral of $|F(u)|^2$ over some part of the sphere tells us the power carried through that sector by the wave coming back from S.

I. The first part of this problem is concerned with preparatory considerations about (14.1.2) and the far field mapping.

(14.1a) State the variational formulation of (14.1.2) complete with appropriate function spaces.

HINT: The derivation is given in [NPDE, Ex. 2.9.6]. You simply ignore the fact that we deal with $\mathbb{C}$-valued functions. Also recall subproblem (13.4e).

Solution: Using integration by parts and the boundary condition at ∂D we get the variational formulation:

Find $u \in H^1(D)$ such that

$$\int_D \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}x - \int_D k(\boldsymbol{x})^2 uv \,\mathrm{d}x + \int_{\partial D} \mathrm{i}k(\boldsymbol{x}) uv \,\mathrm{d}S = \int_D fv \,\mathrm{d}x \quad \forall v \in H^1(D). \tag{14.1.8}$$

(14.1b) Prove that, if $f \equiv 0$ in (14.1.2), then $u(\boldsymbol{x}) = 0$ and $\mathbf{grad}\, u \cdot \boldsymbol{n}(\boldsymbol{x}) = 0$ for $\boldsymbol{x} \in \partial D$.

HINT: Here you have to use complex conjugation $z \mapsto \overline{z}$ at some point and that $|u(\boldsymbol{x})|^2 = u(\boldsymbol{x})\overline{u}(\boldsymbol{x})$. Test with a function depending on u and consider imaginary and real part of the resulting equation separately.

Solution:

Considering $v = u$ and the products in (14.1.8) as scalar products between *complex* numbers, we obtain for the homogeneous equation:

$$\int_D |\,\mathbf{grad}\, u|^2 - k(\boldsymbol{x})^2 |u|^2 \,\mathrm{d}x + \int_{\partial D} \mathrm{i}k(\boldsymbol{x}) |u|^2 \,\mathrm{d}S = 0.$$

If we take the imaginary part, this leads to

$$\int_{\partial D} k(\boldsymbol{x}) |u|^2 \,\mathrm{d}S = 0$$

and thus $u|_{\partial D} \equiv 0$. From the boundary condition in (14.1.2), we also obtain that $\mathbf{grad}\, u \cdot \boldsymbol{n} \equiv 0$ on ∂D.

The result in subproblem (14.1b) implies uniqueness of the solution to the variational problem from subtask (14.1a). Indeed, if u_1 and u_2 are two solutions to the variational formulation that you derived, then $u_2 - u_1$ satisfies the associated homogeneous equation. From subproblem (14.1b), we have that $u_1 - u_2 \equiv 0$ and $\mathbf{grad}(u_1 - u_2) \cdot \boldsymbol{n} \equiv 0$ on ∂D. Then, the so-called *unique continuation principle* implies that $u_1 - u_2 \equiv 0$ in the whole D.

(14.1c) Explain why, for fixed $\hat{\boldsymbol{x}} \in \mathbb{S}^1$, the functional $u \mapsto F(u)(\hat{\boldsymbol{x}})$ is not continuous on $H^1(D \setminus \overline{S})$.

HINT: You may appeal to the result presented in [NPDE, § 5.6.13].

Solution: We denote, for fixed $\hat{\boldsymbol{x}} \in \mathbb{S}^1$,

$$F_2(u) = \int_\Gamma \mathbf{grad}\, u(\boldsymbol{y}) \cdot \boldsymbol{n}_\Gamma(\boldsymbol{y}) \mathrm{e}^{-\mathrm{i}k_d \hat{\boldsymbol{x}} \cdot \boldsymbol{y}} \ \mathrm{d}S(\boldsymbol{y}) = \int_\Gamma \mathbf{grad}\, u(\boldsymbol{y}) \cdot \boldsymbol{n}_\Gamma(\boldsymbol{y}) \psi(\boldsymbol{y}) \ \mathrm{d}S(\boldsymbol{y})$$

where $\psi(\boldsymbol{y}) = \mathrm{e}^{-\mathrm{i}k_d \hat{\boldsymbol{x}} \cdot \boldsymbol{y}}$. Then, from [NPDE, § 5.6.13] we get the unboundedness of $F_2(u)$.

II. In the second part of this problem we devise a finite element discretization of (14.1.2) based on the linear Lagrangian finite element space $\mathcal{S}_1^0(\mathcal{M})$, where $\mathcal{M}$ is a triangular mesh of D, which is compatible with ∂S in the sense that ∂S is represented by a closed polygon ∂S_N consisting of edges of $\mathcal{M}$. This permits us to associated every cell of $\mathcal{M}$ with either S or $D \setminus \overline{S}$, depending on which side of ∂S_N it is located.

The location of mesh cells is encoded in a vector of integers `ElemFlag`. Each subdomain has a flag, namely the flag 1 is associated to $D \setminus S$, and the flag 2 is associated to S. Then the vector `ElemFlag` has length equal to the number of elements and

$$\begin{aligned} &\texttt{ElemFlag[k]==1} \quad \Leftrightarrow \quad \text{mesh cell with global index } k \text{ belongs to } D \setminus S\ . \\ &\texttt{ElemFlag[k]==2} \quad \Leftrightarrow \quad \text{mesh cell with global index } k \text{ belongs to } S\ . \end{aligned} \tag{14.1.9}$$

Throughout we are going to use the standard tent function basis of $\mathcal{S}_1^0(\mathcal{M})$.

(14.1d) We want to implement a class representing the wave number $-k^2 = -k^2(\boldsymbol{x})$, $\boldsymbol{x} \in D$, with k as given in (14.1.3). To that end, complete the class

```
template <class GridView>
class KappaFunc{
public:
  using calc_t=double;

  KappaFunc(GridView const& gv, std::vector<int> const& ElemFlags,
            calc_t ks_sq, calc_t kd_sq)
    : idset(gv.indexSet()), ElemFlag(ElemFlags), ks_sq_(ks_sq),
      kd_sq_(kd_sq) {}

  template <class Element>
  calc_t operator()(Element const& e) const{

  }

private:
  typename GridView::IndexSet const& idset;
  std::vector<int> const& ElemFlag;
  calc_t ks_sq_,kd_sq_;
};
```

contained in the header `KappaFunc.hpp`, implementing the method

```
template <class Element>
      calc_t operator()(Element const& e) const
```

that, given an element, computes the value of $-k^2 = -k^2(\boldsymbol{x})$ (we assume that the value of k cannot change inside an element).

HINT: Use the information contained in the vector ElemFlag.

Solution: See Listing 14.1 for the code.

(14.1e) Due to the boundary condition in (14.1.2), we also to evaluate $k = k(\boldsymbol{x})$ on the boundary, that is for $\boldsymbol{x} \in \partial D$. To this aim, complete the class

```
 class KappaBdFunc{
public:
  using calc_t=double;

  KappaBdFunc(calc_t kd_sq)
    : kd_sq_(kd_sq) {}

  template <class Element>
  std::complex<calc_t> operator()(Element const& e) const{

  }

private:
  calc_t kd_sq_;
};
```

contained in the header `KappaFunc.hpp`, with the implementation of the method

```
template <class Element>
      calc_t operator()(Element const& e) const
```

that, given an element, computes the value of $k = k(\boldsymbol{x})$. This function will be called only to evaluate k boundary edges, so you don't have to make the distinction made in (14.1.3).

Solution: See Listing 14.1 for the code.

Listing 14.1: Implementation for `KappaFunc`

```
#ifndef KAPPAFUNC_HPP_
#define KAPPAFUNC_HPP_

// Functor for the coefficient function kappa
template <class GridView>
class KappaFunc{
public:
  using calc_t=double;

  KappaFunc(GridView const& gv, calc_t kd_sq, calc_t ks_sq,
      std::vector<int> const& ElemFlags)
    : idset(gv.indexSet()), kd_sq_(kd_sq), ks_sq_(ks_sq),
        ElemFlag(ElemFlags) {;}
```

```

  template <class Element>
  calc_t operator()(Element const& e) const{
    return ElemFlag[idset.index(e)]==2?-1.*ks_sq_:-1.*kd_sq_;
  }

private:
  typename GridView::IndexSet const& idset;
  std::vector<int> const& ElemFlag;
  calc_t ks_sq_,kd_sq_;
};

class KappaBdFunc{
public:
  using calc_t=double;

  KappaBdFunc(calc_t kd_sq)
    : kd_sq_(kd_sq) {}

  template <class Element>
  std::complex<calc_t> operator()(Element const& e) const{
    return {0,std::sqrt(kd_sq_)};
  }

private:
  calc_t kd_sq_;
};

#endif
```

(14.1f) We now implement a class for the source function $f = f(\boldsymbol{x})$ as defined in (14.1.4). For this, complete the class

```
template <class GridView>
class LoadFunc{
public:
  using calc_t=double;

  LoadFunc(GridView const& gv, std::vector<int> const& ElemFlags,
        calc_t ks_sq, calc_t kd_sq)
    : idset(gv.indexSet()), ElemFlag(ElemFlags), ks_sq_(ks_sq),
      kd_sq_(kd_sq) {}

  template <class Coordinate, class Element>
  std::complex<calc_t> operator()(Coordinate const& x, Element
     const& e) const{
  }

private:
  typename GridView::IndexSet const& idset;
```

```
  std::vector<int> const& ElemFlag;
  calc_t ks_sq_,kd_sq_;
};
```

with the implementation of the method

```
template <class Coordinate, class Element>
std::complex<calc_t> operator()(Coordinate const& x, Element
    const& e) const
```

that computes the source function in the point with coordinates contained in x and belonging to the element e.

HINT: Use again the information contained in the vector ElemFlag.

Solution: See Listing 14.2 for the code.

Listing 14.2: Implementation for `LoadFunc`

```
#ifndef LOADFUNC_HPP_
#define LOADFUNC_HPP_

#include <complex>
#include <cmath>

// Functor for the load vector
template <class GridView>
class LoadFunc{
public:
  using calc_t=double;

  LoadFunc(GridView const& gv, std::vector<int> const& ElemFlags,
        calc_t ks_sq, calc_t kd_sq)
    : idset(gv.indexSet()), ElemFlag(ElemFlags), ks_sq_(ks_sq),
       kd_sq_(kd_sq) {}

  template <class Coordinate, class Element>
  std::complex<calc_t> operator()(Coordinate const& x, Element
      const& e) const{
   if (ElemFlag[idset.index(e)]==2){
     std::complex<calc_t> i={0,1};
     return std::exp(i*std::sqrt(kd_sq_)*x[0])*(ks_sq_-kd_sq_);
    }
    else
      return 0.;
  }

private:
  typename GridView::IndexSet const& idset;
  std::vector<int> const& ElemFlag;
  calc_t ks_sq_,kd_sq_;
};

```

```
#endif
```

(14.1g) Complete the file main.cc, provided in the handout, to compute the finite element solution of (14.1.2)–(14.1.5) and write it in a vtk file.

HINT: Of course you should make use of the functions coded in the previous sub-problems.

HINT: For the local mass matrix, both on the domain and on the boundary, you can use the class LocalMassFarfield from the file LocalMassFarfield.hpp given in the handout. This class is a slight modification of the class LocalMass contained in the folder `local/` of the `npde15` library.

HINT: To assemble the right-hand side, use the class `LocalFunctionFarfield` contained in the header `LocalFunctionFarfield.hpp`, which is a slight modification of the function `LLocalFunction` that you already used to solve the previous assignments. To assemble the finite element matrix, use the routine `MatrixAssembler.hpp` contained in the folder of the handout.

HINT: To select the boundary nodes, you can proceed as in the Radiative Cooling problem that you solved in one of the previous assignments, using the class `LBoundaryNodes`.

Solution: See Listing 14.4 (neglect lines 114-121 and the convergence study).

III. This part of the problem examines the far field mapping (14.1.6) and its accurate evaluation. This will demonstrate another application of the techniques presented in [NPDE, Section 5.6.2].

(14.1h) Refresh yourself on the "cut-off function trick" used to convert the boundary flux functional to the form [NPDE, Eq. (5.6.15)]. Try to understand again, why this "manipulation" is admissible.

Solution: The key to switch from J to J^* in [NPDE, Section 5.6.2] is that $J(u) = J^*(u)$ for the exact solution u of the boundary value problem.

(14.1i) Show that the function $w_{\hat{\boldsymbol{x}}}$ from (14.1.7) satisfies

$$(-\Delta - k_d^2) w_{\hat{\boldsymbol{x}}} = 0 \quad \text{for all } \hat{\boldsymbol{x}} \in \mathbb{S}^1 \ , \tag{14.1.10}$$

where the Laplacian Δ ($\rightarrow$ [NPDE, Rem. 2.5.14]) acts on the independent variable $\boldsymbol{y}$ only.

Solution: Since $\Delta w_{\hat{\boldsymbol{x}}} = -k_d^2 w_{\hat{\boldsymbol{x}}}$, (14.1.10) follows.

(14.1j) In formula (14.1.6), Γ stands for any simple closed path around the scatterer. Show that the far field mapping is independent of the path Γ, more precisely, that, for any fixed $\hat{\boldsymbol{x}} \in \mathbb{S}^1$, you get the same value for $F(u)(\hat{\boldsymbol{x}})$ (u a solution of (14.1.2)), no matter whether you use the paths Γ or Γ_2 drawn in Figure 14.1.

HINT: First prove, appealing to Green's formula [NPDE, Thm. 2.5.9], that for smooth functions u and w on a bounded domain Ω:

$$\int_\Omega \Delta u \, w - u \, \Delta w \, \mathrm{d}\boldsymbol{x} = \int_{\partial\Omega} \mathbf{grad}\, u \cdot \boldsymbol{n} \, w - u \, \mathbf{grad}\, w \cdot \boldsymbol{n} \, \mathrm{d}S \ , \tag{14.1.11}$$

where $\boldsymbol{n}$ is the *outward pointing* unit normal vector field on $\partial\Omega$. Then apply this formula to (14.1.6) for a suitable Ω (enclosed between the two paths) and make use of (14.1.10). Watch the orientation of the normal vectors.

Solution: Using Green's formula:

$$\int_{\Omega} \Delta u \, w \, \mathrm{d}\boldsymbol{x} = \int_{\Omega} \mathbf{grad}\, u \cdot \mathbf{grad}\, w \, \mathrm{d}\boldsymbol{x} - \int_{\partial\Omega} \mathbf{grad}\, u \cdot \boldsymbol{n} \, w \, \mathrm{d}S \tag{14.1.12}$$

and

$$\int_{\Omega} u \, \Delta w \, \mathrm{d}\boldsymbol{x} = \int_{\Omega} \mathbf{grad}\, u \cdot \mathbf{grad}\, w \, \mathrm{d}\boldsymbol{x} - \int_{\partial\Omega} u \, \mathbf{grad}\, w \cdot \boldsymbol{n} \, \mathrm{d}S \tag{14.1.13}$$

Subtracting these two equation we have (14.1.11).

Now, let us consider a subregion $R \subset D$ between two closed paths Γ_1 and Γ_2 around S, which are the outer and inner boundary respectively. Then $\partial R = \Gamma_1 \cup \Gamma_2$ and the outer pointing normal on ∂R is $\boldsymbol{n}_\Gamma$ on Γ_1 and $-\boldsymbol{n}_\Gamma$ on Γ_2.
Using (14.1.11) and considering orientation of the outer pointing normal vector we get:

$$\begin{aligned}
&F_1(u)(\hat{\boldsymbol{x}}) - F_2(u)(\hat{\boldsymbol{x}}) \\
&\quad = \frac{\mathrm{e}^{\mathrm{i}\pi/4}}{\sqrt{8\pi k_d}} \int_{\Gamma_1} u(\boldsymbol{y})(\mathbf{grad}\, w_{\hat{\boldsymbol{x}}})(\boldsymbol{y}) \cdot \boldsymbol{n}_{\Gamma_1}(\boldsymbol{y}) - (\mathbf{grad}\, u)(\boldsymbol{y}) \cdot \boldsymbol{n}_{\Gamma_1}(\boldsymbol{y}) w_{\hat{\boldsymbol{x}}}(\boldsymbol{y}) \;\, \mathrm{d}S(\boldsymbol{y}) + \\
&\qquad + \frac{\mathrm{e}^{\mathrm{i}\pi/4}}{\sqrt{8\pi k_d}} \int_{\Gamma_2} u(\boldsymbol{y})(\mathbf{grad}\, w_{\hat{\boldsymbol{x}}})(\boldsymbol{y}) \cdot \boldsymbol{n}_{\Gamma_2}(\boldsymbol{y}) - (\mathbf{grad}\, u)(\boldsymbol{y}) \cdot \boldsymbol{n}_{\Gamma_2}(\boldsymbol{y}) w_{\hat{\boldsymbol{x}}}(\boldsymbol{y}) \;\, \mathrm{d}S(\boldsymbol{y}) \\
&\quad = \frac{\mathrm{e}^{\mathrm{i}\pi/4}}{\sqrt{8\pi k_d}} \int_{R} \Delta u \, w_{\hat{\boldsymbol{x}}} - u \, \Delta w_{\hat{\boldsymbol{x}}} \, \mathrm{d}\boldsymbol{x} = 0F
\end{aligned}$$

where the last step is due to (14.1.10).

(14.1k) As in [NPDE, Section 5.6.2], we choose a cut-off function $\psi \in \mathcal{C}^0(D \setminus S) \cap H^1(D \setminus \overline{S})$ satisfying

$$\psi\big|_{\partial D} = 1 \quad , \quad \psi_{\partial S} = 0 \quad , \quad \mathbf{grad}\, \psi \text{ bounded.} \tag{14.1.14}$$

Show that for $u, w \in H^1(D \setminus \overline{S})$ with $(-\Delta - k_d^2)w = 0$ in $D \setminus \overline{S}$, we have

$$\begin{aligned}
&\int_{\partial D} u(\boldsymbol{y})(\mathbf{grad}\, w)(\boldsymbol{y}) \cdot \boldsymbol{n}(\boldsymbol{y}) \, \mathrm{d}S(\boldsymbol{y}) \\
&\qquad\qquad = \int_{D \setminus \overline{S}} u(\boldsymbol{y})\psi(\boldsymbol{y}) k_d^2 w(\boldsymbol{y}) + \mathbf{grad}(u\psi)(\boldsymbol{y}) \cdot \mathbf{grad}\, w(\boldsymbol{y}) \, \mathrm{d}\boldsymbol{y} \,.
\end{aligned} \tag{14.1.15}$$

HINT: Use Green's formula [NPDE, Thm. 2.5.9].

Solution: We have:

$$\begin{aligned}
&\int_{\partial D} u(\boldsymbol{y})(\mathbf{grad}\, w)(\boldsymbol{y}) \cdot \boldsymbol{n}(\boldsymbol{y})\, \mathrm{d}S(\boldsymbol{y}) \\
&\quad = \int_{\partial D} \psi(\boldsymbol{y}) u(\boldsymbol{y})(\mathbf{grad}\, w)(\boldsymbol{y}) \cdot \boldsymbol{n}(\boldsymbol{y})\, \mathrm{d}S(\boldsymbol{y}) + \int_{\partial S} \psi(\boldsymbol{y}) u(\boldsymbol{y})(\mathbf{grad}\, w)(\boldsymbol{y}) \cdot \boldsymbol{n}(\boldsymbol{y})\, \mathrm{d}S(\boldsymbol{y}) \\
&\quad = -\int_{D\setminus\bar{S}} \Delta w\, (\psi u)(\boldsymbol{y})\, \mathrm{d}\boldsymbol{y} + \int_{d\setminus\bar{S}} \mathbf{grad}(\psi u)(\boldsymbol{y}) \cdot \mathbf{grad}\, w(\boldsymbol{y})\, \mathrm{d}\boldsymbol{y} \\
&\quad = \int_{D\setminus\overline{S}} u(\boldsymbol{y})\psi(\boldsymbol{y}) k_d^2 w(\boldsymbol{y}) + \mathbf{grad}(u\psi)(\boldsymbol{y}) \cdot \mathbf{grad}\, w(\boldsymbol{y})\, \mathrm{d}\boldsymbol{y}\,.
\end{aligned}$$

(14.1l) Show that for the far field mapping $F(u)$ from (14.1.6) holds

$$F(u)(\hat{\boldsymbol{x}}) = \frac{\mathrm{e}^{\mathrm{i}\pi/4}}{\sqrt{8\pi k_d}} \int_{D\setminus\overline{S}} \mathbf{grad}\, \psi(\boldsymbol{y})(u(\boldsymbol{y})\,(\mathbf{grad}\, w_{\hat{\boldsymbol{x}}})(\boldsymbol{y}) - (\mathbf{grad}\, u)(\boldsymbol{y})\, w_{\hat{\boldsymbol{x}}}(\boldsymbol{y}))\, \mathrm{d}\boldsymbol{y}\,, \tag{14.1.16}$$

for any $\hat{\boldsymbol{x}} \in \mathbb{S}^1$, provided that u solves (14.1.2).

HINT: First switch to the integration path ∂D, using the result of sub-problem (14.1j). Then apply (14.1.15) taking into account (14.1.10).

Solution: Because of (14.1j), we can consider ∂D as integration path. Then, since (14.1.10) holds, we can apply (14.1.15) to both integrals in (14.1.6). The result follows applying the chain rule to $\mathbf{grad}(u\psi)$.

(14.1m) The result of the previous sub-problem suggests that we consider the modified far field mapping

$$F^*(u)(\hat{\boldsymbol{x}}) = \frac{\mathrm{e}^{\mathrm{i}\pi/4}}{\sqrt{8\pi k_d}} \int_{D\setminus\overline{S}} \mathbf{grad}\, \psi(\boldsymbol{y}) \cdot (u(\boldsymbol{y})\,(\mathbf{grad}\, w_{\hat{\boldsymbol{x}}})(\boldsymbol{y}) - (\mathbf{grad}\, u)(\boldsymbol{y})\, w_{\hat{\boldsymbol{x}}}(\boldsymbol{y}))\, \mathrm{d}\boldsymbol{y}\,. \tag{14.1.17}$$

Why is $u \mapsto F^*(u)(\hat{\boldsymbol{x}})$ for fixed $\hat{\boldsymbol{x}} \in \mathbb{S}^1$ a *continuous* linear functional on the energy space $H^1(D \setminus \overline{S})$?

Solution: Setting $C = \frac{\mathrm{e}^{\mathrm{i}\pi/4}}{\sqrt{8\pi k_d}}$, we have:

$$\begin{aligned}
|F^*(u)(\hat{\boldsymbol{x}})|^2 &\le C^2 \Big(\sup_{\boldsymbol{y}} \|\mathbf{grad}\, \psi(\boldsymbol{y})\| \int_{D\setminus\bar{S}} |u(\boldsymbol{y})| \|\mathbf{grad}\, w_{\hat{\boldsymbol{x}}}(\boldsymbol{y})\| + \|\mathbf{grad}\, u(\boldsymbol{y})\| |w_{\hat{\boldsymbol{x}}}(\boldsymbol{y})|\, \mathrm{d}\boldsymbol{y} \Big)^2 \\
&\le C_1 \Big(\|u\|^2_{L^2(D\setminus\bar{S})} |w_{\hat{\boldsymbol{x}}}|^2_{H^1(D\setminus\bar{S})} + \|w_{\hat{\boldsymbol{x}}}\|^2_{L^2(D\setminus\bar{S})} |u|^2_{H^1(D\setminus\bar{S})} \Big) \\
&\le C_2 (\|u\|^2_{L^2(D\setminus\bar{S})} + |u|^2_{H^1(D\setminus\bar{S})}) = C_2 \|u\|^2_{H^1(D\setminus\bar{S})}
\end{aligned}$$

For the second inequality we used the boundedness of $\mathbf{grad}\, \psi$ (from (14.1.20)), Cauchy-Schwarz inequality and the fact that, for $a, b \ge 0$, $(a+b)^2 \le 2(a^2+b^2)$; thus, we set $C_1 = 2C^2 \big(\sup_{\boldsymbol{y}} \|\mathbf{grad}\, \psi(\boldsymbol{y})\|\big)^2$. For the last inequality, the fact that $w_{\hat{\boldsymbol{x}}} \in H^1(d \setminus \bar{S})$ has been exploited.

(14.1n) In the previous sub-problem we have seen that F^* is bounded on $H^1(D \setminus \overline{S})$. Well, we can even do better, when choosing special cut-off functions, which satisfy, in addition to (14.1.20),

$$\psi \equiv 1 \quad \text{close to } \partial D \ , \quad \psi \equiv 0 \quad \text{close to } \partial S \ , \quad \psi \in \mathcal{C}^2(\overline{D}) \ . \tag{14.1.18}$$

Show that for this choice

$$F^*(u)(\hat{\boldsymbol{x}}) = \frac{\mathrm{e}^{\mathrm{i}\pi/4}}{\sqrt{8\pi k_d}} \int\limits_{D\setminus\overline{S}} u(\boldsymbol{y})(\mathbf{grad}\,\psi(\boldsymbol{y}) \cdot (\mathbf{grad}\, w_{\hat{\boldsymbol{x}}})(\boldsymbol{y}) + \operatorname{div}(w_{\hat{\boldsymbol{x}}}\,\mathbf{grad}\,\psi)(\boldsymbol{y}))\,\mathrm{d}\boldsymbol{y} \ . \tag{14.1.19}$$

Explain, why $u \mapsto F^*(u)(\hat{\boldsymbol{x}})$ is even bounded on $L^2(\Omega)$, if (14.1.18) is satisfied.

HINT: Hardly surprising, an application of Green's formula from [NPDE, Thm. 2.5.9] does the trick.

Solution: Applying Green's formula [NPDE, Thm. 2.5.9] to the second term in (14.1.17) we get:

$$\begin{aligned}
\int\limits_{D\setminus\bar{S}} \mathbf{grad}\,\psi(\boldsymbol{y}) \cdot \mathbf{grad}\,u(\boldsymbol{y})\, w_{\hat{\boldsymbol{x}}}\,\mathrm{d}\boldsymbol{y} &= \int\limits_{D\setminus\bar{S}} (w_{\hat{\boldsymbol{x}}}\,\mathbf{grad}\,\psi(\boldsymbol{y})) \cdot \mathbf{grad}\,u(\boldsymbol{y})\,\mathrm{d}\boldsymbol{y} \\
&= \int\limits_{D\setminus\bar{S}} \operatorname{div}(w_{\hat{\boldsymbol{x}}}\,\mathbf{grad}\,\psi(\boldsymbol{y}))\,u(\boldsymbol{y})\,\mathrm{d}\boldsymbol{y} + \int\limits_{\partial(D\setminus\bar{S})} w_{\hat{\boldsymbol{x}}}\,\mathbf{grad}\,\psi(\boldsymbol{y}) \cdot \boldsymbol{n}\,u\,\mathrm{d}\boldsymbol{y} \\
&= \int\limits_{D\setminus\bar{S}} \operatorname{div}(w_{\hat{\boldsymbol{x}}}\,\mathbf{grad}\,\psi(\boldsymbol{y}))\,u(\boldsymbol{y})\,\mathrm{d}\boldsymbol{y}
\end{aligned}$$

where the boundary integral vanishes because of (14.1.18). Inserting this expression in (14.1.17), we get (14.1.19). The new formula is bounded on $L^2(\Omega)$, because, denoting again $C = \frac{\mathrm{e}^{\mathrm{i}\pi/4}}{\sqrt{8\pi k_d}}$ and proceding as in the previous subproblem:

$$\begin{aligned}
|F^*(u)(\hat{\boldsymbol{x}})|^2 &\leq 2C^2\Bigg(\Big(\sup_{\boldsymbol{y}}\|\mathbf{grad}\,\psi(\boldsymbol{y})\|\Big)^2\|u\|^2_{L^2(D\setminus\bar{S})}|w_{\hat{\boldsymbol{x}}}|^2_{H^1(D\setminus\bar{S})} \\
&\quad + \|u\|^2_{L^2(D\setminus\bar{S})} \int\limits_{D\setminus\bar{S}} (\operatorname{div}(w_{\hat{\boldsymbol{x}}}\,\mathbf{grad}\,\psi(\boldsymbol{y})))^2\,\mathrm{d}\boldsymbol{y}\Bigg) \\
&= \|u\|^2_{L^2(D\setminus\bar{S})} 2C^2\Bigg(\Big(\sup_{\boldsymbol{y}}\|\mathbf{grad}\,\psi(\boldsymbol{y})\|\Big)^2|w_{\hat{\boldsymbol{x}}}|^2_{H^1(D\setminus\bar{S})} + \int\limits_{D\setminus\bar{S}} (\operatorname{div}(w_{\hat{\boldsymbol{x}}}\,\mathbf{grad}\,\psi(\boldsymbol{y})))^2\,\mathrm{d}\boldsymbol{y}\Bigg)
\end{aligned}$$

For the integral $\int\limits_{D\setminus\bar{S}} (\operatorname{div}(w_{\hat{\boldsymbol{x}}}\,\mathbf{grad}\,\psi(\boldsymbol{y})))^2\,\mathrm{d}\boldsymbol{y}$, we have $\operatorname{div}(w_{\hat{\boldsymbol{x}}}\,\mathbf{grad}\,\psi(\boldsymbol{y})) = \mathbf{grad}\,w_{\hat{\boldsymbol{x}}}(\boldsymbol{y}) \cdot \mathbf{grad}\,\psi(\boldsymbol{y}) + w_{\hat{\boldsymbol{x}}}\Delta\psi(\boldsymbol{y})$; since $\psi \in \mathcal{C}^2(\bar{D})$, both addends are continuous on $\bar{D}$ and thus they also belong to $L^2(D \setminus \bar{S})$ and thus the integral is bounded.
In the end, we have that $u \mapsto F^*(u)(\hat{\boldsymbol{x}})$ is bounded on $L^2(\Omega)$.

(14.1o) For fixed $\hat{\boldsymbol{x}} \in \mathbb{S}^1$ we consider the output error $|F^*(u)(\hat{\boldsymbol{x}}) - F^*(u_N)(\hat{\boldsymbol{x}})|$, where $u_N \in \mathcal{S}_1^0(\mathcal{M})$ is the finite element solution introduced in Part **II** of the problem. Moreover, we assume (14.1.18).

Establish what will be the asymptotic dependence of this output error on the meshwidth $h_{\mathcal{M}}$, if $\psi \in \mathcal{C}^2(\overline{D} \setminus S)$ and

- both D and S are discs,
- and we deal with a family of triangular meshes whose shape regularity measures ($\rightarrow$ [NPDE, Def. 5.3.36]) are uniformly small.

HINT: You may take for granted that polygonal boundary approximation does not affect the asymptotic convergence for lowest order Lagrangian finite elements, *cf.* [NPDE, Section 5.5.2]. Then rely on [NPDE, Thm. 5.6.7], state the dual problem in strong form based on (14.1.19) and use elliptic regularity theory from [NPDE, Thm. 5.4.2].

Solution: The dual problem in strong form is:

$$g_F \in H^1(D \setminus \bar{S}) :$$
$$\int_{D\setminus\bar{S}} \mathbf{grad}\, v\, \mathbf{grad}\, g_F \,\mathrm{d}x - \int_D k(\boldsymbol{x})^2 v g_F \,\mathrm{d}x + \int_{\partial D} \mathrm{i}k(\boldsymbol{x}) v g_F \,\mathrm{d}S = F^*(v)(\hat{\boldsymbol{x}}) \quad \forall v \in H^1(D \setminus \bar{S})$$

with $F^*(v)(\hat{\boldsymbol{x}})$ as defined in (14.1.19).
Because of [NPDE, Thm. 5.4.2], $g_F \in H^2(D \setminus \bar{S})$ and thus, using [NPDE, Thm. 5.6.7], we expect and order of convergence 2 in the meshwidth.

(14.1p) Implement the function

```
 template <class GradPsiFunc>
complex_t Farfield(DofHandler const& dofh, GridView const& gv,
    Vector const& u, Coordinate const& p, calc_t const& k,
    GradPsiFunc const& GradPsi)
```

to compute the far field in the point p of the unit sphere. We will use the stable formula (14.1.17) for the far field.

The input argument k denotes the wavenumber for $w_{\hat{\boldsymbol{x}}}$ and u is the vector containing the solution u to Helmholtz equation.

Solution: See Listing 14.3.

Listing 14.3: Implementation for `Farfield`

```
#ifndef FARFIELD_HPP_
#define FARFIELD_HPP_

#include <cassert>
#include <cstdlib>
#include <iostream>
#include <cmath>
```

```
#include <vector>
// Dune includes
#include <dune/grid/alugrid.hh>
#include <dune/grid/alugrid/2d/alugrid.hh>
#include <dune/grid/common/gridfactory.hh>
// LFEM includes
#include "LocalFunctionFarfield.hpp"
#include "LoadFunc.hpp"

namespace NPDE15{
const int world_dim = 2;

using calc_t = double;
using complex_t = std::complex<calc_t>;

using Vector = Eigen::VectorXcd;
using GridType = Dune::ALUSimplexGrid<2, 2>;
using GridView = GridType::LeafGridView;
using Coordinate  = Dune::FieldVector<calc_t, world_dim>;
using DofHandler  = NPDE15::LDofHandler<GridView>;

template <class GradPsiFunc>
complex_t Farfield(DofHandler const& dofh, GridView const& gv, Vector const& u,
                   Coordinate const& p, calc_t const& k, GradPsiFunc const& GradPsi){

  // type for a complex coordinate (2-dim complex vector)
  using CCoordinate = Dune::FieldVector<std::complex<calc_t>,world_dim>;

  // quadrature rule:
  Dune::P1LocalFiniteElement<calc_t,calc_t,world_dim> localFE;
  typedef typename Dune::QuadratureRules<calc_t, world_dim> QuadRules;
  const Dune::QuadratureRule<calc_t, world_dim>& QuadRule = QuadRules::rule(localFE.type(), 10);

  complex_t J = 0.;
  complex_t i = {0,1};
  complex_t factor = exp(i*M_PI/4.)/sqrt(8.*M_PI*k);
  auto wave_ = [&i, &k](Coordinate const& x, Coordinate const& p){ return exp(-i*k*(p[0]*x[0]+p[1]*x[1])); };
  for (auto eit = gv.template begin<0>(); eit!=gv.template end<0>(); ++eit){
    auto const& egeom=eit->geometry();
    assert(localFE.type()==eit->type());
```

```
    for (auto qr : QuadRule){
      auto const& local_pos = qr.position();
      // determinant of transformation from reference element
      double jac_det = egeom.integrationElement(local_pos);
      // jacobian inverse transposed for transformation rule
      auto &jacInvTransp=
         egeom.jacobianInverseTransposed(local_pos);

      std::vector<Dune::FieldMatrix<calc_t,1,world_dim>>
         ref_gradients;
      // gradients on reference element evaluated at the quad
         points
      localFE.localBasis().evaluateJacobian(local_pos,
         ref_gradients);

      std::vector<Coordinate> gradients(ref_gradients.size());
      // transform reference gradients to real element gradients:
      for (unsigned j=0; j<gradients.size(); ++j)
     jacInvTransp.mv(ref_gradients[j][0], gradients[j]);

      CCoordinate grad_u; grad_u[0]=grad_u[1]=0.;
      CCoordinate temp;
      for (unsigned j=0; j<gradients.size(); ++j){
     unsigned globalidx=dofh(*eit,j);
     temp = gradients[j];
     temp *= u[globalidx];
     grad_u += temp;
      }
      complex_t uval=0.;
      complex_t temp2 = 0.;
      std::vector<Dune::FieldVector<calc_t,1> >
         shapefct_values;
      localFE.localBasis().evaluateFunction(local_pos,
         shapefct_values);
      for(unsigned j=0; j<shapefct_values.size(); ++j){
     unsigned globalidx=dofh(*eit,j);
     temp2 = shapefct_values[j];
     temp2 *= u[globalidx];
     uval += temp2;
      }

      Coordinate global_pos = egeom.global(local_pos);
      CCoordinate gradPsi_val = GradPsi(global_pos, *eit);

      CCoordinate grad_wave = p;
      complex_t wave = wave_(global_pos,p);
      grad_wave *= (-i*k*wave);
```

```
        // update flux
        J+=
            factor*qr.weight()*jac_det*(grad_wave*gradPsi_val*uval-grad_u*gra
      }

    }
    return J;

}
}

#endif
```

As cut-off function we use

$$\psi(\boldsymbol{y}) = \frac{\|\boldsymbol{y}\|^2 - R_{\text{in}}^2}{R_{\text{out}}^2 - R_{\text{in}}^2}, \tag{14.1.20}$$

where R_{out} is the radius of ∂D and R_{in} the radius of ∂S.

The gradient of $\psi = \psi(\boldsymbol{y})$ is implemented in the class GradPsiFunc, provided in the file GradPsiFunc.hpp of the handout.

(14.1q) Modify the file main.cc implemented in task (14.1g) in order including the far field computation at the current angle (the angles can be initialized at the beginning of the file).

Produce a plot of the absolute value of the far field.

Solution: See Listing 14.4, lines 114-121 (the angle can be substituted by an array of angles), and Figure 14.2.

(14.1r) Finally, we want to study the convergence of the far field mapping. Fixing a point $\hat{x} \in \mathbb{S}^1$, we want to estimate the asymptotic behavior of $|F^*(u)(\hat{\boldsymbol{x}}) - F^*(u_N)(\hat{\boldsymbol{x}})|$, as stated in subproblem (14.1o). Of course, as $F^*(u)(\hat{\boldsymbol{x}})$ we consider (14.1.17), as in the previous subproblem. We have at our disposal the meshes, `Helmholtz_mesh1`, ..., `Helmhotz_mesh5`, ordered from the coarser to the finest one, and obtained by successive refinements. Since we don't have an analytical solution for the far field, we consider the discrete solution on the finest grid as reference solution.
Adapt the file `main.cc` including the convergence study. Test the routine with some points in the unit circle $\mathbb{S}^1$; which order of convergence do you observe?

Solution:

Listing 14.4: Implementation for `main.cc`

```
1 #include <stdexcept>
2 #include <cassert>
3 #include <cstdlib>
4 #include <iostream>
5 #include <cmath>
6 #include <Eigen/Sparse>
```

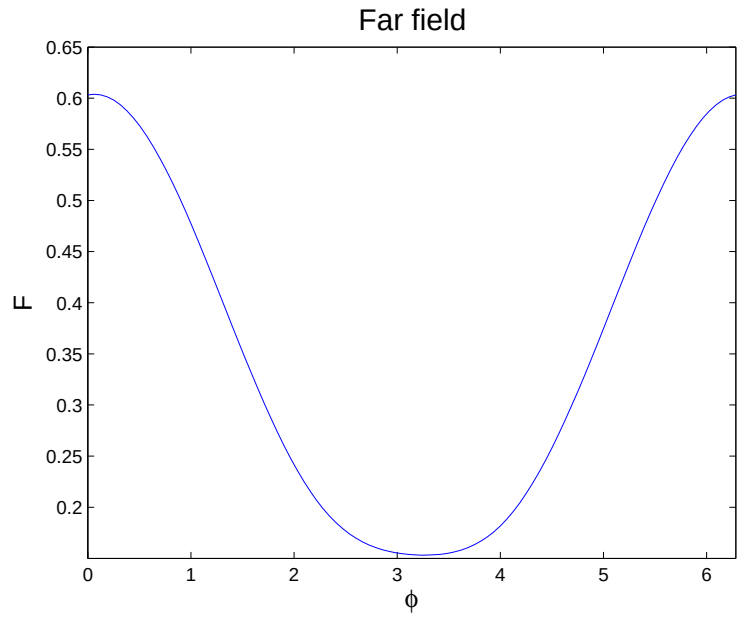

Figure 14.2: Plot for subproblem (14.1q)

```
#include <Eigen/Dense>
#include <Eigen/Cholesky>
#include <vector>
#include <fstream>
// Dune includes
#include "../config.h"
#include <dune/common/exceptions.hh>
#include <dune/grid/alugrid.hh>
#include <dune/grid/alugrid/2d/alugrid.hh>
#include <dune/grid/common/gridfactory.hh>
#include <dune/grid/io/file/vtk/subsamplingvtkwriter.hh>
#include <dune/grid/io/file/gmshreader.hh>
// NPDE15 includes
#include "npde15/Pardiso.hpp"
#include "npde15/global/VectorAssembler.hpp"
// LFEM includes
#include "MatrixAssembler.hpp"
#include "../LFEM_Laplace_Neumann/AnalyticalLocalLaplace.hpp"
#include "../LFEM_Laplace_Dirichlet/LDofHandler.hpp"
#include "LBoundaryDofs.hpp"
#include "LocalMassFarfield.hpp"
#include "LocalFunctionFarfield.hpp"
#include "KappaFunc.hpp"
#include "LoadFunc.hpp"
#include "GradPsiFunc.hpp"
#include "Farfield.hpp"

```

```
const int world_dim = 2;

using calc_t = double;
using complex_t = std::complex<calc_t>;

using Matrix = Eigen::SparseMatrix<std::complex<calc_t>,
    Eigen::RowMajor>;
using Vector = Eigen::VectorXcd;
using IndexVector = std::vector<bool>;
using GridType = Dune::ALUSimplexGrid<2, 2>;
using GridView = GridType::LeafGridView;
using Coordinate  = Dune::FieldVector<calc_t, world_dim>;
using DofHandler  = NPDE15::LDofHandler<GridView>;

int main(int argc, char *argv[]){
  try{

    calc_t freq   = 10.e7;
    calc_t omega  = 2*M_PI*freq;
    calc_t c0     = 3.e8;
    calc_t lambda = c0/freq;
    calc_t k      = omega/c0;
    calc_t ks_sq=2.*k*k,kd_sq=1.*k*k;
    int nlevels =5;
    double FFvalues[5], error[4], Ndofs[5];
    Coordinate p;
    double Angle = 3.*M_PI/2.;
    p[0]=cos(Angle);
    p[1]=sin(Angle);

    for(int level=1; level<=nlevels; ++level){
    // load the grid from file
    std::string FileName = "Helmholtz_mesh" +
        std::to_string(level) + ".msh";

    // Declare and create mesh using the Gmsh file
    std::vector<int> ElemFlag; // will hold Element flags
    std::vector<int> BndFlag; // will hold Boundary flags
    Dune::GridFactory<GridType> gridFactory;
    Dune::GmshReader<GridType>::read(gridFactory,
        FileName.c_str(), BndFlag, ElemFlag, false, true);
    std::unique_ptr<GridType>
        workingGrid(gridFactory.createGrid());
    workingGrid->loadBalance();
    // Get the Gridview
    GridView gv = workingGrid->leafGridView();

```

```
    // Initialize dof-handler
    DofHandler dofh(gv);

    unsigned N = dofh.size();
    std::cout << "Solving for N =" << N << " unknowns.\n";
    Ndofs[level-1] = N;

    LoadFunc<GridView> f(gv,ElemFlag,ks_sq,kd_sq);

    // assemble rhs
    Vector Phi(N); Phi.setZero();
    NPDE15::VectorAssembler<DofHandler> vecAssembler(dofh);
    vecAssembler(Phi, NPDE15::LocalFunctionFarfield(f));

    // assemble the system matrix
    std::vector<Eigen::Triplet<std::complex<calc_t>>>
        triplets;
    NPDE15::MatrixAssembler<DofHandler> matAssembler(dofh);

    KappaFunc<GridView> kappa(gv,kd_sq,ks_sq,ElemFlag);
    matAssembler(triplets, NPDE15::LocalMassFarfield(kappa));
    matAssembler(triplets, NPDE15::AnalyticalLocalLaplace());
    /* Add boundary contribution */
    NPDE15::LBoundaryDofs<DofHandler> get_bnd_dofs(dofh);
    IndexVector robin_dofs;
    get_bnd_dofs(robin_dofs);
    KappaBdFunc kappabd(kd_sq);
    matAssembler(triplets,NPDE15::LocalMassFarfield(kappabd),robin_dofs);

    Matrix A(N, N);
    A.setFromTriplets(triplets.begin(), triplets.end());
    A.makeCompressed();

    // solution vector u
    Vector u(N); u.setZero();
    // solve the system
    u = Phi/A; // short-hand, see Pardiso.hpp for more information

    // with this data, the mesh needs to span an annulus:
    calc_t R_in   = 3./5.;
    calc_t R_out  = 3.;

    GradPsiFunc<GridView> gradPsi(gv,ElemFlag,R_in,R_out);

    // farfield calculation:
    FFvalues[level-1]=abs(Farfield(dofh,gv,u,p,k,gradPsi));

```

```
    // plot absolute value of the solution:
    if(level==nlevels){
    std::vector<double> solution(N),rhs(N);
    for (unsigned i=0;i<N;++i)
     solution[i]=abs(u[i]);

    std::cout << "\n\nWriting solution to vtk file... ";
    Dune::VTKWriter<GridView> vtkwriter(gv);
    std::stringstream name;
    name << "solution";
    vtkwriter.addVertexData(solution, "u(x)");
    vtkwriter.write(name.str().c_str());
    std::cout << "Done.\n";
    }
}

  for(unsigned level=1; level< nlevels; ++level)
     error[level-1] =
        std::abs(FFvalues[level-1]-FFvalues[nlevels-1]);

    std::ofstream outnd("Ndofs.dat", std::ios::out |
       std::ios::binary );
    std::ofstream outerr("FFerror.dat", std::ios::out |
       std::ios::binary );
  for(unsigned level=1;level<nlevels; ++level){
    outnd << Ndofs[level-1]<<std::endl;
    outerr << error[level-1]<<std::endl;
  }
    outnd.close( );
    outerr.close( );

  }

  // catch exceptions
  catch (Dune::Exception &e){
    std::cerr << "Dune reported error: " << e << std::endl;
  }
  catch (...)
    std::cerr << "Unknown exception thrown!" << std::endl;
  }
  return 0;
}
```

The order of convergence with respect to the number of degrees of freedom that we observe are:

- 1.0591 for the point $(1, 0)$, see Figure 14.3 for the convergence plot;
- 1.0759 for the point $(0, 1)$;

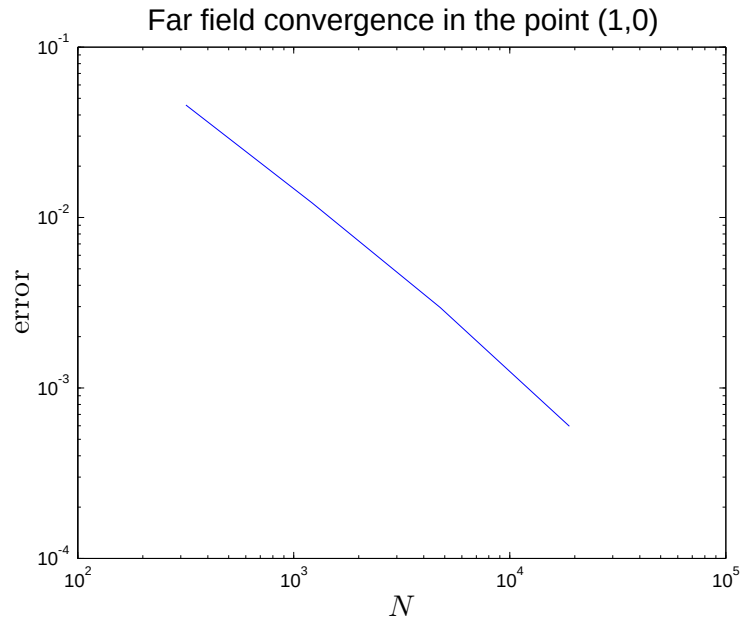

Figure 14.3: Plot for subproblem (14.1r)

- 1.0531 for the point $(-1, 0)$;
- 1.0727 for the point $(0, -1)$;

On the basis of these observations, we can say that the order of convergence is around 1.06 with respect to the number of degrees of freedom, i.e. around nearly 2.1 with respect to the meshwidth, as we expected from the estimate in (14.1o).

Problem 14.2 Electrostatic Force (Part I)

A straight cylindrical wire is enclosed in a straight cylindrical conducting pipe as depicted in 14.4. There is a voltage drop between both. If the axial extension of both wire and pipe is very long, translational symmetry along the axis can be assumed, which allows two-dimensional modelling. As explained in [NPDE, Section 2.2.2], the electrostatic potential u in the homogeneous space Ω between the surface Γ_1 of the wire and the inner wall Γ_0 of the pipe, is given as the solution of the 2nd-order elliptic boundary value problem

$$-\Delta u = 0 \qquad \text{in}\,\Omega, \tag{14.2.1}$$

with Dirichlet boundary conditions

$$u = 0 \quad \text{on}\,\Gamma_0, \qquad u = 1 \quad \text{on}\,\Gamma_1. \tag{14.2.2}$$

Of course, a suitable scaling is assumed that yielded the non-dimensional equation (14.2.1). In this problem the use of a finite element method to compute the total attracting *force* between wire and pipe approximately is explored.

Caution: In contrast to the cases discussed in class, the output functionals studied in this problem are *non-linear*!

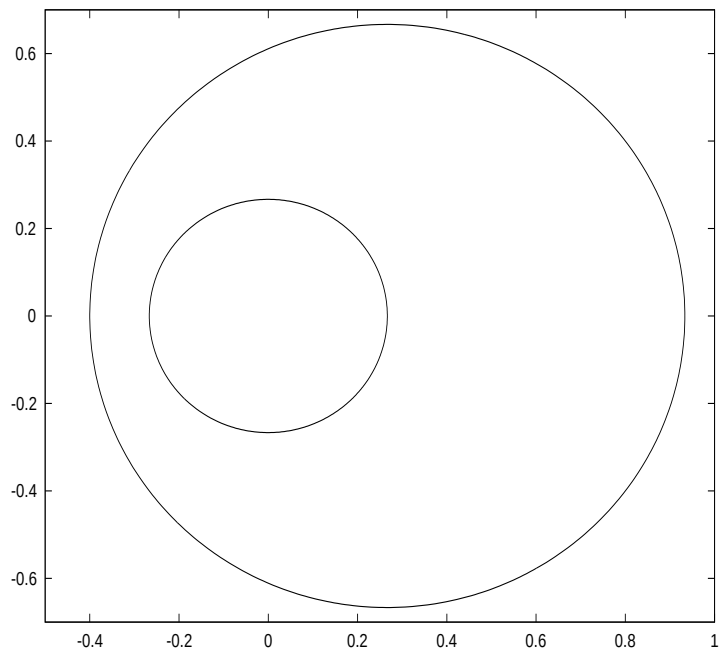

Figure 14.4: The domain used in Problem 14.2

In the above model the electrostatic force on the wire can be computed from the potential u according to

$$\boldsymbol{F}(u) = \frac{1}{2}\int_{\Gamma_1} (\operatorname{grad} u(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}))\operatorname{grad} u(\boldsymbol{x})\,\mathrm{d}S \tag{14.2.3}$$

To understand this formula recall $\mathbf{E} = -\,\mathbf{grad}\,u$, see [NPDE, Eq. (2.2.11)], and that $\mathbf{E}\cdot\boldsymbol{n}$ gives the surface charge density on a conductor. Thus the integrand in (14.2.3) can be read as the force density effected by the electric field "pulling at the surface charges".

In order to obtain an analytical solution for u the following special geometric situation will be considered. The pipe wall Γ_0 is a circle centered in $(4/15, 0)$ with radius $2/3$, and the wire boundary Γ_1 is a circle centered in the origin, with radius $4/15$.

(14.2a) Show that the solution to (14.2.1)-(14.2.2) is

$$u(\boldsymbol{x}) = \frac{1}{\log 2}(\log\|\boldsymbol{x}-\boldsymbol{a}\| - \log\|\boldsymbol{x}-\boldsymbol{b}\|) - 1 \tag{14.2.4}$$

where $\boldsymbol{a} = (-16/15, 0)$ and $\boldsymbol{b} = (-1/15, 0)$ are the positions of the point charges.

HINT: You may use a MATLAB or C++ code to verify that the boundary conditions are satisfied. Alternatively, you may appeal to the Apollonius circle theorem that you may have heard about in secondary school.

Solution: A function on the form $\log\|\boldsymbol{x}-\boldsymbol{a}\|$ can be written as $\log r$ for a suitable choice of polar coordinates (with origin in $\boldsymbol{a}$). The radial part of the Laplacian in polar coordinates is

$$\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u}{\partial r}\right),$$

and since $\frac{\partial \log r}{\partial r} = \frac{1}{r}$, we see that this evaluates to zero.

See Listing 14.5 for the boundary condition test.

Listing 14.5: Implementation for (14.2a).

```
function bdtest

    points = 100;
    angle = linspace(0,2*pi,100);
    angle = angle(1:end-1);
    points = points - 1;

    inner = [cos(angle); sin(angle)] * 4/15;
    outer = [4/15+cos(angle)*2/3; sin(angle)*2/3];

    nrm = @(x) sqrt(sum(x.^2,1));
    func = @(x) (log(nrm(x+repmat([16/15;0],1,size(x,2)))) -
       log(nrm(x+repmat([1/15;0],1,size(x,2)))))/log(2) - 1;

    norm(func(inner)-1) * sqrt(points)
    norm(func(outer)) * sqrt(points)

end
```

(14.2b) Compute the force between wire and pipe for the exact solution found in subproblem (14.2a)

HINT: Exact computation is tedious. Therefore, you may use "overkill" numerical quadrature (e.g. trapezoidal rule with 10^6 points) within a MATLAB or C++ code to obtain a good approximation for the exact value of the force functional. The gradient of u is

$$\operatorname{grad} u(\boldsymbol{x}) = \frac{1}{\log 2}\left(\frac{\boldsymbol{x}-\boldsymbol{a}}{\|\boldsymbol{x}-\boldsymbol{a}\|^2} - \frac{\boldsymbol{x}-\boldsymbol{b}}{\|\boldsymbol{x}-\boldsymbol{b}\|^2}\right).$$

HINT: You should obtain the force value $\boldsymbol{F} = (13.0776, 0)$.

Solution: See Listing 14.6. The force acts, unsurprisingly, only in the x-direction, and has a value there of 13.0776.

Listing 14.6: Implementation for (14.2b).

```
function force = exactforce

    points = 1e6;
    angle = linspace(0,2*pi,points);
    angle = angle(1:end-1);
    points = points - 1;

    inner = [cos(angle); sin(angle)] * 4/15;

    nrm = @(x) sum(x.^2,1);
```

```
    ext = @(x) repmat(x,2,1);
    xa = @(x) (x+repmat([16/15;0],1,size(x,2)));
    xb = @(x) (x+repmat([1/15;0],1,size(x,2)));
    grad = @(x) (xa(x)./ext(nrm(xa(x))) -
       xb(x)./ext(nrm(xb(x))))/log(2);
    n = @(x) -x./ext(sqrt(nrm(x)));

    force =
       ext(sum(grad(inner).*n(inner),1)).*grad(inner)*((2*pi*4/15)/2);
    force = sum(force,2)/points;

end
```

(14.2c) Complete `main.cc` in order to solve (14.2.1)-(14.2.2) using a linear Lagrangian finite element method.

HINT: Use your already implemented classes `DofHandler`, `MatrixAssembler`, `VectorAssembler`, `BoundaryDofs`, and local assemblers, developed in the previous assignments (also available in the corresponding solution folders).

HINT: Use `BoundaryDofs` to find the indices of the boundary edges. Then, loop through each boundary edge and check the norm of the vertices of that edge. If the norms are not much more than $4/15$, the edge is part of the inner boundary.

Solution: See Listing 14.7 for the code. This also include the code required for the rest of the problem. The relevant part for this subproblem is from line 80 to 152.

Listing 14.7: Implementation of `main`

```
31 const int world_dim = 2;
32 using calc_t = double;
33
34 using Matrix = Eigen::SparseMatrix<calc_t, Eigen::RowMajor>;
35 using Vector = Eigen::VectorXd;
36 using IndexVector = std::vector<bool>;
37 using GridType = Dune::ALUSimplexGrid<2, 2>;
38 using GridView = GridType::LeafGridView;
39 using Coordinate  = Dune::FieldVector<calc_t, world_dim>;
40 using DofHandler  = NPDE15::LDofHandler<GridView>;
41
42 template <class Function>
43 double L2Error(DofHandler dofh, Vector const& FN, Function const& Fex) {
44   GridView const& gv = dofh.gv;
45   Dune::PkLocalFiniteElement<calc_t,calc_t,world_dim,1> localFE;
46   typedef typename Dune::QuadratureRule<calc_t, world_dim> QuadRule_t;
47   typedef typename Dune::QuadratureRules<calc_t, world_dim> QuadRules;
48   const QuadRule_t & quadRule = QuadRules::rule(localFE.type(), 10);
49
50   calc_t L2e=0.;
51   for (auto it=gv.template begin<0>();it!=gv.template end<0>();++it){
52     auto const& e=*it;
53     auto egeom = e.geometry();
54     assert(localFE.type()==e.type());
55     for (auto qr : quadRule){
```

```
      auto const& local_pos=qr.position();
      auto const& global_pos=egeom.global(local_pos);
      const calc_t F_val = Fex(global_pos); // Point value of exact solution
      // determinant of transformation from reference element
      double jac_det = egeom.integrationElement(local_pos);

      // Fetch values of all local shape functions in a point on reference element
      std::vector<Dune::FieldVector<calc_t,1> > lsf_vals(localFE.localBasis().size());
      localFE.localBasis().evaluateFunction(local_pos,lsf_vals);
      // Add up contributions of local shape functions
      calc_t FN_val = 0.0;
      for (int i=0; i<localFE.localBasis().size(); ++i) FN_val += FN[dofh(e, i)]*lsf_vals[i];

      // Another summand in the quadrature formula
      const calc_t err_pt = F_val - FN_val; // Point error
      // and add weighted difference to the l2 error
      L2e += jac_det * (qr.weight()) * err_pt * err_pt;
    }
  }
  return sqrt(L2e);
}

int main(int argc, char *argv[]){
  try{
    Vector L2e(3); L2e.setZero();
    Vector Ndofs(3); Ndofs.setZero();
    for (int level=1; level<=3; ++level){
      // load the grid from file
      std::string fileName = "Annulus_" + std::to_string(level) +".msh";

      // Declare and create mesh using the Gmsh file
      Dune::GridFactory<GridType> gridFactory;
      Dune::GmshReader<GridType>::read(gridFactory, fileName.c_str(), false, true);{\small\lstinline[style=cpp]{
      std::unique_ptr<GridType> workingGrid(gridFactory.createGrid());
      workingGrid->loadBalance();

      // Get the Gridview
      GridView gv = workingGrid->leafGridView();

      // Initialize dof-handler
      DofHandler dofh(gv);

      unsigned N = dofh.size();

      // Get boundary nodes
      IndexVector dirichlet_dofs(N);
      NPDE15::LBoundaryNodes<DofHandler> get_bnd_dofs(dofh);
      get_bnd_dofs(dirichlet_dofs);
      dofh.set_inactive(dirichlet_dofs);

      // Dirichlet data
      auto g = [](Coordinate const& x){
```

```
      double tol = 1/15.;
      double norm = sqrt(x[0]*x[0] +
         x[1]*x[1]);{\small\lstinline[style=cpp]{
      if(norm < 4/15. +tol) return 1.0;
      return 0.0;
       };
       Vector G(N); G.setZero();
       // loop over cells
       for (auto it=gv.template begin<0>();it!=gv.template end<0>();++it){
      auto const& e=*it;
      auto egeom = e.geometry();
      for (unsigned i=0;i<3;++i){
        unsigned loctoglob = dofh(e, i);
        if (!dofh.active(loctoglob))
          G[loctoglob] = g(egeom.corner(i));
      }
       }

       // assemble rhs and set dirichlet dofs to dirichlet data
       Vector Phi(N); Phi.setZero();
       NPDE15::VectorAssembler<DofHandler> vecAssembler(dofh);
       vecAssembler.set_inactive(Phi, G);

       // assemble the system matrix
       std::vector<Eigen::Triplet<calc_t>> triplets;
       NPDE15::MatrixAssembler<DofHandler> matAssembler(dofh);
       matAssembler(triplets, NPDE15::AnalyticalLocalLaplace());
       matAssembler.set_inactive(triplets);

       Matrix A(N, N);{\small\lstinline[style=cpp]{
       A.setFromTriplets(triplets.begin(), triplets.end());
       A.makeCompressed();

       // solution vector U
       Vector U(N); U.setZero();

       // solve the system
       U = Phi/A; // short-hand, see Pardiso.hpp for more information

       auto u_ex = [](Coordinate const& x){
      return (log(sqrt(pow(x[0]+16./15,2) +
         x[1]*x[1]))-log(sqrt(pow(x[0]+1./15,2) + x[1]*x[1])))/log(2.0) -
         1.0;
       };

       Vector U_ex(N); U_ex.setZero();
       NPDE15::InterpFunction<DofHandler> interpolator(dofh);
       interpolator(U_ex, u_ex);

       L2e[level-1] = L2Error(dofh, U,u_ex);
       Ndofs[level-1] = N;

     }
     std::ofstream outnd("Ndofs.dat", std::ios::out | std::ios::binary );
     outnd << Ndofs;
     outnd.close( );
```

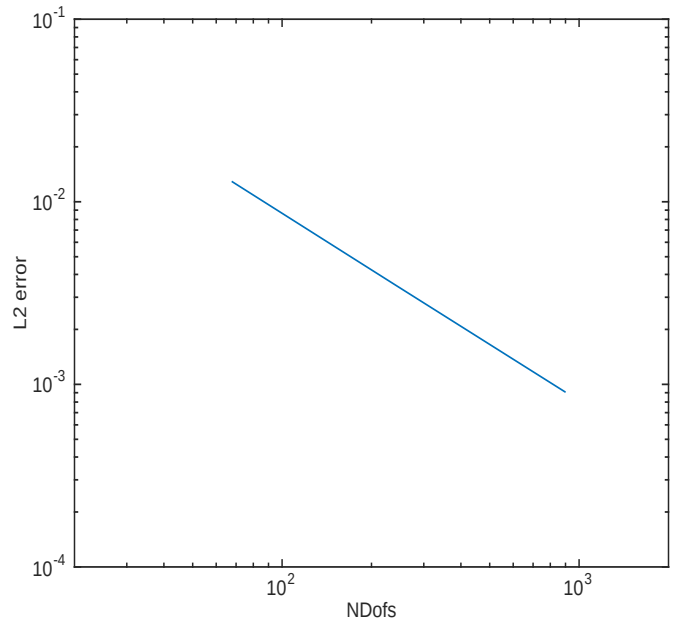

Figure 14.5: Error plot for subproblem (14.2d).

```
    std::ofstream outL2("L2Error.dat", std::ios::out | std::ios::binary );
    outL2 << L2e;
    outL2.close( );
    }
```

(14.2d) In `main.cc` add a function

```
template <class Function>
double L2Error(DofHandler dofh, Vector const& FN, Function const&
    Fex)
```

which calculates $\|F_{ex} - F_N\|_{L^2(\Omega)}$ when given the DofHandler `dofh`, the coefficient vector `FN` associated to the finite element function F_N, and `Fex` given in procedural form.

Use it to compute and plot approximate L^2-errors $\|u - u_N\|_{L^2(\Omega)}$ for the given sequence of meshes in terms of the meshwidth or the number of degrees of freedom. Which type of convergence do you (and should you in light of [NPDE, Section 5.6.3]) observe?

HINT: Types of convergence are defined [NPDE, Section 1.6.2]. The sequence of meshes is given by `Annulus_i.msh`, `i=1..3` available in the repository folder.

Solution: See Listing 14.7 for the computation of the errors. The plot is found in Figure 14.5. The convergence is algebraic, and the reported rate is close to 1, in terms of number of degrees of freedom, which is what we expect.

Problem 14.3 Electrostatic Force (Part II)

In [NPDE, Section 5.6.1] we learned that we can expect enhanced rates of algebraic h-convergence for continuous linear output functionals when evaluating them for finite element Galerkin solutions of linear variational problems. We also saw in [NPDE, Section 5.6.2] for the example of

boundary flux functionals that switching to an equivalent continuous form of the functional can bring about dramatic gains in accuracy.

In this problem we apply this policy to the *non-linear* electrostatic force functional from Problem 14.2. We are going to replace the original force functional with an equivalent one that enjoys better continuity properties. This will be explored in numerical experiments.

(14.3a) Show that the functional $u \mapsto \boldsymbol{F}(u)$ from (14.2.3) is *not* linear.

Solution: It is enough to observe that $\boldsymbol{F}(2u) = 4\boldsymbol{F}(u)$.

(14.3b) Complete the function

```
Dune::FieldVector<double,2> Force(DofHandler const& dofh, GridView
    const& gv, Vector const& U)
```

(in the handout file `Force.hpp`) that returns the force functional $\boldsymbol{F}(u)$ from (14.2.3) for a finite element solution $u_N \in \mathcal{S}_1^0(\mathcal{M})$, where information on the triangular mesh data structure and boundary nodes are contained in `dofh` and `gv`. The vector `U` passes the nodal basis expansion coefficients of a finite element function.

HINT: This subproblem requires you to compute normals of edges on the (inner) boundary. To get them, you can use the method `unitOuterNormal` contained in the `Dune::Intersection` class.

HINT: Formula (14.2.3) requires integration only on part of the boundary (Γ_1). To select the correct boundary edges, proceed as you did in (14.2c) to set the boundary conditions.

The integrand can be evaluated exactly, so there is no need for quadrature.

Solution: To evaluate the formula

$$\boldsymbol{F}(u_N) = \frac{1}{2}\int_{\Gamma_1} (\operatorname{grad} u_N(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}))\operatorname{grad} u_N(\boldsymbol{x})\,\mathrm{d}S, \tag{14.3.1}$$

we note that gradients of piecewise linear functions are constant vectors on each part $\partial K \cap \Gamma_1$ of a mesh cell K (in the code we evaluate $\operatorname{grad} u_N(\boldsymbol{x})$ at the arbitrary point $\boldsymbol{x} = \mathbf{0}$). For a straight triangle also n is constant on $\partial K \cap \Gamma_1$ and the whole integrand of (14.3.1) is independet of $\boldsymbol{x}$. This is implemented in Listing 14.8. Note that we do transform the gradient from the reference triangle $\widehat{K}$ to the real triangle K.

Listing 14.8: Implementation for `force`.

```
function F = force(Mesh, u, bdflag)

    loc = find(Mesh.BdFlags == bdflag);

    F = 0;

    for i = loc'

        vidx = Mesh.Elements(max(Mesh.Edge2Elem(i,:)),:);
        coords_elem = Mesh.Coordinates(vidx,:);
        coords_edge = Mesh.Coordinates(Mesh.Edges(i,:),:);
```

```
        bK = coords_elem(1,:);
        BK = [coords_elem(2,:)-bK; coords_elem(3,:)-bK];
        inv_BK = inv(BK);

        grad = grad_shap_LFE([0 0]);
        grad = (u(vidx(1))*grad(:,1:2) +
           u(vidx(2))*grad(:,3:4) + u(vidx(3))*grad(:,5:6)) *
           transpose(inv_BK);

        n = coords_edge(2,:) - coords_edge(1,:);
        length = norm(n);
        n = [n(2), -n(1)];
        n = n/length;
        midpoint = (coords_edge(2,:) + coords_edge(1,:))/2;
        if dot(midpoint, n) > 0
            n = -n;
        end

        value = dot(grad, n) * grad / 2;
        F = F + value * length;

    end

end
```

(14.3c) Show that there are functions $u \in H^1(\Omega)$ for which $\boldsymbol{F}(u)$ from (14.2.3) is not defined (i.e. "$\boldsymbol{F}(u) = \infty$").

HINT: Examine [NPDE, § 5.6.13]. Try $u(\boldsymbol{x}) = \left(\|\boldsymbol{x}\| - \frac{4}{15}\right)^{\alpha}$ for some suitably chosen α.

Solution: The gradient of such a function is

$$\operatorname{grad} u(r) = \alpha\left(r - \frac{4}{15}\right)^{\alpha-1} \boldsymbol{r},$$

where r is the distance to the origin, and $\boldsymbol{r}$ is the radial element vector. The integral is over the inner boundary integrates, we have $r = 4/15$ there, and so the gradient will be infinite for $\alpha < 1$. The function u is in $H^1(\Omega)$ if $\|u\|_{H^1(\Omega)} < \infty$, in particular

$$|u|^2_{H^1(\Omega)} = \int_0^{2\pi} \int_{4/15}^{r^{\text{out}}(\phi)} \operatorname{grad} u(r) \cdot \operatorname{grad} u(r) r \, \mathrm{d}r \, \mathrm{d}\phi = \alpha^2 \int_0^{2\pi} \int_{4/15}^{r^{\text{out}}(\phi)} \left(r - \frac{4}{15}\right)^{2\alpha-1} \mathrm{d}r \, \mathrm{d}\phi$$

which fulfills $|u|^2_{H^1(\Omega)} < \infty$ for $2\alpha - 1 \geq -1$, i.e., $\alpha \geq 0$. Thus we have a counter-example for $0 \leq \alpha < 1$.

(14.3d) Prepare for the three given meshes $\ell = 1, 2, 3$ a plot for the convergence in term of $\boldsymbol{F}(u_\ell)$. What (kind and rate) of convergence $\boldsymbol{F}(u_N) \to \boldsymbol{F}(u)$ in terms of the meshwidth do you observe?

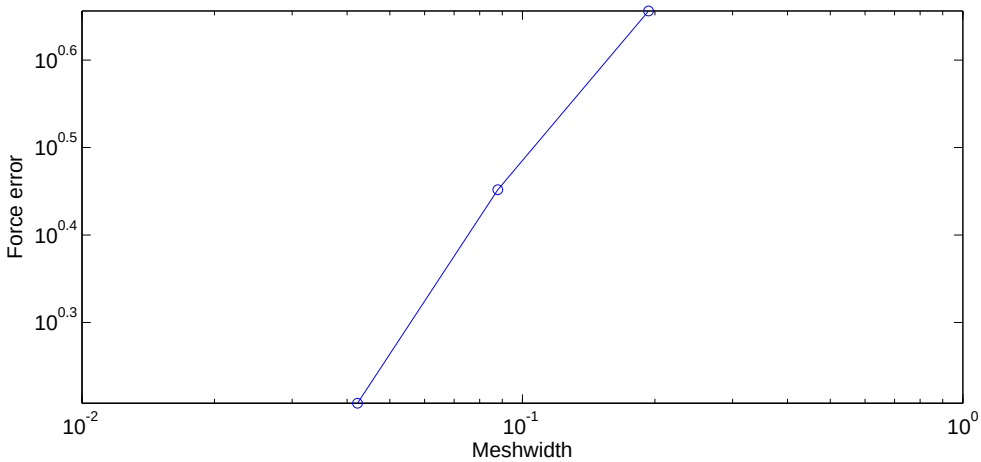

Figure 14.6: Convergence of the direct computation for the Force.

HINT: You can solve this task updating the file `main.cc` from Problem 14.2 with the computation of the force.

Solution:

The implementation in Listing 14.9 is based on the previous implementation for (14.2d). In Figure 14.6 we observe algebraic convergence with a rather poor rate:

```
>> main2
convergence rate in terms of meshwidth: 0.677729
```

which corresponds to $1/3$ in terms of number degrees of freedom.

Listing 14.9: Implementation for (14.3d).

```
clear all;

n = zeros(1,3);
h = zeros(1,3);
f = zeros(3,2);
fs = zeros(3,2);

for l = 1:3

    Mesh = load_Mesh(['coords_' num2str(l) '.dat'], ['elems_'
       num2str(l) '.dat']);
    Mesh.ElemFlag = ones(size(Mesh.Elements,1),1);
    Mesh = add_Edges(Mesh);
    Mesh = bdflags(Mesh);
    Mesh = add_Edge2Elem(Mesh);

    A = assemMat_LFE(Mesh, @STIMA_Lapl_LFE);
    L = zeros(size(Mesh.Coordinates,1),1);

```

```
    dir_inner = @(x,varargin) ones(size(x,1),1);
    dir_outer = @(x,varargin) zeros(size(x,1),1);
    [U1, Free1] = assemDir_LFE(Mesh, -1, dir_inner);
    [U2, Free2] = assemDir_LFE(Mesh, -2, dir_outer);
    U = U1 + U2;
    Free = intersect(Free1, Free2);

    L = L - A*U;
    U(Free) = A(Free,Free)\L(Free);

    nrm = @(x) sqrt(sum(x.^2,1));
    func = @(x,varargin)
       ((log(nrm(x'+repmat([16/15;0],1,size(x',2)))) -
       log(nrm(x'+repmat([1/15;0],1,size(x',2))))))/log(2) -
       1)';

    n(l) = size(Mesh.Coordinates, 1);
    h(l) = get_MeshWidth(Mesh);
    f(l,:) = force(Mesh, U, -1);
    fs(l,:) = forcestable(Mesh, U);
end

ef = exactforce;
f_err = sqrt(sum((f-repmat(ef',3,1)).^2,2))';
p = polyfit(log(h), log(f_err), 1);
fprintf('convergence rate in terms of meshwidth: %f\n', p(1))
figure(1);
loglog(h, f_err, 'o-');
xlabel('Meshwidth');
ylabel('Force error');

conf.myFontsize = 13;
conf.size = [700, 300];
conf.its = 1:1;
fig_prepare(conf); legend off
saveas(gcf, 'main2.eps', 'epsc')
```

(14.3e) According to subproblem (14.3c) it might be possible to encounter infinitely large electrostatic forces attracting the wire towards the pipe. Why will this never be observed?

Solution: These functions are not solutions to (14.2.1). Since we deal with a second-order elliptic boundary value problem posed on a smooth domain and with smooth coefficients, solutions will belong to $H^2(\Omega)$, which is sufficient to ensure that the gradient is in $(H^1(\Omega))^3$. Hence, by the trace theorem, the force functional will attain a finite value.

(14.3f) [NPDE, Section 5.6.2] strikingly demonstrated the benefit of replacing an unbounded linear output functional with an equivalent bounded one. By subproblem (14.3c) the output func-

tional $\boldsymbol{F}$ from (14.2.3) may be haunted by the same problems as [NPDE, Eq. (5.6.11)], though it is non-linear. Thus, it may pay off to reformulate (14.2.3) in a form $\widetilde{\boldsymbol{F}}$ that is continuous on $H^1(\Omega)$, and which agrees with $\boldsymbol{F}$ for all solutions of (14.2.1).

Show that, for all $\boldsymbol{x} \in \Gamma_1$, we have

$$\frac{1}{2}(\operatorname{grad} u(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})) \operatorname{grad} u(\boldsymbol{x}) = \boldsymbol{T}(u)(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}),$$

where $\boldsymbol{T}$ is the so-called Maxwell stress tensor

$$\boldsymbol{T}(u)(\boldsymbol{x}) = \operatorname{grad} u(\boldsymbol{x}) \cdot \operatorname{grad} u(\boldsymbol{x})^\top - \frac{1}{2}\|\operatorname{grad} u(\boldsymbol{x})\|^2 \cdot \boldsymbol{I} \in \mathbb{R}^{2,2} \ ,$$

where $\boldsymbol{I}$ is the 2×2 identity matrix.

HINT: If u is constant on Γ_1, $\operatorname{grad} u$ will be parallel to $\boldsymbol{n}$ there.

Solution: Assume that $\operatorname{grad} u = c\boldsymbol{n}$ for some c. Then we have $\operatorname{grad} u \cdot \boldsymbol{n} = c$, and

$$\frac{1}{2}(\operatorname{grad} u \cdot \boldsymbol{n}) \operatorname{grad} u = \frac{1}{2}c^2\boldsymbol{n}.$$

For the other side, we have $\|\operatorname{grad} u\|^2 = c^2$, so we have

$$\boldsymbol{T}(u) \cdot \boldsymbol{n} = c^2\boldsymbol{n} \cdot \boldsymbol{n}^\top \cdot \boldsymbol{n} - \frac{1}{2}c^2\boldsymbol{n} = \frac{1}{2}c^2\boldsymbol{n},$$

because $\boldsymbol{n}^\top \cdot \boldsymbol{n} = \|\boldsymbol{n}\|^2 = 1$.

(14.3g) Show that if u solves (14.2.1), then

$$\operatorname{div} \boldsymbol{T}(u)(\boldsymbol{x}) = \boldsymbol{0}$$

for all $\boldsymbol{x} \in \Omega$. Here, div is row-wise divergence, i.e. it takes the divergence of each *row* $\boldsymbol{T}_i$ of the matrix $\boldsymbol{T}$ (regarded as a vector) and returns a vector:

$$\operatorname{div}\begin{pmatrix} t_{11}(\boldsymbol{x}) & t_{12}(\boldsymbol{x}) \\ t_{21}(\boldsymbol{x}) & t_{22}(\boldsymbol{x}) \end{pmatrix} = \begin{pmatrix} \dfrac{\partial t_{11}}{\partial x_1}(\boldsymbol{x}) + \dfrac{\partial t_{12}}{\partial x_2}(\boldsymbol{x}) \\ \dfrac{\partial t_{21}}{\partial x_1}(\boldsymbol{x}) + \dfrac{\partial t_{22}}{\partial x_2}(\boldsymbol{x}) \end{pmatrix}. \tag{14.3.2}$$

Solution: Row i of $\boldsymbol{T}$ is

$$\boldsymbol{T}_i = \frac{\partial u}{\partial x_i} \operatorname{grad} u^\top - \frac{1}{2}\left[\|\operatorname{grad} u\|^2\right]_i,$$

where $[s]_i$ means a vector with zeros, except for the value s in position i. Thus,

$$\operatorname{div} \boldsymbol{T}_i = \frac{\partial u}{\partial x_i} \operatorname{div} \operatorname{grad} u^\top + \frac{\partial^2 x}{\partial x_1 \partial x_i}\frac{\partial u}{\partial x_1} + \frac{\partial^2 x}{\partial x_2 \partial x_i}\frac{\partial u}{\partial x_2} - \frac{1}{2}\frac{\partial u}{\partial x_i}\|\operatorname{grad} u\|^2.$$

The first term will vanish due to (14.2.1). The rest will cancel out after computing the last term.

(14.3h) Show that if u solves (14.2.1), then

$$\boldsymbol{F}(u) = \widetilde{\boldsymbol{F}}(u) := \int_\Omega \boldsymbol{T}(u)(\boldsymbol{x}) \operatorname{grad} \Psi(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x}, \tag{14.3.3}$$

where $\Psi \in \mathcal{C}^\infty(\overline{\Omega})$ satisfies $\Psi(\boldsymbol{x}) = 1$ on Γ_1 and $\Psi(\boldsymbol{x}) = 0$ on Γ_0.

HINT: First, show that $\boldsymbol{F}(u) = \int_{\partial\Omega} \Psi \boldsymbol{T} \cdot \boldsymbol{n} \,\mathrm{d}S$, and then use Gauss' theorem (the divergence theorem, [NPDE, Thm. 2.5.7]). Be inspired by the derivation of [NPDE, Eq. (5.6.15)].

Solution:

$$\begin{aligned}
\boldsymbol{F}(u) &= \frac{1}{2} \int_{\Gamma_1} (\operatorname{grad} u(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})) \operatorname{grad} u(\boldsymbol{x}) \,\mathrm{d}S \\
&= \int_{\Gamma_1} \boldsymbol{T}(u)(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) \,\mathrm{d}S = \int_{\partial\Omega} (\Psi(\boldsymbol{x}) \boldsymbol{T}(u)(\boldsymbol{x})) \cdot \boldsymbol{n}(\boldsymbol{x}) \,\mathrm{d}S \\
&= \int_\Omega \operatorname{div}(\Psi \boldsymbol{T}) \,\mathrm{d}\boldsymbol{x} \\
&= \int_\Omega (\boldsymbol{T}(u)(\boldsymbol{x}) \cdot \operatorname{grad} \Psi(\boldsymbol{x}) + \Psi(\boldsymbol{x}) \operatorname{div} \boldsymbol{T}(u)(\boldsymbol{x})) \,\mathrm{d}\boldsymbol{x} \\
&= \int_\Omega \boldsymbol{T}(u)(\boldsymbol{x}) \operatorname{grad} \Psi(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = \widetilde{\boldsymbol{F}}(u).
\end{aligned}$$

(14.3i) Show that $\widetilde{\boldsymbol{F}}(v)$, defined in (14.3.3), is is bounded for all $v \in H^1(\Omega)$, that is

$$|\widetilde{\boldsymbol{F}}(v)| \leq C |v|_{H^1}^2,$$

where C does not depend on v.

Solution:

$$\begin{aligned}
|\widetilde{\boldsymbol{F}}(v)| &= \left| \int_\Omega \left(\operatorname{grad} v \cdot \operatorname{grad} v^\top - \frac{1}{2} \|\operatorname{grad} v\|^2 \boldsymbol{I} \right) \operatorname{grad} \Psi \,\mathrm{d}\boldsymbol{x} \right| \\
&\leq \|\operatorname{grad} \Psi\| \int_\Omega \frac{3}{2} \|\operatorname{grad} v\|^2 \,\mathrm{d}\boldsymbol{x} \leq C(\Psi) |v|_{H^1}^2,
\end{aligned}$$

where $C(\Psi) = \frac{3}{2} \|\operatorname{grad} \Psi\|_{L^\infty}$.

(14.3j) Code the function

```
Dune::FieldVector<double,2> ForceStable(DofHandler const& dofh,
    GridView const& gv, Vector const& U)
```

(in the handout file `ForceStable.hpp`) that implements the force functional $\widetilde{\boldsymbol{F}}(u)$ from (14.3.3).

HINT: Use $\Psi = u^{\mathrm{ex}}$ (the exact solution from (14.2.4)) and the three-point local quadrature rule that relies on the midpoints of the edges of a triangle.

Solution: See Listing 14.10. The code first compute `T1`, `T2`, `T3` which contains $\boldsymbol{T}(b_N^i)(x_k), i = 1, 2, 3$ the values of the transformed shape function $\hat{b}^i$ evaluated at the queadrature points x_k. In a second step we compute `grad` which is the value of $\operatorname{grad} \Psi(x_k)$ for $\Psi = u^{\mathrm{ex}}$. To evaluate $\operatorname{grad} u^{\mathrm{ex}}(x_k)$, note that

$$\operatorname{grad} \log \|\boldsymbol{x}\| = \frac{1}{\|\boldsymbol{x}\|} \operatorname{grad} \|\boldsymbol{x}\| = \frac{\boldsymbol{x}}{\|\boldsymbol{x}\|^2}.$$

Listing 14.10: Implementation for `forcestable`.

```
function Ft = forcestable(Mesh, u)

    nElements = size(Mesh.Elements,1);

    grad_N = grad_shap_LFE([0.5 0; 0.5 0.5; 0 0.5]);

    Ft = [0; 0];

    for i= 1:nElements

        vidx = Mesh.Elements(i,:);

        bK = Mesh.Coordinates(vidx(1),:);
        BK = [Mesh.Coordinates(vidx(2),:)-bK;
           Mesh.Coordinates(vidx(3),:)-bK];
        inv_BK = inv(BK);
        det_BK = abs(det(BK));

        x = [0.5 0; 0.5 0.5; 0 0.5]*BK+ones(3,1)*bK;

        grad_u_FE = (u(vidx(1))*grad_N(:,1:2)+ ...
                     u(vidx(2))*grad_N(:,3:4)+ ...
                     u(vidx(3))*grad_N(:,5:6))*transpose(inv_BK);

        T1 = grad_u_FE(1,:)' * grad_u_FE(1,:) -
           0.5*norm(grad_u_FE(1,:))^2*eye(2);
        T2 = grad_u_FE(2,:)' * grad_u_FE(2,:) -
           0.5*norm(grad_u_FE(2,:))^2*eye(2);
        T3 = grad_u_FE(3,:)' * grad_u_FE(3,:) -
           0.5*norm(grad_u_FE(3,:))^2*eye(2);

        nrm = @(x) sum(x.^2,1);
        ext = @(x) repmat(x,2,1);
        xa = @(x) (x+repmat([16/15;0],1,size(x,2)));
        xb = @(x) (x+repmat([1/15;0],1,size(x,2)));
        grad = @(x) (xa(x')./ext(nrm(xa(x'))) -
           xb(x')./ext(nrm(xb(x'))))'/log(2);

        loc = (T1*grad(x(1,:))' + T2*grad(x(2,:))' +
           T3*grad(x(3,:))') * det_BK / 6;
        Ft = Ft + loc;

    end

    Ft = Ft';

```

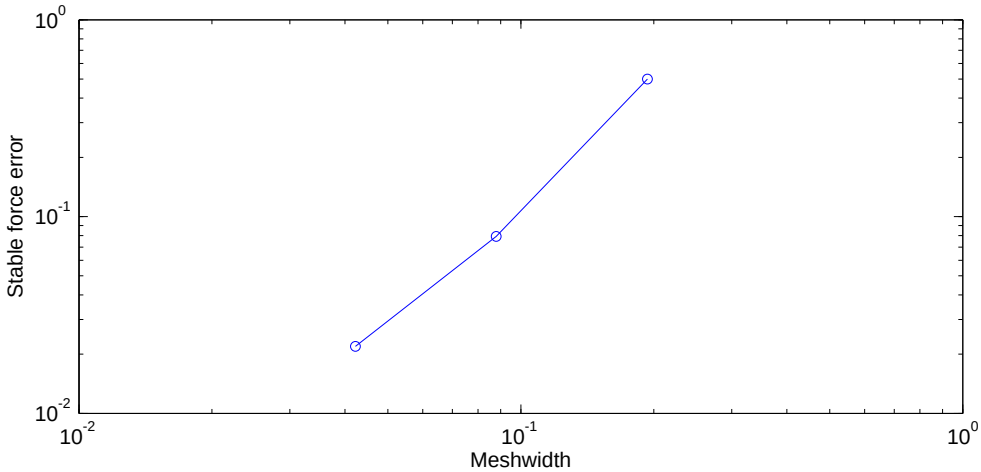

Figure 14.7: Convergence of the stabilized computation for the Force.

```
end
```

(14.3k) Compute and plot the errors $\widetilde{\boldsymbol{F}}(u_N) - \boldsymbol{F}(u)$ for the given meshes, describe the observed convergence in terms of the meshwidth and cpompare with subproblem (14.3d).

HINT: Again, modify the file `main.cc` from Problem 14.2.

Solution:

The implementation in Listing 14.9 is based on the previous implementation for (14.2d). In Figure 14.7 we see a significant improvement over Figure 14.6 (notice the axis). The convergence is

```
>> main3
convergence rate in terms of meshwidth: 2.059620
```

again algebraic, and the reported rate is close to 2 in terms of meshwidth, which is 1 in terms of number of degrees of freedom, and that is the same as the reported rate for the L^2-error of the finite element solution.

Listing 14.11: Implementation for (14.3k).

```
clear all;

n = zeros(1,3);
h = zeros(1,3);
f = zeros(3,2);
fs = zeros(3,2);

for l = 1:3

    Mesh = load_Mesh(['coords_' num2str(l) '.dat'], ['elems_'
       num2str(l) '.dat']);
    Mesh.ElemFlag = ones(size(Mesh.Elements,1),1);
```

```
    Mesh = add_Edges(Mesh);
    Mesh = bdflags(Mesh);
    Mesh = add_Edge2Elem(Mesh);

    A = assemMat_LFE(Mesh, @STIMA_Lapl_LFE);
    L = zeros(size(Mesh.Coordinates,1),1);

    dir_inner = @(x,varargin) ones(size(x,1),1);
    dir_outer = @(x,varargin) zeros(size(x,1),1);
    [U1, Free1] = assemDir_LFE(Mesh, -1, dir_inner);
    [U2, Free2] = assemDir_LFE(Mesh, -2, dir_outer);
    U = U1 + U2;
    Free = intersect(Free1, Free2);

    L = L - A*U;
    U(Free) = A(Free,Free)\L(Free);

    nrm = @(x) sqrt(sum(x.^2,1));
    func = @(x,varargin)
       ((log(nrm(x'+repmat([16/15;0],1,size(x',2)))) -
       log(nrm(x'+repmat([1/15;0],1,size(x',2)))))/log(2) -
       1)';

    n(l) = size(Mesh.Coordinates, 1);
    h(l) = get_MeshWidth(Mesh);
    f(l,:) = force(Mesh, U, -1);
    fs(l,:) = forcestable(Mesh, U);
end

ef = exactforce;
fs_err = sqrt(sum((fs-repmat(ef',3,1)).^2,2))';
p = polyfit(log(h), log(fs_err), 1);
fprintf('convergence rate in terms of meshwidth: %f\n', p(1))
figure(1);
loglog(h, fs_err, 'o-');
xlabel('Meshwidth');
ylabel('Stable force error');

conf.myFontsize = 13;
conf.size = [700, 300];
conf.its = 1:1;
fig_prepare(conf); legend off
saveas(gcf, 'main3.eps', 'epsc')
```

Problem 14.4 Least-Squares Galerkin Discretization

On a bounded polygon $\Omega \subset \mathbb{R}^2$ we consider the stationary linear advection problem

$$\begin{aligned} \boldsymbol{v}(\boldsymbol{x}) \cdot \operatorname{grad} u &= f \quad \text{in } \Omega, \\ u &= g \quad \text{on } \Gamma_{\text{in}} := \{\boldsymbol{x} \in \partial\Omega \,|\, \boldsymbol{v}(\boldsymbol{x}) \cdot \boldsymbol{n} < 0\}, \end{aligned} \tag{14.4.1}$$

where $\mathbf{v} : \overline{\Omega} \mapsto \mathbb{R}^2$ is a given continuous velocity field, $f \in \mathcal{C}^0(\overline{\Omega})$ a source term, and $g \in \mathcal{C}^0(\overline{\Gamma}_{\text{in}})$ boundary values for the unknown u on the inflow boundary Γ_{in}.

The so-called *least squares variational formulation* of (14.4.1) boils down to a linear variational problem

$$u \in V : \quad \mathsf{a}(u, w) = \ell(w) \quad \forall w \in V, \tag{14.4.2}$$

with

$$\mathsf{a}(u, w) := \langle \boldsymbol{v} \cdot \operatorname{grad} u, \boldsymbol{v} \cdot \operatorname{grad} w \rangle_{L^2}, \quad \ell(w) := \langle \boldsymbol{v} \cdot \operatorname{grad} w, f \rangle_{L^2}. \tag{14.4.3}$$

(14.4a) Specify an appropriate function space V for the least squares variational formulation.

HINT: The Dirichlet boundary conditions in (14.4.1) should be treated as *essential boundary conditions*.

Solution: The function space must consist of those functions u which are differentiable, and for which the energy norm is finite.

$$V = \{u \,|\, \mathsf{a}(u, u) < \infty, u = g \text{ on } \Gamma_{\text{in}}\} = \left\{u \,\middle|\, \boldsymbol{v} \cdot \operatorname{grad} u \in L^2(\Omega), u = g \text{ on } \Gamma_{\text{in}}\right\}.$$

(14.4b) The least squares variational formulation (14.4.2) is equivalent to a minimization problem for a functional J of the form

$$J(u) := \|T(u, f)\|^2_{L^2(\Omega)}, \tag{14.4.4}$$

where T is an expression involving the functions u and f. What is $T(u, f)$ in concrete terms.

Solution: The expression for T is

$$T(u, f) = \boldsymbol{v} \cdot \operatorname{grad} u - f.$$

(14.4c) Consider the linear 2nd-order scalar elliptic boundary value problem

$$\begin{aligned} -\operatorname{div}(\mathbf{A}(\boldsymbol{x}) \operatorname{grad} u) &= f \quad \text{in } \Omega, \\ u &= g \quad \text{on } \Gamma_{\text{in}}, \\ (\mathbf{A}(\boldsymbol{x}) \operatorname{grad} u) \cdot \boldsymbol{n} &= 0 \quad \text{on } \partial\Omega \setminus \Gamma_{\text{in}}, \end{aligned} \tag{14.4.5}$$

where $\mathbf{A} : \overline{\Omega} \mapsto \mathbb{R}^{2\times 2}$ is a continuous matrix-valued function with $\mathbf{A}(\boldsymbol{x}) = \mathbf{A}(\boldsymbol{x})^\top$ for all $\boldsymbol{x} \in \Omega$. Which choice of $\mathbf{A}$ makes the bilinear forms of the *standard* (i.e. not least squares) variational formulation of (14.4.5) and the variational problem (14.4.2) agree?

HINT: For the standard variational formulation of (14.4.5), see [NPDE, Eq. (2.4.5)].

Solution: The bilinear form for (14.4.5) is

$$\int_\Omega \operatorname{grad} w^\top \mathbf{A}(\boldsymbol{x}) \operatorname{grad} u \, \mathrm{d}\boldsymbol{x}$$

If we set that integrand equal to the integrand from the previous bilinear form we get the identity

$$\operatorname{grad} w^\top \mathbf{A}(\boldsymbol{x}) \operatorname{grad} u = (\boldsymbol{v}(\boldsymbol{x}) \cdot \operatorname{grad} w)(\boldsymbol{v}(\boldsymbol{x}) \cdot \operatorname{grad} u) = \operatorname{grad} w^\top \boldsymbol{v}(\boldsymbol{x})\boldsymbol{v}(\boldsymbol{x})^\top \operatorname{grad} u,$$

so we require $\mathbf{A}(\boldsymbol{x}) = \boldsymbol{v}(\boldsymbol{x})\boldsymbol{v}(\boldsymbol{x})^\top$, assuming here of course that $\boldsymbol{v}$ is a column vector.

(14.4d) Implement the class LocalLaplace which provides a method

```
template <class Element>
void operator()(Element const& e, ElementMatrix &local) const
```

to compute the element matrix associated to the bilinear form for (14.4.5) using linear Lagrangian finite elements and Dune::QuadratureRule<calc_t, elem_dim>.

HINT: Implementation of `LocalMass` in subproblem (8.1e) and `LocalLaplaceFromVector` in subproblem (8.2c) might be useful.

HINT: Keep in mind the constructor LocalLaplaceC(Function const& q) takes a function `q` in procedural form which returns a 2×2-matrix.

Solution: See Listing 14.12 for the code.

Listing 14.12: Implementation of `LocalLaplace`

```
#ifndef LOCALLAPLACE_HPP_
#define LOCALLAPLACE_HPP_

#include <dune/localfunctions/lagrange/pk.hh>
#include <dune/geometry/quadraturerules.hh>
#include <dune/common/fmatrix.hh>

namespace NPDE15{

  template <class Function>
  class LocalLaplaceC{
  public:
    using calc_t = double;
    using ElementMatrix = typename Dune::FieldMatrix<calc_t,3,3>;

    LocalLaplaceC(Function const& q) : q_(q) {};

    template <class Element>
    void operator()(Element const& e, ElementMatrix &local) const{
      const int world_dim = Element::dimension;
      const int elem_dim  = Element::mydimension;
      typedef typename Dune::QuadratureRule<calc_t, elem_dim> QuadRule_t;
      typedef typename Dune::QuadratureRules<calc_t, elem_dim> QuadRules;
      const QuadRule_t & quadRule = QuadRules::rule(e.type(), 3);
      Dune::PkLocalFiniteElement<calc_t,calc_t,elem_dim, 1> localFE;
      local = 0.0;

      auto const& egeom = e.geometry();
      for (auto qr : quadRule){
```

```
      auto const& local_pos=qr.position();
      // jacobian inverse transposed for transformation rule
      auto &jacInvTransp = egeom.jacobianInverseTransposed(local_pos);
      std::vector<Dune::FieldMatrix<calc_t,1,world_dim>> ref_gradients;
      // gradients on reference element evaluated at the quad points
      localFE.localBasis().evaluateJacobian(local_pos, ref_gradients);

      std::vector<Dune::FieldVector<calc_t,world_dim> >
          gradients(ref_gradients.size());
      // transform reference gradients to real element gradients:
      for (unsigned i=0;i<gradients.size();++i)
        jacInvTransp.mv(ref_gradients[i][0], gradients[i]);
      // determinant of transformation from reference element
      double jac_det = egeom.integrationElement(local_pos);
      // evaluate coefficient matrix
      auto A = q_(egeom.global(local_pos));
      // add local contributions
      for (unsigned i=0;i<local.N();++i){
        for (unsigned j=0;j<local.M();++j){
          Dune::FieldVector<calc_t,world_dim> y;
          y = 0; A.mv(gradients[j],y);
          local[i][j]+=(gradients[i]*y)*qr.weight()*jac_det;
        }
      }

       }
     }
   private:
     Function const& q_;
   };

 template <class Function>
 LocalLaplaceC<Function> LocalLaplace(Function const& q){
   return LocalLaplaceC<Function>(q);
  }

}
#endif
```

(14.4e) In `main.cc` write a method

```
template < class VectorFunction, class Function >
  void solveAdvBVP(DofHandler const& dofh, VectorFunction const& v,
      Function const& g, Vector & U)
```

that solves (14.4.1) in the case $f \equiv 0$ by means of the least squares Galerkin approach based on the variational formulation (14.4.2) and piecewise linear Lagrangian finite elements. The argument `v` provides the velocity field in procedural form. This function should return a column vector $\in \mathbb{R}^2$. The `g`-argument is also given in procedural form and passes the real valued function g. The Vector `U` is filled with the obtained solution.

Solution: See Listing 14.13 for the code.

Listing 14.13: Implementation of `solveAdvBVP()`

```
template <class VectorFunction, class Function>
```

```
void solveAdvBVP( DofHandler & dofh, VectorFunction const& v,
                  Function const& g, Vector & U){
      unsigned N = dofh.size();
      std::cout << "Solving for N =" << N << " unknowns.\n";

      // Get boundary nodes
      IndexVector dirichlet_dofs(N);
      NPDE15::LBoundaryNodes<DofHandler> get_bnd_dofs(dofh);
      get_bnd_dofs(dirichlet_dofs);
      dofh.set_inactive(dirichlet_dofs);

      Vector Phi(N); Phi.setZero();
      // Non-Homogeneous Dirichlet data
      NPDE15::VectorAssembler<DofHandler> vecAssembler(dofh);
      Vector G(N); G.setZero();
      // loop over cells
      for (auto it=dofh.gv.template begin<0>();it!=dofh.gv.template
          end<0>();++it){
     auto const& e=*it;
     auto egeom = e.geometry();
     for (unsigned i=0;i<3;++i){
       unsigned loctoglob = dofh(e, i);
       if (!dofh.active(loctoglob))
         G[loctoglob] = g(egeom.corner(i));
     }
      }
      vecAssembler.set_inactive(Phi, G);

      auto a = [&v](Coordinate const x) {
     Dune::FieldMatrix<calc_t, 2, 2> Ax;
     auto vV = v(x);
     for(int i=0; i<2; i++)
       for(int j=0; j<2; j++)
         Ax[i][j] = vV[i]*vV[j];

     return Ax;};

      // assemble the system matrix
      std::vector<Eigen::Triplet<calc_t>> triplets;
      NPDE15::MatrixAssembler<DofHandler> matAssembler(dofh);
      matAssembler(triplets, NPDE15::LocalLaplace(a));
      matAssembler.set_inactive(triplets);

      Matrix A(N, N);
      A.setFromTriplets(triplets.begin(), triplets.end());
      A.makeCompressed();

      // solution vector U
      U.setZero();
      // solve the system
      U = Phi/A; // short-hand, see Pardiso.hpp for more information

};
```

(14.4f) Complete the class

```
template <class Function, class VectorFunction>
LocalFunctionLSQ(Function const& f, VectorFunction const& v)
```

by writing the method

```
template <class Element, class Vector>
void operator()(Element const& e, Vector &local) const
```

that computes the right-hand side vector for the variational problem (14.4.2), when piecewise linear Lagrangian finite elements are employed for its Galerkin discretization.

The arguments `v` and `f` in the constructor, give the velocity field $\mathbf{v}$ and source term f in procedural form, see subproblem (14.4e). Vertex based quadrature (2D trapezoidal rule) is to be used for local computations.

HINT: Base your implementation on `LocalFunction`.

Solution: See Listing 14.14 for the code.

Listing 14.14: Implementation of `LocalFunctionLSQ`

```
#ifndef LOCALFUNCTIONLSQ_HPP_
#define LOCALFUNCTIONLSQ_HPP_

#include <stdexcept>
#include <vector>
#include <dune/localfunctions/lagrange/pk.hh>
#include <dune/geometry/quadraturerules.hh>
#include <dune/common/exceptions.hh>
#include <dune/common/fvector.hh>

namespace NPDE15{

  template <class Function, class VectorFunction>
  class LocalFunctionLSQC{
  public:
    using calc_t=double;

    LocalFunctionLSQC(Function const& f, VectorFunction const& v)
      : f_(f), v_(v) {};

    template <class Vector, class Element>
    void operator()(Element const& e, Vector &local) const;
  private:
    Function const& f_;
    VectorFunction const& v_;
  };

  template <class Function, class VectorFunction>
  LocalFunctionLSQC<Function, VectorFunction> LocalFunctionLSQ(Function const& f,
                                                VectorFunction const& v){
    return LocalFunctionLSQC<Function, VectorFunction>(f, v);
  }

  template <class Function, class VectorFunction>
  template <class Vector, class Element>
```

```
  void LocalFunctionLSQC<Function, VectorFunction>::operator()(Element
      const& e, Vector &local) const{
    const int world_dim = Element::dimension;
    const int elem_dim  = Element::mydimension;
    typedef typename Dune::QuadratureRule<calc_t, elem_dim> QuadRule_t;
    typedef typename Dune::QuadratureRules<calc_t, elem_dim> QuadRules;
    const QuadRule_t & quadRule = QuadRules::rule(e.type(), 3);
    Dune::PkLocalFiniteElement<calc_t,calc_t,elem_dim, 1> localFE;
    unsigned M=localFE.localBasis().size();
    local = Vector(M);

    auto const& egeom = e.geometry();
    for (auto qr : quadRule){
      auto const& local_pos=qr.position();
      // jacobian inverse transposed for transformation rule
      auto &jacInvTransp = egeom.jacobianInverseTransposed(local_pos);
      std::vector<Dune::FieldMatrix<calc_t,1,world_dim>> ref_gradients;
      // gradients on reference element evaluated at the quad points
      localFE.localBasis().evaluateJacobian(local_pos, ref_gradients);

      std::vector<Dune::FieldVector<calc_t,world_dim> >
          gradients(ref_gradients.size());
      // transform reference gradients to real element gradients:
      for (unsigned i=0;i<gradients.size();++i)
     jacInvTransp.mv(ref_gradients[i][0], gradients[i]);
      // determinant of transformation from reference element
      double jac_det = egeom.integrationElement(local_pos);
      // evaluate coefficient
      double val_f = f_(egeom.global(local_pos));
      auto val_v = v_(egeom.global(local_pos));
      // add local contributions
      for (unsigned i=0;i<3;++i){
     local[i] = (val_v*gradients[i])*val_f*qr.weight()*jac_det;
      }
    }
  };

}
#endif
```

(14.4g) Assume $g = 0$. Code a method

```
template < class VectorFunction, class Function >
  void solveAdvBVP_LSQ(DofHandler const& dofh, VectorFunction
      const& v, Vector & U)
```

in `main.cc`, that computes the coefficient vector `U` of the least squares solution of (14.4.1) obtained by a linear Lagrangian finite element Galerkin solution of the related least squares variational problem (14.4.2). The arguments have the same meaning as in subproblem (14.4f).

HINT: You may copy large parts of your implementation of `solveAdvBVP` from subproblem (14.4e). Also use `LocalFunctionLSQ`.

HINT: For debugging, you may found the output `test_call_out.txt` corresponding to the given file `main.cc`.

Solution: See Listing 14.15 for the code.

Listing 14.15: Implementation of `solveAdvBVP_LSQ()`

```
template <class VectorFunction, class Function>
void solveAdvBVP_LSQ( DofHandler & dofh, VectorFunction const& v,
                      Function const& f, Vector & U){
      unsigned N = dofh.size();
      std::cout << "Solving for N =" << N << " unknowns.\n";

      // Get boundary nodes
      IndexVector dirichlet_dofs(N);
      NPDE15::LBoundaryNodes<DofHandler> get_bnd_dofs(dofh);
      get_bnd_dofs(dirichlet_dofs);
      dofh.set_inactive(dirichlet_dofs);

      // Homogeneous Dirichlet data
      Vector G(N); G.setZero();

      // assemble rhs and set dirichlet dofs to dirichlet data
      Vector Phi(N); Phi.setZero();tor
      NPDE15::VectorAssembler<DofHandler> vecAssembler(dofh);
      vecAssembler(Phi, NPDE15::LocalFunctionLSQ(f,v));
      vecAssembler.set_inactive(Phi, G);

      auto a = [&v](Coordinate const x) {
     Dune::FieldMatrix<calc_t, 2, 2> Ax;
     auto vV = v(x);
     for(int i=0; i<2; i++)
       for(int j=0; j<2; j++)
         Ax[i][j] = vV[i]*vV[j];

     return Ax;};

      // assemble the system matrix
      std::vector<Eigen::Triplet<calc_t>> triplets;
      NPDE15::MatrixAssembler<DofHandler> matAssembler(dofh);
      matAssembler(triplets, NPDE15::LocalLaplace(a));
      matAssembler.set_inactive(triplets);

      Matrix A(N, N);
      A.setFromTriplets(triplets.begin(), triplets.end());
      A.makeCompressed();

      // solution vector U
      U.setZero();
      // solve the system
      U = Phi/A; // short-hand, see Pardiso.hpp for more information

}
```

References

[NPDE] Lecture Slides for the course "Numerical Methods for Partial Differential Equa-

tions”.SVN revision # 79326.

[NCSE] Lecture Slides for the course “Numerical Methods for CSE”.

[LehrFEM] LehrFEM manual.

Last modified on August 6, 2015